Chamber complexes, Coxeter systems, and buildings

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

March 17, 2025

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

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 0 1 SucSuc*]:
assumes *0*: $n = 0 \Longrightarrow P$
and *1*: $n = 1 \Longrightarrow P$
and *SucSuc*: $\bigwedge m.\ n = Suc\ (Suc\ m) \Longrightarrow P$
shows *P*
⟨*proof*⟩

lemma *nat-even-induct* [*case-names - 0 SucSuc*]:
assumes *even*: *even n*
and *0*: *P 0*
and *SucSuc*: $\bigwedge m.\ even\ m \Longrightarrow P\ m \Longrightarrow P\ (Suc\ (Suc\ m))$
shows *P n*
⟨*proof*⟩

lemma *nat-induct-step2* [*case-names 0 1 SucSuc*]:
assumes *0*: *P 0*
and *1*: *P 1*
and *SucSuc*: $\bigwedge m.\ P\ m \Longrightarrow P\ (Suc\ (Suc\ m))$
shows *P n*
⟨*proof*⟩

1.2 Logic

lemma *ex1-unique*: $\exists !x.\ P\ x \Longrightarrow P\ a \Longrightarrow P\ b \Longrightarrow a{=}b$
⟨*proof*⟩

lemma *not-the1*:

assumes $\exists!x.\ P\ x\ y \neq (THE\ x.\ P\ x)$
shows $\neg\ P\ y$
⟨*proof*⟩

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 *neither*: $\neg P \Longrightarrow \neg Q \Longrightarrow R$
shows R
⟨*proof*⟩

1.3 Sets

lemma *bex1-equality*: $[\![\ \exists!x{\in}A.\ P\ x;\ x{\in}A;\ P\ x;\ y{\in}A;\ P\ y\ ]\!] \Longrightarrow x{=}y$
⟨*proof*⟩

lemma *prod-ballI*: $(\bigwedge a\ b.\ (a,b){\in}A \Longrightarrow P\ a\ b) \Longrightarrow \forall(a,b){\in}A.\ P\ a\ b$
⟨*proof*⟩

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

lemma *set-decomp-subset*:
$[\![\ U = A{\cup}B;\ A{\subseteq}X;\ B{\subseteq}Y;\ X{\subseteq}U;\ X{\cap}Y = \{\}\ ]\!] \Longrightarrow A = X$
⟨*proof*⟩

lemma *insert-subset-equality*: $[\![\ a{\notin}A;\ a{\notin}B;\ insert\ a\ A = insert\ a\ B\ ]\!] \Longrightarrow A{=}B$
⟨*proof*⟩

lemma *insert-compare-element*: $a{\notin}A \Longrightarrow insert\ b\ A = insert\ a\ A \Longrightarrow b{=}a$
⟨*proof*⟩

lemma *card1*:
assumes $card\ A = 1$
shows $\exists a.\ A = \{a\}$
⟨*proof*⟩

lemma *singleton-pow*: $a{\in}A \Longrightarrow \{a\}{\in}Pow\ A$
⟨*proof*⟩

definition *separated-by* :: $'a\ set\ set \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$
where $separated\text{-}by\ w\ x\ y \equiv \exists A\ B.\ w{=}\{A,B\} \wedge x{\in}A \wedge y{\in}B$

lemma *separated-byI*: $x{\in}A \Longrightarrow y{\in}B \Longrightarrow separated\text{-}by\ \{A,B\}\ x\ y$
⟨*proof*⟩

lemma *separated-by-disjoint*: $[\![\ separated\text{-}by\ \{A,B\}\ x\ y;\ A{\cap}B{=}\{\};\ x{\in}A\ ]\!] \Longrightarrow y{\in}B$
⟨*proof*⟩

lemma *separated-by-in-other*: *separated-by* $\{A,B\}$ x $y \Longrightarrow x \notin A \Longrightarrow x \in B \wedge y \in A$
⟨*proof*⟩

lemma *separated-by-not-empty*: *separated-by* w x $y \Longrightarrow w \neq \{\}$
⟨*proof*⟩

lemma *not-self-separated-by-disjoint*: $A \cap B = \{\} \Longrightarrow \neg$ *separated-by* $\{A,B\}$ x x
⟨*proof*⟩

1.4 Functions and relations

1.4.1 Miscellaneous

lemma *cong-let*: (*let* $x = y$ *in* f x) $= f$ y ⟨*proof*⟩

lemma *sym-sym*: *sym* $(A \times A)$ ⟨*proof*⟩

lemma *trans-sym*: *trans* $(A \times A)$ ⟨*proof*⟩

lemma *map-prod-sym*: *sym* $A \Longrightarrow$ *sym* (*map-prod* f f ‘ A)
⟨*proof*⟩

abbreviation *restrict1* :: $('a \Rightarrow 'a) \Rightarrow 'a$ *set* $\Rightarrow ('a \Rightarrow 'a)$
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 ‘ $B = f$‘B
⟨*proof*⟩

1.4.2 Equality of functions restricted to a set

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

lemma *fun-eq-onI*: $(\bigwedge a.\ a \in A \Longrightarrow f\ a = g\ a) \Longrightarrow$ *fun-eq-on* f g A
⟨*proof*⟩

lemma *fun-eq-onD*: *fun-eq-on* f g $A \Longrightarrow a \in A \Longrightarrow f\ a = g\ a$
⟨*proof*⟩

lemma *fun-eq-on-UNIV*: (*fun-eq-on* f g *UNIV*) $= (f = g)$
⟨*proof*⟩

lemma *fun-eq-on-subset*: *fun-eq-on* f g $A \Longrightarrow B \subseteq A \Longrightarrow$ *fun-eq-on* f g B
⟨*proof*⟩

lemma *fun-eq-on-sym*: *fun-eq-on* f g $A \Longrightarrow$ *fun-eq-on* g f A
⟨*proof*⟩

lemma *fun-eq-on-trans*: *fun-eq-on* f g $A \Longrightarrow$ *fun-eq-on* g h $A \Longrightarrow$ *fun-eq-on* f h A
⟨*proof*⟩

lemma *fun-eq-on-cong*: *fun-eq-on f h A* ⟹ *fun-eq-on g h A* ⟹ *fun-eq-on f g A*
⟨*proof*⟩

lemma *fun-eq-on-im* : *fun-eq-on f g A* ⟹ *B*⊆*A* ⟹ *f*‘*B* = *g*‘*B*
⟨*proof*⟩

lemma *fun-eq-on-subset-and-diff-imp-eq-on*:
assumes *A*⊆*B* *fun-eq-on f g A* *fun-eq-on f g* (*B*−*A*)
shows *fun-eq-on f g B*
⟨*proof*⟩

lemma *fun-eq-on-set-and-comp-imp-eq*:
fun-eq-on f g A ⟹ *fun-eq-on f g* (−*A*) ⟹ *f* = *g*
⟨*proof*⟩

lemma *fun-eq-on-bij-betw*: *fun-eq-on f g A* ⟹ *bij-betw f A B* = *bij-betw g A B*
⟨*proof*⟩

lemma *fun-eq-on-restrict1*: *fun-eq-on* (*restrict1 f A*) *f A*
⟨*proof*⟩

abbreviation *fixespointwise f A* ≡ *fun-eq-on f id A*

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

lemmas *fixespointwise-subset-and-diff-imp-eq-on* =
fun-eq-on-subset-and-diff-imp-eq-on[*of* - - - *id*]

lemma *id-fixespointwise*: *fixespointwise id A*
⟨*proof*⟩

lemma *fixespointwise-im*: *fixespointwise f A* ⟹ *B*⊆*A* ⟹ *f*‘*B* = *B*
⟨*proof*⟩

lemma *fixespointwise-comp*:
fixespointwise f A ⟹ *fixespointwise g A* ⟹ *fixespointwise* (*g*∘*f*) *A*
⟨*proof*⟩

lemma *fixespointwise-insert*:
assumes *fixespointwise f A* *f* ‘ (*insert a A*) = *insert a A*
shows *fixespointwise f* (*insert a A*)
⟨*proof*⟩

lemma *fixespointwise-restrict1*:
fixespointwise f A ⟹ *fixespointwise* (*restrict1 f B*) *A*

⟨*proof*⟩

lemma *fold-fixespointwise*:
$\forall x{\in}set\ xs.\ fixespointwise\ (f\ x)\ A \Longrightarrow fixespointwise\ (fold\ f\ xs)\ A$
⟨*proof*⟩

lemma *funpower-fixespointwise*:
assumes *fixespointwise f A*
shows $fixespointwise\ (f\frown\frown n)\ A$
⟨*proof*⟩

1.4.3 Injectivity, surjectivity, bijectivity, and inverses

lemma *inj-on-to-singleton*:
assumes *inj-on f A* $f`A = \{b\}$
shows $\exists\, a.\ A = \{a\}$
⟨*proof*⟩

lemmas $inj\text{-}inj\text{-}on = subset\text{-}inj\text{-}on[of\ \text{-}\ UNIV,\ OF\ \text{-}\ subset\text{-}UNIV]$

lemma *inj-on-eq-image'*: $[\![\ inj\text{-}on\ f\ A;\ X{\subseteq}A;\ Y{\subseteq}A;\ f`X{\subseteq}f`Y\ ]\!] \Longrightarrow X{\subseteq}Y$
⟨*proof*⟩

lemma *inj-on-eq-image*: $[\![\ inj\text{-}on\ f\ A;\ X{\subseteq}A;\ Y{\subseteq}A;\ f`X{=}f`Y\ ]\!] \Longrightarrow X{=}Y$
⟨*proof*⟩

lemmas $inj\text{-}eq\text{-}image = inj\text{-}on\text{-}eq\text{-}image[OF\ \text{-}\ subset\text{-}UNIV\ subset\text{-}UNIV]$

lemma *induced-pow-fun-inj-on*:
assumes *inj-on f A*
shows $inj\text{-}on\ ((`)\ f)\ (Pow\ A)$
⟨*proof*⟩

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

lemma *induced-pow-fun-surj*:
$((`)\ f)\ `\ (Pow\ A) = Pow\ (f`A)$
⟨*proof*⟩

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

lemma *bij-betw-the-inv-into-onto*: $bij\text{-}betw\ f\ A\ B \Longrightarrow the\text{-}inv\text{-}into\ A\ f\ `\ B = A$
⟨*proof*⟩

lemma *bij-betw-imp-bij-betw-Pow*:
  assumes *bij-betw f A B*
  shows *bij-betw ((‘) f) (Pow A) (Pow B)*
  ⟨*proof*⟩

lemma *comps-fixpointwise-imp-bij-betw*:
  assumes *f‘X⊆Y g‘Y⊆X fixespointwise (g∘f) X fixespointwise (f∘g) Y*
  shows *bij-betw f X Y*
  ⟨*proof*⟩

lemma *set-permutation-bij-restrict1*:
  assumes *bij-betw f A A*
  shows *bij (restrict1 f A)*
⟨*proof*⟩

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

lemma *the-inv-into-the-inv-into*:
  inj-on f A ⟹ a∈A ⟹ the-inv-into (f‘A) (the-inv-into A f) a = f a
  ⟨*proof*⟩

lemma *the-inv-into-f-im-f-im*:
  assumes *inj-on f A x⊆A*
  shows *the-inv-into A f ‘ f ‘ x = x*
  ⟨*proof*⟩

lemma *f-im-the-inv-into-f-im*:
  assumes *inj-on f A x⊆f‘A*
  shows *f ‘ the-inv-into A f ‘ x = x*
  ⟨*proof*⟩

lemma *the-inv-leftinv*: *bij f ⟹ the-inv f ∘ f = id*
  ⟨*proof*⟩

1.4.4 Induced functions on sets of sets and lists of sets

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

abbreviation *setsetmapim* :: *('a⇒'b) ⇒ 'a set set ⇒ 'b set set* (**infix** ‹⊢› *70*)
  where *f⊢X ≡ ((‘) f) ‘ X*

abbreviation *setlistmapim* :: *('a⇒'b) ⇒ 'a set list ⇒ 'b set list* (**infix** ‹⊨› *70*)
  where *f⊨Xs ≡ map ((‘) f) Xs*

lemma *setsetmapim-comp*: *(f∘g)⊢A = f⊢(g⊢A)*
  ⟨*proof*⟩

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

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

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

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

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

1.4.5 Induced functions on quotients

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

lemma *respects-imp-unique-image-rel*: *f respects r* $\Longrightarrow y \in f`r``\{a\} \Longrightarrow y = f\ a$
⟨*proof*⟩

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

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

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

lemma *quotientfun-classrep-equality*:
⟦ *refl-on A r*; *f respects r*; $a \in A$ ⟧ $\Longrightarrow$ *quotientfun f* $(r``\{a\}) = f\ a$
⟨*proof*⟩

1.4.6 Support of a function

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

lemma *suppI-contra*: $x \notin supp\ f \Longrightarrow f\ x = 0$
⟨*proof*⟩

lemma *suppD-contra*: $f\ x = 0 \Longrightarrow x \notin supp\ f$
⟨*proof*⟩

abbreviation *restrict0* :: $('a \Rightarrow 'b::zero) \Rightarrow 'a\ set \Rightarrow ('a \Rightarrow 'b)$
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*⟩

1.5 Lists

1.5.1 Miscellaneous facts

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

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

lemma *distinct-count-list*:
$distinct\ xs \Longrightarrow count\text{-}list\ xs\ a = (if\ a \in set\ xs\ then\ 1\ else\ 0)$
⟨*proof*⟩

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

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

1.5.2 Cases

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

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

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

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

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

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

1.5.3 Induction

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

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

⟨*proof*⟩

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

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

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

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

1.5.4 Alternating lists

primrec *alternating-list* :: $nat \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a\ list$
where *zero*: *alternating-list* $0\ s\ t = []$
| *Suc* : *alternating-list* $(Suc\ k)\ s\ t =$
alternating-list $k\ s\ t$ @ [*if even* k *then* s *else* t]
— could be defined using Cons, but we want the alternating list to always start with the same letter as it grows, and it's easier to do that via append

lemma *alternating-list2*: *alternating-list* $2\ s\ t = [s,t]$
⟨*proof*⟩

lemma *length-alternating-list*: *length* (*alternating-list n s t*) = *n*
⟨*proof*⟩

lemma *alternating-list-Suc-Cons*:
alternating-list (*Suc k*) *s t* = *s* # *alternating-list k t s*
⟨*proof*⟩

lemma *alternating-list-SucSuc-ConsCons*:
alternating-list (*Suc* (*Suc k*)) *s t* = *s* # *t* # *alternating-list k s t*
⟨*proof*⟩

lemma *alternating-list-alternates*:
alternating-list n s t = *as*@[*a*,*b*,*c*]@*bs* ⟹ *a*=*c*
⟨*proof*⟩

lemma *alternating-list-split*:
alternating-list (*m*+*n*) *s t* = *alternating-list m s t* @
(*if even m then alternating-list n s t else alternating-list n t s*)
⟨*proof*⟩

lemma *alternating-list-append*:
even m ⟹
alternating-list m s t @ *alternating-list n s t* = *alternating-list* (*m*+*n*) *s t*
odd m ⟹
alternating-list m s t @ *alternating-list n t s* = *alternating-list* (*m*+*n*) *s t*
⟨*proof*⟩

lemma *rev-alternating-list*:
rev (*alternating-list n s t*) =
(*if even n then alternating-list n t s else alternating-list n s t*)
⟨*proof*⟩

lemma *set-alternating-list*: *set* (*alternating-list n s t*) ⊆ {*s*,*t*}
⟨*proof*⟩

lemma *set-alternating-list1*:
assumes *n* ≥ *1*
shows *s* ∈ *set* (*alternating-list n s t*)
⟨*proof*⟩

lemma *set-alternating-list2*:
n ≥ *2* ⟹ *set* (*alternating-list n s t*) = {*s*,*t*}
⟨*proof*⟩

lemma *alternating-list-in-lists*: *a*∈*A* ⟹ *b*∈*A* ⟹ *alternating-list n a b* ∈ *lists A*
⟨*proof*⟩

1.5.5 Binary relation chains

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

fun *binrelchain* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a list* ⇒ *bool*
  where *binrelchain P* [] = *True*
    | *binrelchain P* [*x*] = *True*
    | *binrelchain P* (*x* # *y* # *xs*) = (*P x y* ∧ *binrelchain P* (*y*#*xs*))

lemma *binrelchain-Cons-reduce*: *binrelchain P* (*x*#*xs*) ⟹ *binrelchain P xs*
  ⟨*proof*⟩

lemma *binrelchain-append-reduce1*: *binrelchain P* (*xs*@*ys*) ⟹ *binrelchain P xs*
⟨*proof*⟩

lemma *binrelchain-append-reduce2*:
  binrelchain P (*xs*@*ys*) ⟹ *binrelchain P ys*
⟨*proof*⟩

lemma *binrelchain-Conssnoc-reduce*:
  binrelchain P (*x*#*xs*@[*y*]) ⟹ *binrelchain P xs*
  ⟨*proof*⟩

lemma *binrelchain-overlap-join*:
  binrelchain P (*xs*@[*x*]) ⟹ *binrelchain P* (*x*#*ys*) ⟹ *binrelchain P* (*xs*@*x*#*ys*)
  ⟨*proof*⟩

lemma *binrelchain-join*:
  ⟦ *binrelchain P* (*xs*@[*x*]); *binrelchain P* (*y*#*ys*); *P x y* ⟧ ⟹
    binrelchain P (*xs* @ *x* # *y* # *ys*)
  ⟨*proof*⟩

lemma *binrelchain-snoc*:
  binrelchain P (*xs*@[*x*]) ⟹ *P x y* ⟹ *binrelchain P* (*xs*@[*x*,*y*])
  ⟨*proof*⟩

lemma *binrelchain-sym-rev*:
  assumes ⋀*x y*. *P x y* ⟹ *P y x*
  shows *binrelchain P xs* ⟹ *binrelchain P* (*rev xs*)
⟨*proof*⟩

lemma *binrelchain-remdup-adj*:
  binrelchain P (*xs*@[*x*,*x*]@*ys*) ⟹ *binrelchain P* (*xs*@*x*#*ys*)
  ⟨*proof*⟩

abbreviation *proper-binrelchain P xs* ≡ *binrelchain P xs* ∧ *distinct xs*

lemma *binrelchain-obtain-proper*:
  x≠*y* ⟹ *binrelchain P* (*x*#*xs*@[*y*]) ⟹

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

lemma *binrelchain-trans-Cons-snoc*:
assumes $\bigwedge x\ y\ z.\ P\ x\ y \Longrightarrow P\ y\ z \Longrightarrow P\ x\ z$
shows *binrelchain P* (*x#xs*@[*y*]) $\Longrightarrow$ *P x y*
⟨*proof*⟩

lemma *binrelchain-cong*:
assumes $\bigwedge x\ y.\ P\ x\ y \Longrightarrow Q\ x\ y$
shows *binrelchain P xs* $\Longrightarrow$ *binrelchain Q xs*
⟨*proof*⟩

lemma *binrelchain-funcong-Cons-snoc*:
assumes $\bigwedge x\ y.\ P\ x\ y \Longrightarrow f\ y = f\ x$ *binrelchain P* (*x#xs*@[*y*])
shows $f\ y = f\ x$
⟨*proof*⟩

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 *Q x* $\Longrightarrow$ *binrelchain P* (*x#zs*@[*y*]) $\Longrightarrow$ $f\ y = f\ x$
⟨*proof*⟩

lemma *binrelchain-setfuncong-Cons-snoc*:
⟦ $\forall x\in A.\ \forall y.\ P\ x\ y \longrightarrow y\in A$; $\forall x\in A.\ \forall y.\ P\ x\ y \longrightarrow f\ y = f\ x$; $x\in A$;
binrelchain P (*x#zs*@[*y*]) ⟧ $\Longrightarrow$ $f\ y = f\ x$
⟨*proof*⟩

lemma *binrelchain-propcong-Cons-snoc*:
assumes $\bigwedge x\ y.\ Q\ x \Longrightarrow P\ x\ y \Longrightarrow Q\ y$
shows *Q x* $\Longrightarrow$ *binrelchain P* (*x#xs*@[*y*]) $\Longrightarrow$ *Q y*
⟨*proof*⟩

1.5.6 Set of subseqs

lemma *subseqs-Cons*: *subseqs* (*x#xs*) = *map* (*Cons x*) (*subseqs xs*) @ (*subseqs xs*)
⟨*proof*⟩

abbreviation *ssubseqs xs* ≡ *set* (*subseqs xs*)

lemma *nil-ssubseqs*: [] ∈ *ssubseqs xs*
⟨*proof*⟩

lemma *ssubseqs-Cons*: *ssubseqs* (*x#xs*) = (*Cons x*) ‘ (*ssubseqs xs*) ∪ *ssubseqs xs*
⟨*proof*⟩

lemma *ssubseqs-refl*: *xs* ∈ *ssubseqs xs*
⟨*proof*⟩

lemma *ssubseqs-subset*: $as \in ssubseqs\ bs \Longrightarrow ssubseqs\ as \subseteq ssubseqs\ bs$
⟨*proof*⟩

lemma *ssubseqs-lists*:
$as \in lists\ A \Longrightarrow bs \in ssubseqs\ as \Longrightarrow bs \in lists\ A$
⟨*proof*⟩

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

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

1.6 Orders and posets

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

1.6.1 Morphisms of posets

locale *OrderingSetMap* =
  domain : *ordering less-eq less*
+ *codomain*: *ordering less-eq′ less′*
  for *less-eq* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** ‹≤› *50*)
  and *less* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** ‹<› *50*)
  and *less-eq′* :: $'b \Rightarrow 'b \Rightarrow bool$ (**infix** ‹≤∗› *50*)
  and *less′* :: $'b \Rightarrow 'b \Rightarrow bool$ (**infix** ‹<∗› *50*)
+ **fixes** P :: $'a\ set$
  and f :: $'a \Rightarrow 'b$
  assumes *ordsetmap*: $a \in P \Longrightarrow b \in P \Longrightarrow a \le b \Longrightarrow f\ a \le\ast f\ b$
begin

lemma *comp*:
  assumes *OrderingSetMap less-eq′ less′ less-eq″ less″ Q g*
    $f`P \subseteq Q$
  shows *OrderingSetMap less-eq less less-eq″ less″ P* $(g \circ f)$
⟨*proof*⟩

lemma *subset*: $Q \subseteq P \Longrightarrow$ *OrderingSetMap* $(\le)\ (<)\ (\le\ast)\ (<\ast)\ Q\ f$
⟨*proof*⟩

end

locale *OrderingSetIso* = *OrderingSetMap less-eq less less-eq′ less′ P f*
  for *less-eq* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** ‹≤› *50*)
  and *less* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** ‹<› *50*)

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

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

lemma (**in** *OrderingSetMap*) *isoI*:
assumes *inj-on f P* $\bigwedge a\ b.\ a\in P \Longrightarrow b\in P \Longrightarrow f\ a \le\!* f\ b \Longrightarrow a \le b$
shows *OrderingSetIso less-eq less less-eq' less' P f*
⟨*proof*⟩

lemma *OrderingSetIsoI-orders-greater2less*:
fixes *f* :: *'a::order* ⇒ *'b::order*
assumes *inj-on f P* $\bigwedge a\ b.\ a \in P \Longrightarrow b \in P \Longrightarrow (b\le a) = (f\ a \le f\ b)$
shows *OrderingSetIso* (*greater-eq::'a⇒'a⇒bool*) (*greater::'a⇒'a⇒bool*)
(*less-eq::'b⇒'b⇒bool*) (*less::'b⇒'b⇒bool*) *P f*
⟨*proof*⟩

context *OrderingSetIso*
begin

lemmas *ordsetmap* = *ordsetmap*

lemma *ordsetmap-strict*: ⟦ *a∈P*; *b∈P*; *a<b* ⟧ ⟹ *f a <∗ f b*
⟨*proof*⟩

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

lemma *rev-ordsetmap*: ⟦ *a∈P*; *b∈P*; *f a ≤∗ f b* ⟧ ⟹ *a ≤ b*
⟨*proof*⟩

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

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

lemma *rev-ordsetmap-strict*: ⟦ *a∈P*; *b∈P*; *f a <∗ f b* ⟧ ⟹ *a < b*
⟨*proof*⟩

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

lemma *iso-subset*:
  $Q \subseteq P \Longrightarrow$ *OrderingSetIso* $(\leq)$ $(<)$ $(\leq *)$ $(<*)$ Q f
  ⟨*proof*⟩

lemma *iso-dual*:
  ‹*OrderingSetIso* $(\lambda a\ b.$ *less-eq* $b\ a)$ $(\lambda a\ b.$ *less* $b\ a)$
   $(\lambda a\ b.$ *less-eq'* $b\ a)$ $(\lambda a\ b.$ *less'* $b\ a)$ P f›
  ⟨*proof*⟩

end

lemma *induced-pow-fun-subset-ordering-iso*:
  assumes *inj-on* f A
  shows *subset-ordering-iso* (*Pow* A) ((‘) f)
⟨*proof*⟩

1.6.2 More *arg-min*

lemma *is-arg-minI*:
  $[\![$ $P\ x$; $\bigwedge y.\ P\ y \Longrightarrow \neg\ m\ y < m\ x$ $]\!] \Longrightarrow$ *is-arg-min* $m\ P\ x$
⟨*proof*⟩

lemma *is-arg-min-linorderI*:
  $[\![$ $P\ x$; $\bigwedge y.\ P\ y \Longrightarrow m\ x \leq (m\ y$::-::*linorder*$)$ $]\!] \Longrightarrow$ *is-arg-min* $m\ P\ x$
⟨*proof*⟩

lemma *is-arg-min-eq*:
  $[\![$ *is-arg-min* $m\ P\ x$; $P\ z$; $m\ z = m\ x$ $]\!] \Longrightarrow$ *is-arg-min* $m\ P\ z$
⟨*proof*⟩

lemma *is-arg-minD1*: *is-arg-min* $m\ P\ x \Longrightarrow P\ x$
⟨*proof*⟩

lemma *is-arg-minD2*: *is-arg-min* $m\ P\ x \Longrightarrow P\ y \Longrightarrow \neg\ m\ y < m\ x$
⟨*proof*⟩

lemma *is-arg-min-size*: **fixes** m :: *'a* ⇒ *'b*::*linorder*
shows *is-arg-min* $m\ P\ x \Longrightarrow m\ x = m$ (*arg-min* $m\ P$)
⟨*proof*⟩

lemma *is-arg-min-size-subprop*:
  fixes m :: *'a*⇒*'b*::*linorder*
  assumes *is-arg-min* $m\ P\ x$ $Q\ x$ $\bigwedge y.\ Q\ y \Longrightarrow P\ y$
  shows m (*arg-min* $m\ Q$) $= m$ (*arg-min* $m\ P$)
⟨*proof*⟩

1.6.3 Bottom of a set

context *ordering*
begin

definition *has-bottom* :: $'a$ *set* $\Rightarrow$ *bool*
  where *has-bottom* $P \equiv \exists z{\in}P.\ \forall x{\in}P.\ z \leq x$

lemma *has-bottomI*: $z{\in}P \Longrightarrow (\bigwedge x.\ x{\in}P \Longrightarrow z \leq x) \Longrightarrow$ *has-bottom* P
  ⟨*proof*⟩

lemma *has-uniq-bottom*: *has-bottom* $P \Longrightarrow \exists!z{\in}P.\ \forall x{\in}P.\ z{\leq}x$
  ⟨*proof*⟩

definition *bottom* :: $'a$ *set* $\Rightarrow$ $'a$
  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* $P \leq x$
  ⟨*proof*⟩

lemma *bottomI*: $z{\in}P \Longrightarrow (\bigwedge y.\ y{\in}P \Longrightarrow z \leq y) \Longrightarrow z =$ *bottom* P
  ⟨*proof*⟩

end

lemma *has-bottom-pow*: *order.has-bottom* (*Pow* A)
  ⟨*proof*⟩

lemma *bottom-pow*: *order.bottom* (*Pow* A) = {}
⟨*proof*⟩

context *OrderingSetMap*
begin

abbreviation *dombot* $\equiv$ *domain.bottom* P
abbreviation *codbot* $\equiv$ *codomain.bottom* (f‘P)

lemma *im-has-bottom*: *domain.has-bottom* $P \Longrightarrow$ *codomain.has-bottom* (f‘P)
  ⟨*proof*⟩

lemma *im-bottom*: *domain.has-bottom* $P \Longrightarrow f$ *dombot* = *codbot*
  ⟨*proof*⟩

end

lemma (**in** *OrderingSetIso*) *pullback-has-bottom*:
  assumes *codomain.has-bottom* (f‘P)
  shows *domain.has-bottom* P
⟨*proof*⟩

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

⟦ *domain.has-bottom P*; *x*∈*P*; *f x* = *codomain.bottom* (*f*‘*P*) ⟧ ⟹
  x = *domain.bottom P*
⟨*proof*⟩

1.6.4 Minimal and pseudominimal elements in sets

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

context *ordering*
begin

definition *minimal-in* :: ′*a set* ⇒ ′*a* ⇒ *bool*
  where *minimal-in P x* ≡ *x*∈*P* ∧ (∀ *z*∈*P*. ¬ *z* < *x*)

definition *pseudominimal-in* :: ′*a set* ⇒ ′*a* ⇒ *bool*
  where *pseudominimal-in P x* ≡ *minimal-in* (*P* − {*bottom P*}) *x*
— only makes sense for *has-bottom P*

lemma *minimal-inD1*: *minimal-in P x* ⟹ *x*∈*P*
  ⟨*proof*⟩

lemma *minimal-inD2*: *minimal-in P x* ⟹ *z*∈*P* ⟹ ¬ *z* < *x*
  ⟨*proof*⟩

lemma *pseudominimal-inD1*: *pseudominimal-in P x* ⟹ *x*∈*P*
  ⟨*proof*⟩

lemma *pseudominimal-inD2*:
  pseudominimal-in P x ⟹ *z*∈*P* ⟹ *z*<*x* ⟹ *z* = *bottom P*
  ⟨*proof*⟩

lemma *pseudominimal-inI*:
  assumes *x*∈*P x* ≠ *bottom P* ⋀*z*. *z*∈*P* ⟹ *z*<*x* ⟹ *z* = *bottom P*
  shows *pseudominimal-in P x*
  ⟨*proof*⟩

lemma *pseudominimal-ne-bottom*: *pseudominimal-in P x* ⟹ *x* ≠ *bottom P*
  ⟨*proof*⟩

lemma *pseudominimal-comp*:
  ⟦ *pseudominimal-in P x*; *pseudominimal-in P y*; *x*≤*y* ⟧ ⟹ *x* = *y*
  ⟨*proof*⟩

end

lemma *pseudominimal-in-pow*:
  assumes *order.pseudominimal-in* (*Pow A*) *x*
  shows ∃ *a*∈*A*. *x* = {*a*}
⟨*proof*⟩

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

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

lemma (**in** *OrderingSetIso*) *pseudominimal-map*:
domain.has-bottom $P \Longrightarrow$ *domain.pseudominimal-in* $P\ x \Longrightarrow$
codomain.pseudominimal-in (f‘P) ($f\ x$)
⟨*proof*⟩

lemma (**in** *OrderingSetIso*) *pullback-pseudominimal-in*:
⟦ *domain.has-bottom* P; $x \in P$; *codomain.pseudominimal-in* (f‘P) ($f\ x$) ⟧ $\Longrightarrow$
domain.pseudominimal-in $P\ x$
⟨*proof*⟩

1.6.5 Set of elements below another

abbreviation (**in** *ordering*) *below-in* :: *'a set* ⇒ *'a* ⇒ *'a set* (**infix** ‹.≤› *70*)
where $P.{\le}x \equiv \{y \in P.\ y \le x\}$

abbreviation (**in** *ord*) *below-in* :: *'a set* ⇒ *'a* ⇒ *'a set* (**infix** ‹.≤› *70*)
where $P.{\le}x \equiv \{y \in P.\ y \le x\}$

context *ordering*
begin

lemma *below-in-refl*: $x \in P \Longrightarrow x \in P.{\le}x$
⟨*proof*⟩

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

lemma *bottom-in-below-in*: *has-bottom* $P \Longrightarrow x \in P \Longrightarrow$ *bottom* $P \in P.{\le}x$
⟨*proof*⟩

lemma *below-in-singleton-is-bottom*:
⟦ *has-bottom* P; $x \in P$; $P.{\le}x = \{x\}$ ⟧ $\Longrightarrow x =$ *bottom* P
⟨*proof*⟩

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

lemma *bottom-below-in-relative*:
⟦ *has-bottom* $(P.{\le}y)$; $x \in P$; $x \le y$ ⟧ $\Longrightarrow$ *bottom* $(P.{\le}x) =$ *bottom* $(P.{\le}y)$

⟨*proof*⟩

lemma *has-bottom-pseudominimal-in-below-inI*:
assumes *has-bottom P x*∈*P pseudominimal-in P y y*≤*x*
shows *pseudominimal-in* (*P*.≤*x*) *y*
⟨*proof*⟩

lemma *has-bottom-pseudominimal-in-below-in*:
assumes *has-bottom P x*∈*P pseudominimal-in* (*P*.≤*x*) *y*
shows *pseudominimal-in P y*
⟨*proof*⟩

lemma *pseudominimal-in-below-in*:
assumes *has-bottom* (*P*.≤*y*) *x*∈*P x*≤*y pseudominimal-in* (*P*.≤*x*) *w*
shows *pseudominimal-in* (*P*.≤*y*) *w*
⟨*proof*⟩

lemma *collect-pseudominimals-below-in-less-eq-top*:
assumes *OrderingSetIso less-eq less* (⊆) (⊂) (*P*.≤*x*) *f*
f‘(*P*.≤*x*) = *Pow A a* ⊆ {*y*. *pseudominimal-in* (*P*.≤*x*) *y*}
defines *w* ≡ *the-inv-into* (*P*.≤*x*) *f* (⋃(*f*‘*a*))
shows *w* ≤ *x*
⟨*proof*⟩

lemma *collect-pseudominimals-below-in-poset*:
assumes *OrderingSetIso less-eq less* (⊆) (⊂) (*P*.≤*x*) *f*
f‘(*P*.≤*x*) = *Pow A*
a ⊆ {*y*. *pseudominimal-in* (*P*.≤*x*) *y*}
defines *w* ≡ *the-inv-into* (*P*.≤*x*) *f* (⋃(*f*‘*a*))
shows *w* ∈ *P*
⟨*proof*⟩

lemma *collect-pseudominimals-below-in-eq*:
assumes *x*∈*P OrderingSetIso less-eq less* (⊆) (⊂) (*P*.≤*x*) *f*
f‘(*P*.≤*x*) = *Pow A a* ⊆ {*y*. *pseudominimal-in* (*P*.≤*x*) *y*}
defines *w*: *w* ≡ *the-inv-into* (*P*.≤*x*) *f* (⋃(*f*‘*a*))
shows *a* = {*y*. *pseudominimal-in* (*P*.≤*w*) *y*}
⟨*proof*⟩

end

1.6.6 Lower bounds

context *ordering*
begin

definition *lbound-of* :: ′*a* ⇒ ′*a* ⇒ ′*a* ⇒ *bool*
where *lbound-of x y b* ≡ *b*≤*x* ∧ *b*≤*y*

lemma *lbound-ofI*: $b \leq x \Longrightarrow b \leq y \Longrightarrow$ *lbound-of* $x\ y\ b$
⟨*proof*⟩

lemma *lbound-ofD1*: *lbound-of* $x\ y\ b \Longrightarrow b \leq x$
⟨*proof*⟩

lemma *lbound-ofD2*: *lbound-of* $x\ y\ b \Longrightarrow b \leq y$
⟨*proof*⟩

definition *glbound-in-of* :: $'a$ *set* $\Rightarrow\ 'a \Rightarrow\ 'a \Rightarrow\ 'a \Rightarrow$ *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*:
$[\![$ $b \in P$; *lbound-of* $x\ y\ b$; $\bigwedge a.\ a \in P \Longrightarrow$ *lbound-of* $x\ y\ a \Longrightarrow a \leq b$ $]\!] \Longrightarrow$
glbound-in-of $P\ x\ y\ b$
⟨*proof*⟩

lemma *glbound-in-ofD-in*: *glbound-in-of* $P\ x\ y\ b \Longrightarrow b \in P$
⟨*proof*⟩

lemma *glbound-in-ofD-lbound*: *glbound-in-of* $P\ x\ y\ b \Longrightarrow$ *lbound-of* $x\ y\ b$
⟨*proof*⟩

lemma *glbound-in-ofD-glbound*:
glbound-in-of $P\ x\ y\ b \Longrightarrow a \in P \Longrightarrow$ *lbound-of* $x\ y\ a \Longrightarrow a \leq b$
⟨*proof*⟩

lemma *glbound-in-of-less-eq1*: *glbound-in-of* $P\ x\ y\ b \Longrightarrow b \leq x$
⟨*proof*⟩

lemma *glbound-in-of-less-eq2*: *glbound-in-of* $P\ x\ y\ b \Longrightarrow b \leq y$
⟨*proof*⟩

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

end

1.6.7 Simplex-like posets

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

context *ordering*
begin

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

lemma *simplex-likeI*:
assumes *finite* P *OrderingSetIso less-eq less* $(\subseteq)$ $(\subset)$ $P\ f$
$f`P = Pow\ (A$::*nat set*)
shows *simplex-like* P
⟨*proof*⟩

lemma *simplex-likeD-finite*: *simplex-like* $P \Longrightarrow$ *finite* P
⟨*proof*⟩

lemma *simplex-likeD-iso*:
simplex-like $P \Longrightarrow$
$\exists f\ A$::*nat set*. *OrderingSetIso less-eq less* $(\subseteq)$ $(\subset)$ $P\ f \wedge f`P = Pow\ A$
⟨*proof*⟩

lemma *simplex-like-has-bottom*: *simplex-like* $P \Longrightarrow$ *has-bottom* P
⟨*proof*⟩

lemma *simplex-like-no-pseudominimal-imp-singleton*:
assumes *simplex-like* P $\bigwedge x.\ \neg$ *pseudominimal-in* $P\ x$
shows $\exists p.\ P = \{p\}$
⟨*proof*⟩

lemma *simplex-like-no-pseudominimal-in-below-in-imp-singleton*:
$[\![\ x{\in}P;$ *simplex-like* $(P.{\le}x)$; $\bigwedge z.\ \neg$ *pseudominimal-in* $(P.{\le}x)\ z\ ]\!] \Longrightarrow$
$P.{\le}x = \{x\}$
⟨*proof*⟩

lemma *pseudo-simplex-like-has-bottom*:
OrderingSetIso less-eq less $(\subseteq)$ $(\subset)$ $P\ f \Longrightarrow f`P = Pow\ A \Longrightarrow$
has-bottom P
⟨*proof*⟩

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* $P\ x \Longrightarrow x \le t$
shows $f\ t = A$
⟨*proof*⟩

lemma *pseudo-simplex-like-below-in-above-pseudominimal-is-top*:
assumes $x{\in}P$ *OrderingSetIso less-eq less* $(\subseteq)$ $(\subset)$ $(P.{\le}x)\ f$
$f`(P.{\le}x) = Pow\ A\ \ t \in P.{\le}x$
$\bigwedge y.$ *pseudominimal-in* $(P.{\le}x)\ y \Longrightarrow y \le t$
shows $t = x$

⟨*proof*⟩

lemma *simplex-like-below-in-above-pseudominimal-is-top*:
assumes $x \in P$ *simplex-like* $(P.\leq x)$ $t \in P.\leq x$
$\bigwedge y$. *pseudominimal-in* $(P.\leq x)$ $y \Longrightarrow y \leq t$
shows $t = x$
⟨*proof*⟩

end

lemma (**in** *OrderingSetIso*) *simplex-like-map*:
assumes *domain.simplex-like* P
shows *codomain.simplex-like* $(f`P)$
⟨*proof*⟩

lemma (**in** *OrderingSetIso*) *pullback-simplex-like*:
assumes *finite* P *codomain.simplex-like* $(f`P)$
shows *domain.simplex-like* P
⟨*proof*⟩

lemma *simplex-like-pow*:
assumes *finite* A
shows *order.simplex-like* (*Pow* A)
⟨*proof*⟩

1.6.8 The superset ordering

abbreviation *supset-has-bottom* ≡ *ordering.has-bottom* $(\supseteq)$
abbreviation *supset-bottom* ≡ *ordering.bottom* $(\supseteq)$
abbreviation *supset-lbound-of* ≡ *ordering.lbound-of* $(\supseteq)$
abbreviation *supset-glbound-in-of* ≡ *ordering.glbound-in-of* $(\supseteq)$
abbreviation *supset-simplex-like* ≡ *ordering.simplex-like* $(\supseteq)$ $(\supset)$
abbreviation *supset-pseudominimal-in* ≡
ordering.pseudominimal-in $(\supseteq)$ $(\supset)$

abbreviation *supset-below-in* :: *'a set set* ⇒ *'a set* ⇒ *'a set set* (**infix** ‹.⊇› *70*)
where $P.\supseteq A$ ≡ *ordering.below-in* $(\supseteq)$ P A

lemma *supset-poset*: *ordering* $(\supseteq)$ $(\supset)$ ⟨*proof*⟩

lemmas *supset-bottomI* = *ordering.bottomI* [*OF supset-poset*]
lemmas *supset-pseudominimal-inI* = *ordering.pseudominimal-inI* [*OF supset-poset*]
lemmas *supset-pseudominimal-inD1* = *ordering.pseudominimal-inD1* [*OF supset-poset*]
lemmas *supset-pseudominimal-inD2* = *ordering.pseudominimal-inD2* [*OF supset-poset*]
lemmas *supset-lbound-ofI* = *ordering.lbound-ofI* [*OF supset-poset*]
lemmas *supset-lbound-of-def* = *ordering.lbound-of-def* [*OF supset-poset*]
lemmas *supset-glbound-in-ofI* = *ordering.glbound-in-ofI* [*OF supset-poset*]
lemmas *supset-pseudominimal-ne-bottom* =
ordering.pseudominimal-ne-bottom[*OF supset-poset*]

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

lemma *OrderingSetIso-pow-complement*:
OrderingSetIso (⊇) (⊃) (⊆) (⊂) (*Pow A*) ((−) *A*)
⟨*proof*⟩

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

end

2 Algebra

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

theory *Algebra*
imports *Prelim*

begin

2.1 Miscellaneous algebra facts

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

lemma (**in** *comm-semiring-1*) *odd-n0*: *odd m* ⟹ *m*≠*0*
⟨*proof*⟩

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

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

context *group-add*
begin

lemma *map-uminus-order2*:
∀ *s*∈*set ss*. *s*+*s*=*0* ⟹ *map* (*uminus*) *ss* = *ss*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *uminus-sum-list-order2*:
∀ *s*∈*set ss*. *s*+*s*=*0* ⟹ − *sum-list ss* = *sum-list* (*rev ss*)
⟨*proof*⟩

end

2.2 The type of permutations of a type

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

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

setup-lifting *type-definition-permutation*

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

lemma *permutation-eq-image*: *a* ‘→ *A* = *a* ‘→ *B* ⟹ *A*=*B*
⟨*proof*⟩

instantiation *permutation* :: (*type*) *zero*
begin
lift-definition *zero-permutation* :: *'a permutation* **is** *id*::*'a*⇒*'a* ⟨*proof*⟩
instance ⟨*proof*⟩
end

instantiation *permutation* :: (*type*) *plus*
begin
lift-definition *plus-permutation* :: *'a permutation* ⇒ *'a permutation* ⇒ *'a permutation*
is *comp*
⟨*proof*⟩
instance ⟨*proof*⟩
end

lemma *plus-permutation-abs-eq*:
bij f ⟹ *bij g* ⟹

Abs-permutation f + *Abs-permutation g* = *Abs-permutation* (*f*∘*g*)
⟨*proof*⟩

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

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

instantiation *permutation* :: (*type*) *uminus*
begin
lift-definition *uminus-permutation* :: *'a permutation* ⇒ *'a permutation*
is λ*f*. *the-inv f*
⟨*proof*⟩
instance ⟨*proof*⟩
end

instantiation *permutation* :: (*type*) *minus*
begin
lift-definition *minus-permutation* :: *'a permutation* ⇒ *'a permutation* ⇒ *'a permutation*
is λ*f g*. *f* ∘ (*the-inv g*)
⟨*proof*⟩
instance ⟨*proof*⟩
end

lemma *minus-permutation-abs-eq*:
bij f ⟹ *bij g* ⟹
Abs-permutation f − *Abs-permutation g* = *Abs-permutation* (*f* ∘ *the-inv g*)
⟨*proof*⟩

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

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

2.3.1 Translation from class *power*.

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

context *monoid-add*
begin

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

abbreviation *nataction* :: *'a* ⇒ *nat* ⇒ *'a* (**infix** ‹+^› *80*)
where *a*+^*n* ≡ *nataction.power a n*

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

lemma *alternating-sum-list-conv-natactation*:
sum-list (*alternating-list* (*2*∗*n*) *s t*) = (*s*+*t*)+^*n*
⟨*proof*⟩

lemma *natactation-add-flip*: (*a*+*b*)+^(*Suc n*) = *a* + (*b*+*a*)+^*n* + *b*
⟨*proof*⟩

end

lemma (**in** *group-add*) *natactation-add-eq0-flip*:
assumes (*a*+*b*)+^*n* = *0*
shows (*b*+*a*)+^*n* = *0*
⟨*proof*⟩

2.3.2 Additive order of an element

context *monoid-add*
begin

definition *add-order* :: ′*a* ⇒ *nat*
where *add-order a* ≡ *if* (∃ *n*>*0*. *a*+^*n* = *0*) *then*
(*LEAST n*. *n*>*0* ∧ *a*+^*n* = *0*) *else 0*

lemma *add-order*: *a*+^(*add-order a*) = *0*
⟨*proof*⟩

lemma *add-order-least*: *n*>*0* ⟹ *a*+^*n* = *0* ⟹ *add-order a* ≤ *n*
⟨*proof*⟩

lemma *add-order-equality*:
⟦ *n*>*0*; *a*+^*n* = *0*; (⋀*m*. *m*>*0* ⟹ *a*+^*m* = *0* ⟹ *n*≤*m*) ⟧ ⟹
add-order a = *n*
⟨*proof*⟩

lemma *add-order0*: *add-order 0* = *1*
⟨*proof*⟩

lemma *add-order-gt0*: (*add-order a* > *0*) = (∃ *n*>*0*. *a*+^*n* = *0*)
⟨*proof*⟩

lemma *add-order-eq0*: *add-order a* = *0* ⟹ *n*>*0* ⟹ *a*+^*n* ≠ *0*
⟨*proof*⟩

lemma *less-add-order-eq-0*:
assumes *a*+^*k* = *0 k* < *add-order a*
shows *k* = *0*

⟨*proof*⟩

lemma *less-add-order-eq-0-contra*: $k>0 \Longrightarrow k < add\text{-}order\ a \Longrightarrow a+\hat{}k \neq 0$
  ⟨*proof*⟩

lemma *add-order-relator*: *add-order* $(a+\hat{}(add\text{-}order\ a)) = 1$
  ⟨*proof*⟩

abbreviation *pair-relator-list* :: *'a ⇒ 'a ⇒ 'a list*
  where *pair-relator-list s t ≡ alternating-list* (*2∗add-order* (*s+t*)) *s t*
abbreviation *pair-relator-halflist* :: *'a ⇒ 'a ⇒ 'a list*
  where *pair-relator-halflist s t ≡ alternating-list* (*add-order* (*s+t*)) *s t*
abbreviation *pair-relator-halflist2* :: *'a ⇒ 'a ⇒ 'a list*
  where *pair-relator-halflist2 s t ≡*
    (*if even* (*add-order* (*s+t*)) *then pair-relator-halflist s t else*
      pair-relator-halflist t s)

lemma *sum-list-pair-relator-list*: *sum-list* (*pair-relator-list s t*) = *0*
  ⟨*proof*⟩

end

context *group-add*
begin

lemma *add-order-add-eq1*: *add-order* (*s+t*) = *1* ⟹ *t* = −*s*
  ⟨*proof*⟩

lemma *add-order-add-sym*: *add-order* (*t+s*) = *add-order* (*s+t*)
⟨*proof*⟩

lemma *pair-relator-halflist-append*:
  pair-relator-halflist s t @ *pair-relator-halflist2 s t* = *pair-relator-list s t*
  ⟨*proof*⟩

lemma *rev-pair-relator-list*: *rev* (*pair-relator-list s t*) = *pair-relator-list t s*
  ⟨*proof*⟩

lemma *pair-relator-halflist2-conv-rev-pair-relator-halflist*:
  pair-relator-halflist2 s t = *rev* (*pair-relator-halflist t s*)
  ⟨*proof*⟩

end

2.4 Partial sums of a list

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

context *monoid-add*

begin

primrec *sums* :: *'a list* ⇒ *'a list*
  where
    sums [] = [*0*]
  | *sums* (*x*#*xs*) = *0* # *map* ((+) *x*) (*sums xs*)

lemma *length-sums*: *length* (*sums xs*) = *Suc* (*length xs*)
  ⟨*proof*⟩

lemma *sums-snoc*: *sums* (*xs*@[*x*]) = *sums xs* @ [*sum-list* (*xs*@[*x*])]
  ⟨*proof*⟩

lemma *sums-append2*:
  sums (*xs*@*ys*) = *butlast* (*sums xs*) @ *map* ((+) (*sum-list xs*)) (*sums ys*)
⟨*proof*⟩

lemma *sums-Cons-conv-append-tl*:
  sums (*x*#*xs*) = *0* # *x* # *map* ((+) *x*) (*tl* (*sums xs*))
  ⟨*proof*⟩

lemma *pullback-sums-map-middle2*:
  map F (*sums xs*) = *ds*@[*d*,*e*]@*es* ⟹
    ∃ *as a bs*. *xs* = *as*@[*a*]@*bs* ∧ *map F* (*sums as*) = *ds*@[*d*] ∧
      d = *F* (*sum-list as*) ∧ *e* = *F* (*sum-list* (*as*@[*a*]))
⟨*proof*⟩

lemma *pullback-sums-map-middle3*:
  map F (*sums xs*) = *ds*@[*d*,*e*,*f*]@*fs* ⟹
    ∃ *as 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*⟩

lemma *pullback-sums-map-double-middle2*:
  assumes *map F* (*sums xs*) = *ds*@[*d*,*e*]@*es*@[*f*,*g*]@*gs*
  shows ∃ *as a bs b cs*. *xs* = *as*@[*a*]@*bs*@[*b*]@*cs* ∧ *d* = *F* (*sum-list as*) ∧
          e = *F* (*sum-list* (*as*@[*a*])) ∧ *f* = *F* (*sum-list* (*as*@[*a*]@*bs*)) ∧
          g = *F* (*sum-list* (*as*@[*a*]@*bs*@[*b*]))
⟨*proof*⟩

end

2.5 Sums of alternating lists

lemma (**in** *group-add*) *uminus-sum-list-alternating-order2*:
  s+*s*=*0* ⟹ *t*+*t*=*0* ⟹ − *sum-list* (*alternating-list n s t*) =
    sum-list (*if even n then alternating-list n t s else alternating-list n s t*)
  ⟨*proof*⟩

context *monoid-add*
begin

lemma *alternating-order2-cancel-1left*:
  s+s=0 $\Longrightarrow$
    sum-list (*s* # (*alternating-list* (*Suc n*) *s t*)) = *sum-list* (*alternating-list n t s*)
  ⟨*proof*⟩

lemma *alternating-order2-cancel-2left*:
  s+s=0 $\Longrightarrow$ *t+t=0* $\Longrightarrow$
    sum-list (*t* # *s* # (*alternating-list* (*Suc* (*Suc n*)) *s t*)) =
      sum-list (*alternating-list n s t*)
    ⟨*proof*⟩

lemma *alternating-order2-even-cancel-right*:
  assumes *st* : *s+s=0 t+t=0*
  and *even-n*: *even n*
  shows *m* ≤ *n* $\Longrightarrow$ *sum-list* (*alternating-list n s t* @ *alternating-list m t s*) =
    sum-list (*alternating-list* (*n−m*) *s t*)
⟨*proof*⟩

end

2.6 Conjugation in *group-add*

2.6.1 Abbreviations and basic facts

context *group-add*
begin

abbreviation *lconjby* :: *'a⇒'a⇒'a*
  where *lconjby x y* ≡ *x+y−x*

abbreviation *rconjby* :: *'a⇒'a⇒'a*
  where *rconjby x y* ≡ *−x+y+x*

lemma *lconjby-add*: *lconjby* (*x+y*) *z* = *lconjby x* (*lconjby y z*)
  ⟨*proof*⟩

lemma *rconjby-add*: *rconjby* (*x+y*) *z* = *rconjby y* (*rconjby x z*)
  ⟨*proof*⟩

lemma *add-rconjby*: *rconjby x y* + *rconjby x z* = *rconjby x* (*y+z*)
  ⟨*proof*⟩

lemma *lconjby-uminus*: *lconjby x* (*−y*) = − *lconjby x y*
  ⟨*proof*⟩

lemma *rconjby-uminus*: *rconjby x* (*−y*) = − *rconjby x y*
  ⟨*proof*⟩

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

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

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

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

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

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

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

lemma *lconjby-eq-conv-rconjby-eq*: *w* = *lconjby x y* ⟹ *y* = *rconjby x w*
⟨*proof*⟩

lemma *rconjby-order2*: *s*+*s* = *0* ⟹ *rconjby x s* + *rconjby x s* = *0*
⟨*proof*⟩

lemma *rconjby-order2-eq-lconjby*:
assumes *s*+*s*=*0*
shows *rconjby s* = *lconjby s*
⟨*proof*⟩

lemma *lconjby-alternating-list-order2*:
assumes *s*+*s*=*0* *t*+*t*=*0*
shows *lconjby* (*sum-list* (*alternating-list k s t*)) (*if even k then s else t*) =
sum-list (*alternating-list* (*Suc* (*2*∗*k*)) *s t*)
⟨*proof*⟩

end

2.6.2 The conjugation sequence

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

context *group-add*
begin

primrec *lconjseq* :: ′*a list* ⇒ ′*a list*

where
  lconjseq [] = []
| *lconjseq* (*x*#*xs*) = *x* # (*map* (*lconjby x*) (*lconjseq xs*))

lemma *length-lconjseq*: *length* (*lconjseq xs*) = *length xs*
⟨*proof*⟩

lemma *lconjseq-snoc*: *lconjseq* (*xs*@[*x*]) = *lconjseq xs* @ [*lconjby* (*sum-list xs*) *x*]
⟨*proof*⟩

lemma *lconjseq-append*:
lconjseq (*xs*@*ys*) = *lconjseq xs* @ (*map* (*lconjby* (*sum-list xs*)) (*lconjseq ys*))
⟨*proof*⟩

lemma *lconjseq-alternating-order2-repeats'*:
fixes *s t* :: *'a*
defines *altst*: *altst* ≡ λ*n*. *alternating-list n s t*
and *altts*: *altts* ≡ λ*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*⟩

lemma *lconjseq-alternating-order2-repeats*:
fixes *s t* :: *'a* **and** *k* :: *nat*
defines *altst*: *altst* ≡ λ*n*. *alternating-list n s t*
and *altts*: *altts* ≡ λ*n*. *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*⟩

lemma *even-count-lconjseq-alternating-order2*:
fixes *s t* :: *'a*
assumes *s*+*s*=*0* *t*+*t*=*0* (*s*+*t*)+^*k* = *0*
shows *even* (*count-list* (*lconjseq* (*alternating-list* (*2***k*) *s t*)) *x*)
⟨*proof*⟩

lemma *order2-hd-in-lconjseq-deletion*:
shows *s*+*s*=*0* ⟹ *s* ∈ *set* (*lconjseq ss*)
⟹ ∃ *as b bs*. *ss* = *as*@[*b*]@*bs* ∧ *sum-list* (*s*#*ss*) = *sum-list* (*as*@*bs*)
⟨*proof*⟩

end

2.6.3 The action on signed *group-add* elements

Here we construct an action of a group on itself by conjugation, where group elements are endowed with an auxiliary sign by pairing with a boolean element. In multiple applications of this action, the auxiliary sign helps keep track of how many times the elements conjugating and being conjugated are the same. This action arises in exploring reduced expressions of group elements as words in a set of generators of order two (in particular, in a Coxeter group).

type-synonym *'a signed* = *'a*×*bool*

definition *signed-funaction* :: (*'a*⇒*'a*⇒*'a*) ⇒ *'a* ⇒ *'a signed* ⇒ *'a signed*
  where *signed-funaction f s x* ≡ *map-prod* (*f s*) (λ*b*. *b* ≠ (*fst x* = *s*)) *x*
  — so the sign of x is flipped precisely when its first component is equal to s

context *group-add*
begin

abbreviation *signed-lconjaction* ≡ *signed-funaction lconjby*
abbreviation *signed-rconjaction* ≡ *signed-funaction rconjby*

lemmas *signed-lconjactionD* = *signed-funaction-def*[*of lconjby*]
lemmas *signed-rconjactionD* = *signed-funaction-def*[*of rconjby*]

abbreviation *signed-lconjpermutation* :: *'a* ⇒ *'a signed permutation*
  where *signed-lconjpermutation s* ≡ *Abs-permutation* (*signed-lconjaction s*)

abbreviation *signed-list-lconjaction* :: *'a list* ⇒ *'a signed* ⇒ *'a signed*
  where *signed-list-lconjaction ss* ≡ *foldr signed-lconjaction ss*

lemma *signed-lconjaction-fst*: *fst* (*signed-lconjaction s x*) = *lconjby s* (*fst x*)
  ⟨*proof*⟩

lemma *signed-lconjaction-rconjaction*:
  signed-lconjaction s (*signed-rconjaction s x*) = *x*
⟨*proof*⟩

lemma *signed-rconjaction-by-order2-eq-lconjaction*:
  s+*s*=*0* ⟹ *signed-rconjaction s* = *signed-lconjaction s*
  ⟨*proof*⟩

lemma *inj-signed-lconjaction*: *inj* (*signed-lconjaction s*)
⟨*proof*⟩

lemma *surj-signed-lconjaction*: *surj* (*signed-lconjaction s*)
  ⟨*proof*⟩

lemma *bij-signed-lconjaction*: *bij* (*signed-lconjaction s*)
  ⟨*proof*⟩

lemma *the-inv-signed-lconjaction*:
  the-inv (*signed-lconjaction s*) = *signed-rconjaction s*
⟨*proof*⟩

lemma *the-inv-signed-lconjaction-by-order2*:
$s+s=0 \Longrightarrow$ *the-inv* (*signed-lconjaction s*) = *signed-lconjaction s*
⟨*proof*⟩

lemma *signed-list-lconjaction-fst*:
fst (*signed-list-lconjaction ss x*) = *lconjby* (*sum-list ss*) (*fst x*)
⟨*proof*⟩

lemma *signed-list-lconjaction-snd*:
shows $\forall$ *s*∈*set ss*. $s+s=0 \Longrightarrow$ *snd* (*signed-list-lconjaction ss x*)
= (*if even* (*count-list* (*lconjseq* (*rev ss*)) (*fst x*)) *then snd x else* ¬*snd x*)
⟨*proof*⟩

end

2.7 Cosets

2.7.1 Basic facts

lemma *set-zero-plus′* [*simp*]: (*0*::′*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*.
⟨*proof*⟩

lemma *lcoset-0*: (*w*::′*a*::*monoid-add*) +*o 0* = {*w*}
⟨*proof*⟩

lemma *lcoset-refl*: (*0*::′*a*::*monoid-add*) ∈ *A* ⟹ *a* ∈ *a* +*o A*
⟨*proof*⟩

lemma *lcoset-eq-reps-subset*:
(*a*::′*a*::*group-add*) +*o A* ⊆ *a* +*o B* ⟹ *A* ⊆ *B*
⟨*proof*⟩

lemma *lcoset-eq-reps*: (*a*::′*a*::*group-add*) +*o A* = *a* +*o B* ⟹ *A* = *B*
⟨*proof*⟩

lemma *lcoset-inj-on*: *inj* ((+*o*) (*a*::′*a*::*group-add*))
⟨*proof*⟩

lemma *lcoset-conv-set*: (*a*::′*g*::*group-add*) ∈ *b* +*o A* ⟹ −*b* + *a* ∈ *A*
⟨*proof*⟩

2.7.2 The supset order on cosets

lemma *supset-lbound-lcoset-shift*:
supset-lbound-of X Y B ⟹
ordering.lbound-of (⊇) (*a* +*o X*) (*a* +*o Y*) (*a* +*o B*)
⟨*proof*⟩

lemma *supset-glbound-in-of-lcoset-shift*:
  fixes P :: *'a::group-add set set*
  assumes *supset-glbound-in-of* $P\ X\ Y\ B$
  shows *supset-glbound-in-of* $((+o)\ a$ ` $P)\ (a\ +o\ X)\ (a\ +o\ Y)\ (a\ +o\ B)$
  ⟨*proof*⟩

2.7.3 The afforded partition

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

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

2.8 Groups

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

2.8.1 Locale definition and basic facts

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

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

lemma *SubgroupD1*: *Subgroup* $H \Longrightarrow$ *Group* H ⟨*proof*⟩

lemma *zero-closed* : $0 \in G$
⟨*proof*⟩

lemma *uminus-closed*: $g{\in}G \Longrightarrow -g{\in}G$
  ⟨*proof*⟩

lemma *add-closed*: $g{\in}G \Longrightarrow h{\in}G \Longrightarrow g+h \in G$
  ⟨*proof*⟩

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

lemma *lconjby-closed*: $g{\in}G \Longrightarrow x{\in}G \Longrightarrow$ *lconjby* $g\ x \in G$
  ⟨*proof*⟩

lemma *lconjby-set-closed*: $g{\in}G \Longrightarrow A{\subseteq}G \Longrightarrow$ *lconjby* g ` $A \subseteq G$
  ⟨*proof*⟩

lemma *set-lconjby-subset-closed*:
$H \subseteq G \Longrightarrow A \subseteq G \Longrightarrow (\bigcup h \in H.\ lconjby\ h\ `\ A) \subseteq G$
⟨*proof*⟩

lemma *sum-list-map-closed*: $set\ (map\ f\ as) \subseteq G \Longrightarrow (\sum a \leftarrow as.\ f\ a) \in G$
⟨*proof*⟩

lemma *sum-list-closed*: $set\ as \subseteq G \Longrightarrow sum\text{-}list\ as \in G$
⟨*proof*⟩

end

2.8.2 Sets with a suitable binary operation

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

locale *BinOpSetGroup* =
fixes G :: $'a\ set$
and $binop$ :: $'a \Rightarrow 'a \Rightarrow 'a$
and e :: $'a$
assumes *closed* : $g \in G \Longrightarrow h \in G \Longrightarrow binop\ g\ h \in G$
and *assoc* :
$[\![g \in G;\ h \in G;\ k \in G]\!] \Longrightarrow binop\ (binop\ g\ h)\ k = binop\ g\ (binop\ h\ k)$
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*: $g \in G \Longrightarrow \forall x \in G.\ binop\ g\ x = x \Longrightarrow g = e$
⟨*proof*⟩

lemma *unique-inverse*:
assumes $g \in G$
shows $\exists !h.\ h \in G \wedge binop\ g\ h = e \wedge binop\ h\ g = e$
⟨*proof*⟩

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

definition *Abs-G-perm* :: $'a \Rightarrow 'a\ permutation$
where $Abs\text{-}G\text{-}perm\ g \equiv Abs\text{-}permutation\ (G\text{-}perm\ g)$

abbreviation $\mathfrak{p} \equiv Abs\text{-}G\text{-}perm$ — the injection into type *permutation*
abbreviation $\mathfrak{ip} \equiv the\text{-}inv\text{-}into\ G\ \mathfrak{p}$ — the reverse correspondence

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

lemma *G-perm-comp*:
$g{\in}G \Longrightarrow h{\in}G \Longrightarrow G\text{-}perm\ g \circ G\text{-}perm\ h = G\text{-}perm\ (binop\ g\ h)$
⟨*proof*⟩

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

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

lemma *the-inverseD*:
assumes $g{\in}G$
shows $\mathfrak{i}\ g \in G$ $binop\ g\ (\mathfrak{i}\ g) = e$ $binop\ (\mathfrak{i}\ g)\ g = e$
⟨*proof*⟩

lemma *binop-G-comp-binop-*$\mathfrak{i}$*G*: $g{\in}G \Longrightarrow x{\in}G \Longrightarrow binop\ g\ (binop\ (\mathfrak{i}\ g)\ x) = x$
⟨*proof*⟩

lemma *bij-betw-binop-G*:
assumes $g{\in}G$
shows $bij\text{-}betw\ (binop\ g)\ G\ G$
⟨*proof*⟩

lemma *the-inv-into-G-binop-G*:
assumes $g{\in}G$ $x{\in}G$
shows $the\text{-}inv\text{-}into\ G\ (binop\ g)\ x = binop\ (\mathfrak{i}\ g)\ x$
⟨*proof*⟩

lemma *restrict1-the-inv-into-G-binop-G*:
$g{\in}G \Longrightarrow restrict1\ (the\text{-}inv\text{-}into\ G\ (binop\ g))\ G = G\text{-}perm\ (\mathfrak{i}\ g)$
⟨*proof*⟩

lemma *bij-G-perm*: $g{\in}G \Longrightarrow bij\ (G\text{-}perm\ g)$
⟨*proof*⟩

lemma *G-perm-apply*: $g{\in}G \Longrightarrow x{\in}G \Longrightarrow \mathfrak{p}\ g \rightarrow x = binop\ g\ x$
⟨*proof*⟩

lemma *G-perm-apply-identity*: $g{\in}G \Longrightarrow \mathfrak{p}\ g \rightarrow e = g$
⟨*proof*⟩

lemma *the-inv-G-perm*:
$g{\in}G \Longrightarrow the\text{-}inv\ (G\text{-}perm\ g) = G\text{-}perm\ (\mathfrak{i}\ g)$
⟨*proof*⟩

lemma *Abs-G-perm-diff*:
$g{\in}G \Longrightarrow h{\in}G \Longrightarrow \mathfrak{p}\ g - \mathfrak{p}\ h = \mathfrak{p}\ (binop\ g\ (\mathfrak{i}\ h))$
⟨*proof*⟩

lemma *Group*: *Group pG*
⟨*proof*⟩

lemma *inj-on-𝔭-G*: *inj-on* 𝔭 *G*
⟨*proof*⟩

lemma *homs*:
$\bigwedge g\ h.\ g{\in}G \Longrightarrow h{\in}G \Longrightarrow \mathfrak{p}\ (binop\ g\ h) = \mathfrak{p}\ g + \mathfrak{p}\ h$
$\bigwedge x\ y.\ x{\in}pG \Longrightarrow y{\in}pG \Longrightarrow binop\ (\mathfrak{ip}\ x)\ (\mathfrak{ip}\ y) = \mathfrak{ip}\ (x{+}y)$
⟨*proof*⟩

lemmas *inv-correspondence-into* =
the-inv-into-into[*OF inj-on-𝔭-G*, *of - G*, *simplified*]

lemma *inv-correspondence-conv-apply*: $x \in pG \Longrightarrow \mathfrak{ip}\ x = x{\rightarrow}e$
⟨*proof*⟩

end

2.8.3 Cosets of a *Group*

context *Group*
begin

lemma *lcoset-refl*: $a \in a +o\ G$
⟨*proof*⟩

lemma *lcoset-el-reduce*:
assumes $a \in G$
shows $a +o\ G = G$
⟨*proof*⟩

lemma *lcoset-el-reduce0*: $0 \in a +o\ G \Longrightarrow a +o\ G = G$
⟨*proof*⟩

lemma *lcoset-subgroup-imp-eq-reps*:
$Group\ H \Longrightarrow w +o\ H \subseteq w' +o\ G \Longrightarrow w' +o\ G = w +o\ G$
⟨*proof*⟩

lemma *lcoset-closed*: $a{\in}G \Longrightarrow A{\subseteq}G \Longrightarrow a +o\ A \subseteq G$
⟨*proof*⟩

lemma *lcoset-rel-sym*: *sym* (*lcoset-rel G*)
⟨*proof*⟩

lemma *lcoset-rel-trans*: *trans* (*lcoset-rel G*)
⟨*proof*⟩

abbreviation *LCoset-rel* :: *'g set* ⇒ (*'g*×*'g*) *set*
where *LCoset-rel H* ≡ *lcoset-rel H* ∩ (*G*×*G*)

lemma *refl-on-LCoset-rel*: *0*∈*H* ⟹ *refl-on G* (*LCoset-rel H*)
⟨*proof*⟩

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

lemma *lcoset-subgroup-rel-equiv*:
Subgroup H ⟹ *equiv G* (*LCoset-rel H*)
⟨*proof*⟩

lemma *trivial-LCoset*: *H*⊆*G* ⟹ *H* = *LCoset-rel H* `` {*0*}
⟨*proof*⟩

end

2.8.4 The *Group* generated by a set

inductive-set *genby* :: *'a::group-add set* ⇒ *'a set* (‹⟨-⟩›)
for *S* :: *'a set*
where
genby-0-closed : *0*∈⟨*S*⟩ — just in case *S* is empty
| *genby-genset-closed*: *s*∈*S* ⟹ *s*∈⟨*S*⟩
| *genby-diff-closed* : *w*∈⟨*S*⟩ ⟹ *w'*∈⟨*S*⟩ ⟹ *w* − *w'* ∈ ⟨*S*⟩

lemma *genby-Group*: *Group* ⟨*S*⟩
⟨*proof*⟩

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

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

lemma *genby-genset-subset*: *S* ⊆ ⟨*S*⟩
⟨*proof*⟩

lemma *genby-uminus-genset-subset*: *uminus* ` *S* ⊆ ⟨*S*⟩
⟨*proof*⟩

lemma *genby-in-sum-list-lists*:
  fixes S
  defines *S-sum-lists*: *S-sum-lists* ≡ ($\bigcup$*ss*∈*lists* (S ∪ *uminus* ‘ S). {*sum-list ss*})
  shows $w \in \langle S \rangle \Longrightarrow w \in$ *S-sum-lists*
⟨*proof*⟩

lemma *sum-list-lists-in-genby*: *ss* ∈ *lists* (S ∪ *uminus* ‘ S) ⟹ *sum-list ss* ∈ ⟨S⟩
⟨*proof*⟩

lemma *sum-list-lists-in-genby-sym*:
  uminus ‘ $S \subseteq S$ ⟹ *ss* ∈ *lists* S ⟹ *sum-list ss* ∈ ⟨S⟩
  ⟨*proof*⟩

lemma *genby-eq-sum-lists*: ⟨S⟩ = ($\bigcup$*ss*∈*lists* (S ∪ *uminus* ‘ S). {*sum-list ss*})
  ⟨*proof*⟩

lemma *genby-mono*: $T \subseteq S \Longrightarrow \langle T \rangle \subseteq \langle S \rangle$
  ⟨*proof*⟩

lemma (**in** *Group*) *genby-closed*:
  assumes $S \subseteq G$
  shows $\langle S \rangle \subseteq G$
⟨*proof*⟩

lemma (**in** *Group*) *genby-subgroup*: $S \subseteq G \Longrightarrow$ *Subgroup* ⟨S⟩
  ⟨*proof*⟩

lemma *genby-sym-eq-sum-lists*:
  uminus ‘ $S \subseteq S$ ⟹ ⟨S⟩ = ($\bigcup$*ss*∈*lists* S. {*sum-list ss*})
  ⟨*proof*⟩

lemma *genby-empty′*: $w \in \langle \{\} \rangle \Longrightarrow w = 0$
⟨*proof*⟩

lemma *genby-order2′*:
  assumes $s+s=0$
  shows $w \in \langle \{s\} \rangle \Longrightarrow w = 0 \lor w = s$
⟨*proof*⟩

lemma *genby-order2*: $s+s=0 \Longrightarrow \langle \{s\} \rangle = \{0,s\}$
  ⟨*proof*⟩

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

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

lemma *genby-lcoset-empty*: (*w*::′*a*::*group-add*) +*o* ⟨{}⟩ = {*w*}

⟨*proof*⟩

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 $x \in \langle S\rangle \Longrightarrow lconjby\ g\ x \in \langle S\rangle$
⟨*proof*⟩

lemma (**in** *Group*) *genby-set-lconjby-set-rconjby-closed*:
fixes $A :: 'g\ set$
defines $S \equiv (\bigcup g{\in}G.\ lconjby\ g\ `\ A)$
assumes $g{\in}G\ x \in \langle S\rangle$
shows $rconjby\ g\ x \in \langle S\rangle$
⟨*proof*⟩

2.8.5 Homomorphisms and isomorphisms

locale *GroupHom* = *Group G*
for $G :: 'g::group\text{-}add\ set$
+ **fixes** $T :: 'g \Rightarrow 'h::group\text{-}add$
assumes *hom* : $g \in G \Longrightarrow g' \in G \Longrightarrow T\ (g + g') = T\ g + T\ g'$
and *supp*: $supp\ T \subseteq G$
begin

lemma *im-zero*: $T\ 0 = 0$
⟨*proof*⟩

lemma *im-uminus*: $T\ (-\ g) = -\ T\ g$
⟨*proof*⟩

lemma *im-uminus-add*: $g \in G \Longrightarrow g' \in G \Longrightarrow T\ (-g + g') = -\ T\ g + T\ g'$
⟨*proof*⟩

lemma *im-diff*: $g \in G \Longrightarrow g' \in G \Longrightarrow T\ (g - g') = T\ g - T\ g'$
⟨*proof*⟩

lemma *im-lconjby*: $x \in G \Longrightarrow g \in G \Longrightarrow T\ (lconjby\ x\ g) = lconjby\ (T\ x)\ (T\ g)$
⟨*proof*⟩

lemma *im-sum-list-map*:
$set\ (map\ f\ as) \subseteq G \Longrightarrow T\ (\sum a{\leftarrow}as.\ f\ a) = (\sum a{\leftarrow}as.\ T\ (f\ a))$
⟨*proof*⟩

lemma *comp*:
assumes $GroupHom\ H\ S\ T`G \subseteq H$
shows $GroupHom\ G\ (S \circ T)$
⟨*proof*⟩

end

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

lemma *ker-subset-ker-restrict0*: *ker f* $\subseteq$ *ker* (*restrict0 f A*)
⟨*proof*⟩

context *GroupHom*
begin

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

lemma *uminus-add-in-Ker-eq-eq-im*:
$g \in G \Longrightarrow h \in G \Longrightarrow (-g + h \in Ker) = (T\ g = T\ h)$
⟨*proof*⟩

end

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

lemmas *im-zero* = *im-zero*
lemmas *im-uminus* = *im-uminus*

lemma *hom*: $T\ (g+g') = T\ g + T\ g'$
⟨*proof*⟩

lemma *im-diff*: $T\ (g - g') = T\ g - T\ g'$
⟨*proof*⟩

lemma *im-lconjby*: *T* (*lconjby x g*) = *lconjby* (*T x*) (*T g*)
⟨*proof*⟩

lemma *restrict0*:
assumes *Group G*
shows *GroupHom G* (*restrict0 T G*)
⟨*proof*⟩

end

lemma *UGroupHomI*:
assumes $\bigwedge g\ g'.\ T\ (g + g') = T\ g + T\ g'$
shows *UGroupHom T*
⟨*proof*⟩

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*⟩

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

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

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

2.8.6 Normal subgroups

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

context *Group*
begin

lemma *rcoset-rel-conv-lcoset-rel*:
rcoset-rel G = *map-prod uminus uminus* ‘ (*lcoset-rel* G)
⟨*proof*⟩

lemma *rcoset-rel-sym*: *sym* (*rcoset-rel* G)
⟨*proof*⟩

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

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

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

lemma *normal-lconjby-closed*:
⟦ *Subgroup* H; *normal* H; $g \in G$; $h \in H$ ⟧ $\Longrightarrow$ *lconjby* g $h \in H$
⟨*proof*⟩

lemma *normal-rconjby-closed*:
⟦ *Subgroup H*; *normal H*; *g*∈*G*; *h*∈*H* ⟧ ⟹ *rconjby g h* ∈ *H*
⟨*proof*⟩

abbreviation *normal-closure A* ≡ ⟨⋃*g*∈*G*. *lconjby g* ' *A*⟩

lemma (**in** *Group*) *normal-closure*:
assumes *A*⊆*G*
shows *normal* (*normal-closure A*)
⟨*proof*⟩

end

2.8.7 Quotient groups

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

context *Group*
begin

lemma *normal-quotient-add-well-defined*:
assumes *Subgroup H normal H g*∈*G g*′∈*G*
shows *LCoset-rel H* '' {*g*} + *LCoset-rel H* '' {*g*′} = *LCoset-rel H* '' {*g*+*g*′}
⟨*proof*⟩

abbreviation *quotient-set H* ≡ *G* // *LCoset-rel H*

lemma *BinOpSetGroup-normal-quotient*:
assumes *Subgroup H normal H*
shows *BinOpSetGroup* (*quotient-set H*) (+) *H*
⟨*proof*⟩

abbreviation *abs-lcoset-perm H* ≡
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* ≡ *abs-lcoset-perm-lift H g*

notation *abs-lcoset-perm-lift-arg-permutation* (‹⌈-|-⌉› [*51*,*51*] *50*)

end

abbreviation *Group-abs-lcoset-perm-lift-arg-permutation G*′ *g H* ≡
Group.abs-lcoset-perm-lift-arg-permutation G′ *g H*
notation *Group-abs-lcoset-perm-lift-arg-permutation* (‹⌈-|-|-⌉› [*51*,*51*,*51*] *50*)

context *Group*
begin

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

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

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

abbreviation *natural-quotient-hom H* ≡ *restrict0* (λ*g*. ⌈*g*|*H*⌉) *G*

theorem *quotient-group*:
  Subgroup H ⟹ *normal H* ⟹ *Group* (*quotient-group H*)
  ⟨*proof*⟩

lemma *natural-quotient-hom*:
  Subgroup H ⟹ *normal H* ⟹ *GroupHom G* (*natural-quotient-hom H*)
  ⟨*proof*⟩

lemma *natural-quotient-hom-image*:
  natural-quotient-hom H ‘ *G* = *quotient-group H*
  ⟨*proof*⟩

lemma *quotient-group-UN*: *quotient-group H* = (λ*g*. ⌈*g*|*H*⌉) ‘ *G*
  ⟨*proof*⟩

lemma *quotient-identity-rule*: ⟦ *Subgroup H*; *normal H*; *h*∈*H* ⟧ ⟹ ⌈*h*|*H*⌉ = *0*
  ⟨*proof*⟩

lemma *quotient-group-lift-to-quotient-set*:
  ⟦ *Subgroup H*; *normal H*; *g*∈*G* ⟧ ⟹ (⌈*g*|*H*⌉) → *H* = *LCoset-rel H* ‘‘ {*g*}
  ⟨*proof*⟩

end

2.8.8 The induced homomorphism on a quotient group

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

context *GroupHom*

begin

lemma *respects-Ker-lcosets*: $H \subseteq Ker \Longrightarrow T$ *respects* (*LCoset-rel H*)
⟨*proof*⟩

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*; $H \subseteq Ker$; $g \in G$ ⟧ ⟹ *quotient-hom H* (⌈*g*|*H*⌉) = *T g*
⟨*proof*⟩

lemma *quotient-hom*:
assumes *Subgroup H normal H* $H \subseteq Ker$
shows *GroupHom* (*quotient-group H*) (*quotient-hom H*)
⟨*proof*⟩

end

2.9 Free groups

2.9.1 Words in letters of *signed* type

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

abbreviation *pairtrue* ≡ λ*s*. (*s*,*True*)
abbreviation *pairfalse* ≡ λ*s*. (*s*,*False*)

abbreviation *flip-signed* :: ′*a signed* ⇒ ′*a signed*
where *flip-signed* ≡ *apsnd* (λ*b*. ¬*b*)

abbreviation *nflipped-signed* :: ′*a signed* ⇒ ′*a signed* ⇒ *bool*
where *nflipped-signed x y* ≡ *y* ≠ *flip-signed x*

lemma *flip-signed-order2*: *flip-signed* (*flip-signed x*) = *x*
⟨*proof*⟩

abbreviation *charpair* :: ′*a*::*uminus set* ⇒ ′*a* ⇒ ′*a signed*
where *charpair S s* ≡ *if s*∈*S then* (*s*,*True*) *else* (−*s*,*False*)

lemma *map-charpair-uniform*:
ss∈*lists S* ⟹ *map* (*charpair S*) *ss* = *map pairtrue ss*
⟨*proof*⟩

lemma *fst-set-map-charpair-un-uminus*:
  fixes *ss* :: *'a::group-add list*
  shows *ss*∈*lists* (*S* ∪ *uminus* ' *S*) ⟹ *fst* ' *set* (*map* (*charpair S*) *ss*) ⊆ *S*
  ⟨*proof*⟩

abbreviation *apply-sign* :: (*'a*⇒*'b*::*uminus*) ⇒ *'a signed* ⇒ *'b*
  where *apply-sign f x* ≡ (*if snd x then f* (*fst x*) *else* − *f* (*fst x*))

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

abbreviation *proper-signed-list* :: *'a signed list* ⇒ *bool*
  where *proper-signed-list* ≡ *binrelchain nflipped-signed*

lemma *proper-map-flip-signed*:
  proper-signed-list xs ⟹ *proper-signed-list* (*map flip-signed xs*)
  ⟨*proof*⟩

lemma *proper-rev-map-flip-signed*:
  proper-signed-list xs ⟹ *proper-signed-list* (*rev* (*map flip-signed xs*))
  ⟨*proof*⟩

lemma *uniform-snd-imp-proper-signed-list*:
  snd ' *set xs* ⊆ {*b*} ⟹ *proper-signed-list xs*
⟨*proof*⟩

lemma *proper-signed-list-map-uniform-snd*:
  proper-signed-list (*map* (λ*s*. (*s*,*b*)) *as*)
  ⟨*proof*⟩

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

function *prappend-signed-list* :: *'a signed list* ⇒ *'a signed list* ⇒ *'a signed list*
  where *prappend-signed-list xs* [] = *xs*
    | *prappend-signed-list* [] *ys* = *ys*
    | *prappend-signed-list* (*xs*@[*x*]) (*y*#*ys*) = (
      if y = *flip-signed x then prappend-signed-list xs ys else xs* @ *x* # *y* # *ys*
)
  ⟨*proof*⟩
  termination ⟨*proof*⟩

lemma *proper-prappend-signed-list*:
  proper-signed-list xs ⟹ *proper-signed-list ys*
    ⟹ *proper-signed-list* (*prappend-signed-list xs ys*)
⟨*proof*⟩

lemma *fully-prappend-signed-list*:
  prappend-signed-list (*rev* (*map flip-signed xs*)) *xs* = []
  ⟨*proof*⟩

lemma *prappend-signed-list-single-Cons*:
  prappend-signed-list [*x*] (*y*#*ys*) = (*if y* = *flip-signed x then ys else x*#*y*#*ys*)
  ⟨*proof*⟩

lemma *prappend-signed-list-map-uniform-snd*:
  prappend-signed-list (*map* (λ*s*. (*s*,*b*)) *xs*) (*map* (λ*s*. (*s*,*b*)) *ys*) =
    map (λ*s*. (*s*,*b*)) *xs* @ *map* (λ*s*. (*s*,*b*)) *ys*
  ⟨*proof*⟩

lemma *prappend-signed-list-assoc-conv-snoc2Cons*:
  assumes *proper-signed-list* (*xs*@[*y*]) *proper-signed-list* (*y*#*ys*)
  shows *prappend-signed-list* (*xs*@[*y*]) *ys* = *prappend-signed-list xs* (*y*#*ys*)
⟨*proof*⟩

lemma *prappend-signed-list-assoc*:
  ⟦ *proper-signed-list xs*; *proper-signed-list ys*; *proper-signed-list zs* ⟧ ⟹
    prappend-signed-list (*prappend-signed-list xs ys*) *zs* =
      prappend-signed-list xs (*prappend-signed-list ys zs*)
⟨*proof*⟩

lemma *fst-set-prappend-signed-list*:
  fst ‘ *set* (*prappend-signed-list xs ys*) ⊆ *fst* ‘ (*set xs* ∪ *set ys*)
  ⟨*proof*⟩

lemma *collapse-flipped-signed*:
  prappend-signed-list [(*s*,*b*)] [(*s*,¬*b*)] = []
  ⟨*proof*⟩

2.9.2 The collection of proper signed lists as a type

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

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

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

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

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

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

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

setup-lifting *type-definition-freeword*

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

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

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

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

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

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

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

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

abbreviation *Abs-freeletter-add* :: *'a* ⇒ *'a* ⇒ *'a freeword* (**infixl** ‹[+]› *65*)
where *s* [+] *t* ≡ *Abs-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*⟩

lemma *Abs-freelist-Cons*: *Abs-freelist* (*x*#*xs*) = *Abs-freeletter x* + *Abs-freelist xs*
⟨*proof*⟩

lemma *plus-freeword-abs-eq*:
proper-signed-list xs ⟹ *proper-signed-list ys* ⟹
Abs-freeword xs + *Abs-freeword ys* = *Abs-freeword* (*prappend-signed-list xs ys*)
⟨*proof*⟩

lemma *Abs-freeletter-add*: *s* [+] *t* = *Abs-freelist* [*s*,*t*]
⟨*proof*⟩

lemma *uminus-freeword-Abs-eq*:
proper-signed-list xs ⟹
− *Abs-freeword xs* = *Abs-freeword* (*rev* (*map flip-signed xs*))
⟨*proof*⟩

lemma *uminus-Abs-freeword-singleton*:
− *Abs-freeword* [(*s*,*b*)] = *Abs-freeword* [(*s*,¬ *b*)]
⟨*proof*⟩

lemma *Abs-freeword-append-uniform-snd*:
Abs-freeword (*map* (λ*s*. (*s*,*b*)) (*xs*@*ys*)) =
Abs-freeword (*map* (λ*s*. (*s*,*b*)) *xs*) + *Abs-freeword* (*map* (λ*s*. (*s*,*b*)) *ys*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Abs-freelist-inverse*: *freeword* (*Abs-freelist as*) = *map pairtrue as*
⟨*proof*⟩

lemma *Abs-freeword-singleton-conv-apply-sign-freeletter*:
Abs-freeword [*x*] = *apply-sign Abs-freeletter x*
⟨*proof*⟩

lemma *Abs-freeword-conv-freeletter-sum-list*:
proper-signed-list xs $\Longrightarrow$
Abs-freeword xs = ($\sum$ *x*←*xs*. *apply-sign Abs-freeletter x*)
⟨*proof*⟩

lemma *freeword-conv-freeletter-sum-list*:
x = ($\sum$ *s*←*freeword x*. *apply-sign Abs-freeletter s*)
⟨*proof*⟩

lemma *Abs-freeletter-prod-conv-Abs-freeword*:
snd x $\Longrightarrow$ *Abs-freeletter* (*fst x*) = *Abs-freeword* [*x*]
⟨*proof*⟩

2.9.3 Lifts of functions on the letter type

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

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

lemma *extend-map-to-freeword-hom'*:
fixes *f* :: '*a* ⇒ '*b*::*group-add*
defines *h*: *h*::'*a signed* ⇒ '*b* ≡ λ(*s*,*b*). *if b then f s else* − (*f s*)
defines *g*: *g*::'*a signed list* ⇒ '*b* ≡ λ*xs*. *sum-list* (*map h xs*)
shows *g* (*prappend-signed-list xs ys*) = *g xs* + *g ys*
⟨*proof*⟩

lemma *extend-map-to-freeword-hom1*:
fixes *f* :: '*a* ⇒ '*b*::*group-add*
defines *h*::'*a signed* ⇒ '*b* ≡ λ(*s*,*b*). *if b then f s else* − (*f s*)
defines *g*::'*a freeword* ⇒ '*b* ≡ λ*x*. *sum-list* (*map h* (*freeword x*))
shows *g* (*Abs-freeletter s*) = *f s*
⟨*proof*⟩

lemma *extend-map-to-freeword-hom2*:
fixes *f* :: '*a* ⇒ '*b*::*group-add*
defines *h*::'*a signed* ⇒ '*b* ≡ λ(*s*,*b*). *if b then f s else* − (*f s*)
defines *g*::'*a freeword* ⇒ '*b* ≡ λ*x*. *sum-list* (*map h* (*freeword x*))
shows *UGroupHom g*
⟨*proof*⟩

lemma *uniqueness-of-extended-map-to-freeword-hom'*:
fixes *f* :: '*a* ⇒ '*b*::*group-add*
defines *h*: *h*::'*a signed* ⇒ '*b* ≡ λ(*s*,*b*). *if b then f s else* − (*f s*)

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

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

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

Properties of homomorphisms afforded by the universal property

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

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

lemma *additive-freeword-funlift*: *UGroupHom* (*freeword-funlift* f)
⟨*proof*⟩

lemma *freeword-funlift-Abs-freeletter*: *freeword-funlift* f (*Abs-freeletter* s) = f s
⟨*proof*⟩

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*)]) = $-$ f s
⟨*proof*⟩

lemma *freeword-funlift-Abs-freeword-singleton*:
freeword-funlift f (*Abs-freeword* [x]) = *apply-sign* f x
⟨*proof*⟩

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

shows *freeword-funlift f (Abs-freeword (x#xs)) =*
apply-sign f x + freeword-funlift f (Abs-freeword xs)
⟨*proof*⟩

lemma *freeword-funlift-Abs-freeword*:
proper-signed-list xs ⟹ *freeword-funlift f (Abs-freeword xs) =*
(∑ x←xs. apply-sign f x)
⟨*proof*⟩

lemma *freeword-funlift-Abs-freelist*:
freeword-funlift f (Abs-freelist xs) = (∑ x←xs. f x)
⟨*proof*⟩

lemma *freeword-funlift-im′*:
proper-signed-list xs ⟹ *fst ‘ set xs* ⊆ *S* ⟹
freeword-funlift f (Abs-freeword xs) ∈ ⟨*f‘S*⟩
⟨*proof*⟩

2.9.4 Free groups on a set

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

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

definition *FreeGroup* :: *′a set* ⇒ *′a freeword set*
where *FreeGroup S* ≡ {*x. fst ‘ set (freeword x)* ⊆ *S*}

lemma *FreeGroupI-transfer*:
proper-signed-list xs ⟹ *fst ‘ set xs* ⊆ *S* ⟹ *Abs-freeword xs* ∈ *FreeGroup S*
⟨*proof*⟩

lemma *FreeGroupD*: *x* ∈ *FreeGroup S* ⟹ *fst ‘ set (freeword x)* ⊆ *S*
⟨*proof*⟩

lemma *FreeGroupD-transfer*:
proper-signed-list xs ⟹ *Abs-freeword xs* ∈ *FreeGroup S* ⟹ *fst ‘ set xs* ⊆ *S*
⟨*proof*⟩

lemma *FreeGroupD-transfer′*:
Abs-freelist xs ∈ *FreeGroup S* ⟹ *xs* ∈ *lists S*
⟨*proof*⟩

lemma *FreeGroup-0-closed*: *0* ∈ *FreeGroup S*
⟨*proof*⟩

lemma *FreeGroup-diff-closed*:
assumes *x* ∈ *FreeGroup S y* ∈ *FreeGroup S*

shows $x-y \in$ *FreeGroup S*
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *freeword-funlift-im*: *x* ∈ *FreeGroup S* ⟹ *freeword-funlift f x* ∈ ⟨*f* ‘ *S*⟩
⟨*proof*⟩

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

lemma *freeword-funlift-surj*:
fixes *f* :: *'a* ⇒ *'b*::*group-add*
shows *freeword-funlift f* ‘ *FreeGroup S* = ⟨*f*‘*S*⟩
⟨*proof*⟩

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

lemma *uniqueness-of-restricted-lift*:
assumes *GroupHom* (*FreeGroup S*) *T* ∀ *s*∈*S*. *T* (*Abs-freeletter s*) = *f s*
shows *T* = *res-freeword-funlift f S*
⟨*proof*⟩

theorem *FreeGroup-universal-property*:
fixes *f* :: *'a* ⇒ *'b*::*group-add*
shows ∃!*T*::*'a freeword*⇒*'b*. (∀ *s*∈*S*. *T* (*Abs-freeletter s*) = *f s*) ∧

GroupHom (FreeGroup S) T
⟨*proof*⟩

2.9.5 Group presentations

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

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

locale *GroupByPresentation* =
 fixes *S* :: *'a set* — the set of generators
 and *P* :: *'a signed list set* — the set of relator words
 assumes *P-S*: *ps*∈*P* ⟹ *fst* ' *set ps* ⊆ *S*
 and *proper-P*: *ps*∈*P* ⟹ *proper-signed-list ps*
begin

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

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

lemma *P'-FreeS*: *P'* ⊆ *FreeGroup S*
 ⟨*proof*⟩

lemma *relators*: *P'* ⊆ *Q*
 ⟨*proof*⟩

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

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

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

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

lemmas *natural-hom* =

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

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

end

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

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

abbreviation *lift-f* ≡ *freeword-funlift f*

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

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

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

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

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

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

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

⟨*proof*⟩

lemma *induced-hom-equality*:
w ∈ *FreeGroup S* ⟹ *F* (⌈*FreeGroup S*|*w*|*Q*⌉) = *lift-f w*
— algebraic properties of the induced homomorphism could be proved using its properties as a group homomorphism, but it's generally easier to prove them using the algebraic properties of the lift via this lemma
⟨*proof*⟩

lemma *hom-induced-hom*: *GroupHom G F*
⟨*proof*⟩

lemma *induced-hom-Abs-freeletter-equality*:
s∈*S* ⟹ *F* (⌈*FreeGroup S*|*Abs-freeletter s*|*Q*⌉) = *f s*
⟨*proof*⟩

lemma *uniqueness-of-induced-hom′*:
defines *q* ≡ *Group.natural-quotient-hom* (*FreeGroup S*) *Q*
assumes *GroupHom G T* ∀ *s*∈*S*. *T* (⌈*FreeGroup S*|*Abs-freeletter s*|*Q*⌉) = *f s*
shows *T* ∘ *q* = *F* ∘ *q*
⟨*proof*⟩

lemma *uniqueness-of-induced-hom*:
assumes *GroupHom G T* ∀ *s*∈*S*. *T* (⌈*FreeGroup S*|*Abs-freeletter s*|*Q*⌉) = *f s*
shows *T* = *F*
⟨*proof*⟩

theorem *induced-hom-universal-property*:
∃!*F*. *GroupHom G F* ∧ (∀ *s*∈*S*. *F* (⌈*FreeGroup S*|*Abs-freeletter s*|*Q*⌉) = *f s*)
⟨*proof*⟩

lemma *induced-hom-Abs-freelist-conv-sum-list*:
ss∈*lists S* ⟹ *F* (⌈*FreeGroup S*|*Abs-freelist ss*|*Q*⌉) = (∑ *s*←*ss*. *f s*)
⟨*proof*⟩

lemma *induced-hom-surj*: *F*‘*G* = ⟨*f*‘*S*⟩
⟨*proof*⟩

end

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

locale *GroupWithGeneratorsRelators* =
fixes *S* :: ′*g*::*group-add set* — the set of generators
and *R* :: ′*g list set* — the set of relator words

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

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

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

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

lemma *GroupByPresentation-S-P*: *GroupByPresentation S P*
⟨*proof*⟩

lemmas *G-UN* = *GroupByPresentation.G-UN*[*OF GroupByPresentation-S-P*]
lemmas *P′-FreeS* = *GroupByPresentation.P′-FreeS*[*OF GroupByPresentation-S-P*]

lemma *freeliftid-trivial-relator-freeword-R*:
$rs{\in}R \Longrightarrow freeliftid\ (relator\text{-}freeword\ rs) = 0$
⟨*proof*⟩

lemma *freeliftid-trivial-P*: $ps{\in}P \Longrightarrow freeliftid\ (Abs\text{-}freeword\ ps) = 0$
⟨*proof*⟩

lemma *GroupByPresentationInducedFun-S-P-id*:
GroupByPresentationInducedFun S P id
⟨*proof*⟩

lemma *induced-id-Abs-freelist-conv-sum-list*:
$ss{\in}lists\ S \Longrightarrow induced\text{-}id\ (\lceil FreeGroup\ S|Abs\text{-}freelist\ ss|Q\rceil) = sum\text{-}list\ ss$
⟨*proof*⟩

lemma *lconj-relator-freeword-R*:
$[\![\ rs{\in}R;\ proper\text{-}signed\text{-}list\ xs;\ fst\ `\ set\ xs \subseteq S\ ]\!] \Longrightarrow$
$lconjby\ (Abs\text{-}freeword\ xs)\ (relator\text{-}freeword\ rs) \in Q$
⟨*proof*⟩

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

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

$[\![$ *rs*∈*R*; *xs*∈*lists S* $]\!]$ ⟹ *lconjby* (*Abs-freelist xs*) (*relator-freeword rs*) ∈ *Q*
⟨*proof*⟩

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

theorem *induced-id-hom-surj*: *GroupHom G induced-id induced-id ‘ G* = ⟨*S*⟩
⟨*proof*⟩

end

locale *GroupPresentation* = *GroupWithGeneratorsRelators 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* ≡ *the-inv-into G induced-id*

lemma *inv-induced-id-sum-list-S*:
ss ∈ *lists S* ⟹ *inv-induced-id* (*sum-list ss*) = (⌈*FreeGroup S*|*Abs-freelist ss*|*Q*⌉)
⟨*proof*⟩

end

2.10 Words over a generating set

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

context *monoid-add*
begin

abbreviation *word-for A a as* ≡ *as* ∈ *lists A* ∧ *sum-list as* = *a*

definition *reduced-word-for* :: *'a set* ⇒ *'a* ⇒ *'a list* ⇒ *bool*
where *reduced-word-for A a as* ≡ *is-arg-min length* (*word-for A a*) *as*

abbreviation *reduced-word A as* ≡ *reduced-word-for A* (*sum-list as*) *as*
abbreviation *reduced-words-for A a* ≡ *Collect* (*reduced-word-for A a*)

abbreviation *reduced-letter-set* :: *'a set* ⇒ *'a* ⇒ *'a set*
where *reduced-letter-set A a* ≡ ⋃(*set* ‘ (*reduced-words-for A a*))
— will be empty if *a* is not in the set generated by *A*

definition *word-length* :: *'a set* ⇒ *'a* ⇒ *nat*
where *word-length A a* ≡ *length* (*arg-min length* (*word-for A a*))

lemma *reduced-word-forI*:

assumes *as* ∈ *lists A sum-list as* = *a*
⋀*bs*. *bs* ∈ *lists A* ⟹ *sum-list bs* = *a* ⟹ *length as* ≤ *length bs*
shows *reduced-word-for A a as*
⟨*proof*⟩

lemma *reduced-word-forI-compare*:
⟦ *reduced-word-for A a as*; *bs* ∈ *lists A*; *sum-list bs* = *a*; *length bs* = *length as* ⟧
⟹ *reduced-word-for A a bs*
⟨*proof*⟩

lemma *reduced-word-for-lists*: *reduced-word-for A a as* ⟹ *as* ∈ *lists A*
⟨*proof*⟩

lemma *reduced-word-for-sum-list*: *reduced-word-for A a as* ⟹ *sum-list as* = *a*
⟨*proof*⟩

lemma *reduced-word-for-minimal*:
⟦ *reduced-word-for A a as*; *bs* ∈ *lists A*; *sum-list bs* = *a* ⟧ ⟹
length as ≤ *length bs*
⟨*proof*⟩

lemma *reduced-word-for-length*:
reduced-word-for A a as ⟹ *length as* = *word-length A a*
⟨*proof*⟩

lemma *reduced-word-for-eq-length*:
reduced-word-for A a as ⟹ *reduced-word-for A a bs* ⟹ *length as* = *length bs*
⟨*proof*⟩

lemma *reduced-word-for-arg-min*:
as ∈ *lists A* ⟹ *sum-list as* = *a* ⟹
reduced-word-for A a (*arg-min length* (*word-for A a*))
⟨*proof*⟩

lemma *nil-reduced-word-for-0*: *reduced-word-for A 0* []
⟨*proof*⟩

lemma *reduced-word-for-0-imp-nil*: *reduced-word-for A 0 as* ⟹ *as* = []
⟨*proof*⟩

lemma *not-reduced-word-for*:
⟦ *bs* ∈ *lists A*; *sum-list bs* = *a*; *length bs* < *length as* ⟧ ⟹
¬ *reduced-word-for A a as*
⟨*proof*⟩

lemma *reduced-word-for-imp-reduced-word*:
reduced-word-for A a as ⟹ *reduced-word A as*
⟨*proof*⟩

lemma *sum-list-zero-nreduced*:
  $as \neq [] \Longrightarrow$ *sum-list as* $= 0 \Longrightarrow \neg$ *reduced-word A as*
  ⟨*proof*⟩

lemma *order2-nreduced*: $a+a=0 \Longrightarrow \neg$ *reduced-word A* $[a,a]$
  ⟨*proof*⟩

lemma *reduced-word-append-reduce-contra1*:
  assumes ¬ *reduced-word A as*
  shows ¬ *reduced-word A* (*as@bs*)
⟨*proof*⟩

lemma *reduced-word-append-reduce-contra2*:
  assumes ¬ *reduced-word A bs*
  shows ¬ *reduced-word A* (*as@bs*)
⟨*proof*⟩

lemma *contains-nreduced-imp-nreduced*:
  ¬ *reduced-word A bs* ⟹ ¬ *reduced-word A* (*as@bs@cs*)
  ⟨*proof*⟩

lemma *contains-order2-nreduced*: $a+a=0 \Longrightarrow \neg$ *reduced-word A* (*as*@$[a,a]$@*bs*)
  ⟨*proof*⟩

lemma *reduced-word-Cons-reduce-contra*:
  ¬ *reduced-word A as* ⟹ ¬ *reduced-word A* (*a#as*)
  ⟨*proof*⟩

lemma *reduced-word-Cons-reduce*: *reduced-word A* (*a#as*) ⟹ *reduced-word A as*
  ⟨*proof*⟩

lemma *reduced-word-singleton*:
  assumes $a \in A$ $a \neq 0$
  shows *reduced-word A* $[a]$
⟨*proof*⟩

lemma *el-reduced*:
  assumes $0 \notin A$ $as \in$ *lists A sum-list as* $\in A$ *reduced-word A as*
  shows *length as* $= 1$
⟨*proof*⟩

lemma *reduced-letter-set-0*: *reduced-letter-set A 0* $= \{\}$
  ⟨*proof*⟩

lemma *reduced-letter-set-subset*: *reduced-letter-set A a* $\subseteq A$
  ⟨*proof*⟩

lemma *reduced-word-forI-length*:
  ⟦ $as \in$ *lists A*; *sum-list as* $= a$; *length as* $=$ *word-length A a* ⟧ ⟹

reduced-word-for A a as
⟨*proof*⟩

lemma *word-length-le*:
as ∈ *lists A* ⟹ *sum-list as* = *a* ⟹ *word-length A a* ≤ *length as*
⟨*proof*⟩

lemma *reduced-word-forI-length′*:
⟦ *as* ∈ *lists A*; *sum-list as* = *a*; *length as* ≤ *word-length A a* ⟧ ⟹
reduced-word-for A a as
⟨*proof*⟩

lemma *word-length-lt*:
as ∈ *lists A* ⟹ *sum-list as* = *a* ⟹ ¬ *reduced-word-for A a as* ⟹
word-length A a < *length as*
⟨*proof*⟩

end

lemma *in-genby-reduced-letter-set*:
assumes *as* ∈ *lists A sum-list as* = *a*
shows *a* ∈ ⟨*reduced-letter-set A a*⟩
⟨*proof*⟩

lemma *reduced-word-for-genby-arg-min*:
fixes *A* :: ′*a*::*group-add set*
defines *B* ≡ *A* ∪ *uminus* ‘ *A*
assumes *a*∈⟨*A*⟩
shows *reduced-word-for B a* (*arg-min length* (*word-for B a*))
⟨*proof*⟩

lemma *reduced-word-for-genby-sym-arg-min*:
assumes *uminus* ‘ *A* ⊆ *A a*∈⟨*A*⟩
shows *reduced-word-for A a* (*arg-min length* (*word-for A a*))
⟨*proof*⟩

lemma *in-genby-imp-in-reduced-letter-set*:
fixes *A* :: ′*a*::*group-add set*
defines *B* ≡ *A* ∪ *uminus* ‘ *A*
assumes *a* ∈ ⟨*A*⟩
shows *a* ∈ ⟨*reduced-letter-set B a*⟩
⟨*proof*⟩

lemma *in-genby-sym-imp-in-reduced-letter-set*:
uminus ‘ *A* ⊆ *A* ⟹ *a* ∈ ⟨*A*⟩ ⟹ *a* ∈ ⟨*reduced-letter-set A a*⟩
⟨*proof*⟩

end

3 Simplicial complexes

In this section we develop the basic theory of abstract simplicial complexes as a collection of finite sets, where the power set of each member set is contained in the collection. Note that in this development we allow the empty simplex, since allowing it or not seemed of no logical consequence, but of some small practical consequence.

theory *Simplicial*
imports *Prelim*

begin

3.1 Geometric notions

The geometric notions attached to a simplicial complex of main interest to us are those of facets (subsets of codimension one), adjacency (sharing a facet in common), and chains of adjacent simplices.

3.1.1 Facets

definition *facetrel* :: $'a\ set \Rightarrow 'a\ set \Rightarrow bool$ (**infix** ‹◁› *60*)
where $y \lhd x \equiv \exists v.\ v \notin y \wedge x = insert\ v\ y$

lemma *facetrelI*: $v \notin y \Longrightarrow x = insert\ v\ y \Longrightarrow y \lhd x$
⟨*proof*⟩

lemma *facetrelI-card*: $y \subseteq x \Longrightarrow card\ (x-y) = 1 \Longrightarrow y \lhd x$
⟨*proof*⟩

lemma *facetrel-complement-vertex*: $y \lhd x \Longrightarrow x = insert\ v\ y \Longrightarrow v \notin y$
⟨*proof*⟩

lemma *facetrel-diff-vertex*: $v \in x \Longrightarrow x-\{v\} \lhd x$
⟨*proof*⟩

lemma *facetrel-conv-insert*: $y \lhd x \Longrightarrow v \in x - y \Longrightarrow x = insert\ v\ y$
⟨*proof*⟩

lemma *facetrel-psubset*: $y \lhd x \Longrightarrow y \subset x$
⟨*proof*⟩

lemma *facetrel-subset*: $y \lhd x \Longrightarrow y \subseteq x$
⟨*proof*⟩

lemma *facetrel-card*: $y \lhd x \Longrightarrow card\ (x-y) = 1$
⟨*proof*⟩

lemma *finite-facetrel-card*: $finite\ x \Longrightarrow y \lhd x \Longrightarrow card\ x = Suc\ (card\ y)$

⟨*proof*⟩

lemma *facetrelI-cardSuc*: $z \subseteq x \Longrightarrow$ *card* x = *Suc* (*card* z) $\Longrightarrow z \lhd x$
⟨*proof*⟩

lemma *facet2-subset*: $[\![z \lhd x; z \lhd y; x \cap y - z \neq \{\}]\!] \Longrightarrow x \subseteq y$
⟨*proof*⟩

lemma *inj-on-pullback-facet*:
assumes *inj-on* f x $z \lhd f`x$
obtains y **where** $y \lhd x$ $f`y = z$
⟨*proof*⟩

3.1.2 Adjacency

definition *adjacent* :: $'a$ *set* $\Rightarrow$ $'a$ *set* $\Rightarrow$ *bool* (**infix** ‹∼› *70*)
where $x \sim y \equiv \exists z.\ z \lhd x \wedge z \lhd y$

lemma *adjacentI*: $z \lhd x \Longrightarrow z \lhd y \Longrightarrow x \sim y$
⟨*proof*⟩

lemma *empty-not-adjacent*: $\neg \{\} \sim x$
⟨*proof*⟩

lemma *adjacent-sym*: $x \sim y \Longrightarrow y \sim x$
⟨*proof*⟩

lemma *adjacent-refl*:
assumes $x \neq \{\}$
shows $x \sim x$
⟨*proof*⟩

lemma *common-facet*: $[\![z \lhd x; z \lhd y; x \neq y]\!] \Longrightarrow z = x \cap y$
⟨*proof*⟩

lemma *adjacent-int-facet1*: $x \sim y \Longrightarrow x \neq y \Longrightarrow (x \cap y) \lhd x$
⟨*proof*⟩

lemma *adjacent-int-facet2*: $x \sim y \Longrightarrow x \neq y \Longrightarrow (x \cap y) \lhd y$
⟨*proof*⟩

lemma *adjacent-conv-insert*: $x \sim y \Longrightarrow v \in x - y \Longrightarrow x =$ *insert* v $(x \cap y)$
⟨*proof*⟩

lemma *adjacent-int-decomp*:
$x \sim y \Longrightarrow x \neq y \Longrightarrow \exists v.\ v \notin y \wedge x =$ *insert* v $(x \cap y)$
⟨*proof*⟩

lemma *adj-antivertex*:

assumes $x{\sim}y$ $x{\neq}y$
shows $\exists!v.\ v{\in}x{-}y$
⟨*proof*⟩

lemma *adjacent-card*: $x \sim y \Longrightarrow card\ x = card\ y$
⟨*proof*⟩

lemma *adjacent-to-adjacent-int-subset*:
assumes $C \sim D$ $f`C \sim f`D$ $f`C \neq f`D$
shows $f`C \cap f`D \subseteq f`(C{\cap}D)$
⟨*proof*⟩

lemma *adjacent-to-adjacent-int*:
$[\![\ C \sim D;\ f`C \sim f`D;\ f`C \neq f`D\ ]\!] \Longrightarrow f`(C{\cap}D) = f`C \cap f`D$
⟨*proof*⟩

3.1.3 Chains of adjacent sets

abbreviation *adjacentchain* ≡ *binrelchain adjacent*
abbreviation *padjacentchain* ≡ *proper-binrelchain adjacent*

lemmas *adjacentchain-Cons-reduce* = *binrelchain-Cons-reduce* [*of adjacent*]
lemmas *adjacentchain-obtain-proper* = *binrelchain-obtain-proper* [*of - - adjacent*]

lemma *adjacentchain-card*: *adjacentchain* $(x\#xs@[y]) \Longrightarrow card\ x = card\ y$
⟨*proof*⟩

3.2 Locale and basic facts

locale *SimplicialComplex* =
fixes X :: *'a set set*
assumes *finite-simplices*: $\forall x{\in}X.$ *finite* x
and *faces* : $x{\in}X \Longrightarrow y{\subseteq}x \Longrightarrow y{\in}X$

context *SimplicialComplex*
begin

abbreviation *Subcomplex* $Y \equiv Y \subseteq X \wedge$ *SimplicialComplex* Y

definition *maxsimp* $x \equiv x{\in}X \wedge (\forall z{\in}X.\ x{\subseteq}z \longrightarrow z{=}x)$

definition *adjacentset* :: *'a set* ⇒ *'a set set*
where *adjacentset* $x = \{y{\in}X.\ x{\sim}y\}$

lemma *finite-simplex*: $x{\in}X \Longrightarrow$ *finite* x
⟨*proof*⟩

lemma *singleton-simplex*: $v{\in}\bigcup X \Longrightarrow \{v\} \in X$
⟨*proof*⟩

lemma *maxsimpI*: $x \in X \Longrightarrow (\bigwedge z.\ z{\in}X \Longrightarrow x{\subseteq}z \Longrightarrow z{=}x) \Longrightarrow maxsimp\ x$
⟨*proof*⟩

lemma *maxsimpD-simplex*: $maxsimp\ x \Longrightarrow x{\in}X$
⟨*proof*⟩

lemma *maxsimpD-maximal*: $maxsimp\ x \Longrightarrow z{\in}X \Longrightarrow x{\subseteq}z \Longrightarrow z{=}x$
⟨*proof*⟩

lemmas *finite-maxsimp* = *finite-simplex*[*OF maxsimpD-simplex*]

lemma *maxsimp-nempty*: $X \neq \{\{\}\} \Longrightarrow maxsimp\ x \Longrightarrow x \neq \{\}$
⟨*proof*⟩

lemma *maxsimp-vertices*: $maxsimp\ x \Longrightarrow x{\subseteq}\bigcup X$
⟨*proof*⟩

lemma *adjacentsetD-adj*: $y \in adjacentset\ x \Longrightarrow x{\sim}y$
⟨*proof*⟩

lemma *max-in-subcomplex*:
$[\![\ Subcomplex\ Y;\ y \in Y;\ maxsimp\ y\ ]\!] \Longrightarrow SimplicialComplex.maxsimp\ Y\ y$
⟨*proof*⟩

lemma *face-im*:
assumes $w \in X\ y \subseteq f`w$
defines $u \equiv \{a{\in}w.\ f\ a \in y\}$
shows $y \in f{\vdash}X$
⟨*proof*⟩

lemma *im-faces*: $x \in f \vdash X \Longrightarrow y \subseteq x \Longrightarrow y \in f \vdash X$
⟨*proof*⟩

lemma *map-is-simplicial-morph*: *SimplicialComplex* $(f{\vdash}X)$
⟨*proof*⟩

lemma *vertex-set-int*:
assumes *SimplicialComplex Y*
shows $\bigcup(X{\cap}Y) = \bigcup X \cap \bigcup Y$
⟨*proof*⟩

end

3.3 Chains of maximal simplices

Chains of maximal simplices (with respect to adjacency) will allow us to walk through chamber complexes. But there is much we can say about them in simplicial complexes. We will call a chain of maximal simplices proper (using the prefix *p* as a naming convention to denote proper) if no maximal simplex appears more than once in the chain. (Some sources elect to call improper chains prechains, and reserve the name chain to describe a proper chain. And usually a slightly weaker notion of proper is used, requiring only that no maximal simplex appear twice in succession. But it essentially makes no difference, and we found it easier to use *distinct* rather than *binrelchain* (≠).)

context *SimplicialComplex*
begin

definition *maxsimpchain xs* ≡ (∀ *x*∈*set xs*. *maxsimp x*) ∧ *adjacentchain xs*
definition *pmaxsimpchain xs* ≡ (∀ *x*∈*set xs*. *maxsimp x*) ∧ *padjacentchain xs*

function *min-maxsimpchain* :: *'a set list* ⇒ *bool*
  where
    min-maxsimpchain [] = *True*
  | *min-maxsimpchain* [*x*] = *maxsimp x*
  | *min-maxsimpchain* (*x*#*xs*@[*y*]) =
      (*x*≠*y* ∧ *is-arg-min length* (λ*zs*. *maxsimpchain* (*x*#*zs*@[*y*])) *xs*)
  ⟨*proof*⟩
  termination ⟨*proof*⟩

lemma *maxsimpchain-snocI*:
  ⟦ *maxsimpchain* (*xs*@[*x*]); *maxsimp y*; *x*∼*y* ⟧ ⟹ *maxsimpchain* (*xs*@[*x*,*y*])
  ⟨*proof*⟩

lemma *maxsimpchainD-maxsimp*:
  maxsimpchain xs ⟹ *x* ∈ *set xs* ⟹ *maxsimp x*
  ⟨*proof*⟩

lemma *maxsimpchainD-adj*: *maxsimpchain xs* ⟹ *adjacentchain xs*
  ⟨*proof*⟩

lemma *maxsimpchain-CConsI*:
  ⟦ *maxsimp w*; *maxsimpchain* (*x*#*xs*); *w*∼*x* ⟧ ⟹ *maxsimpchain* (*w*#*x*#*xs*)
  ⟨*proof*⟩

lemma *maxsimpchain-Cons-reduce*:
  maxsimpchain (*x*#*xs*) ⟹ *maxsimpchain xs*
  ⟨*proof*⟩

lemma *maxsimpchain-append-reduce1*:
  maxsimpchain (*xs*@*ys*) ⟹ *maxsimpchain xs*
  ⟨*proof*⟩

lemma *maxsimpchain-append-reduce2*:
  maxsimpchain (*xs*@*ys*) ⟹ *maxsimpchain ys*
  ⟨*proof*⟩

lemma *maxsimpchain-remdup-adj*:
maxsimpchain (*xs*@[*x*,*x*]@*ys*) $\Longrightarrow$ *maxsimpchain* (*xs*@[*x*]@*ys*)
⟨*proof*⟩

lemma *maxsimpchain-rev*: *maxsimpchain xs* $\Longrightarrow$ *maxsimpchain* (*rev xs*)
⟨*proof*⟩

lemma *maxsimpchain-overlap-join*:
maxsimpchain (*xs*@[*w*]) $\Longrightarrow$ *maxsimpchain* (*w*#*ys*) $\Longrightarrow$
maxsimpchain (*xs*@*w*#*ys*)
⟨*proof*⟩

lemma *pmaxsimpchain*: *pmaxsimpchain xs* $\Longrightarrow$ *maxsimpchain xs*
⟨*proof*⟩

lemma *pmaxsimpchainI-maxsimpchain*:
maxsimpchain xs $\Longrightarrow$ *distinct xs* $\Longrightarrow$ *pmaxsimpchain xs*
⟨*proof*⟩

lemma *pmaxsimpchain-CConsI*:
⟦ *maxsimp w*; *pmaxsimpchain* (*x*#*xs*); *w*∼*x*; *w* ∉ *set* (*x*#*xs*) ⟧ $\Longrightarrow$
pmaxsimpchain (*w*#*x*#*xs*)
⟨*proof*⟩

lemmas *pmaxsimpchainD-maxsimp* =
maxsimpchainD-maxsimp[*OF pmaxsimpchain*]
lemmas *pmaxsimpchainD-adj* =
maxsimpchainD-adj [*OF pmaxsimpchain*]

lemma *pmaxsimpchainD-distinct*: *pmaxsimpchain xs* $\Longrightarrow$ *distinct xs*
⟨*proof*⟩

lemma *pmaxsimpchain-Cons-reduce*:
pmaxsimpchain (*x*#*xs*) $\Longrightarrow$ *pmaxsimpchain xs*
⟨*proof*⟩

lemma *pmaxsimpchain-append-reduce1*:
pmaxsimpchain (*xs*@*ys*) $\Longrightarrow$ *pmaxsimpchain xs*
⟨*proof*⟩

lemma *maxsimpchain-obtain-pmaxsimpchain*:
assumes *x*≠*y maxsimpchain* (*x*#*xs*@[*y*])
shows ∃ *ys*. *set ys* ⊆ *set xs* ∧ *length ys* ≤ *length xs* ∧
pmaxsimpchain (*x*#*ys*@[*y*])
⟨*proof*⟩

lemma *min-maxsimpchainD-maxsimpchain*:
assumes *min-maxsimpchain xs*
shows *maxsimpchain xs*

⟨*proof*⟩

lemma *min-maxsimpchainD-min-betw*:
min-maxsimpchain $(x\#xs@[y]) \Longrightarrow$ *maxsimpchain* $(x\#ys@[y]) \Longrightarrow$
length ys $\geq$ *length xs*
⟨*proof*⟩

lemma *min-maxsimpchainI-betw*:
assumes $x \neq y$ *maxsimpchain* $(x\#xs@[y])$
$\bigwedge ys.$ *maxsimpchain* $(x\#ys@[y]) \Longrightarrow$ *length xs* $\leq$ *length ys*
shows *min-maxsimpchain* $(x\#xs@[y])$
⟨*proof*⟩

lemma *min-maxsimpchainI-betw-compare*:
assumes $x \neq y$ *maxsimpchain* $(x\#xs@[y])$
min-maxsimpchain $(x\#ys@[y])$ *length xs* = *length ys*
shows *min-maxsimpchain* $(x\#xs@[y])$
⟨*proof*⟩

lemma *min-maxsimpchain-pmaxsimpchain*:
assumes *min-maxsimpchain xs*
shows *pmaxsimpchain xs*
⟨*proof*⟩

lemma *min-maxsimpchain-rev*:
assumes *min-maxsimpchain xs*
shows *min-maxsimpchain* (*rev xs*)
⟨*proof*⟩

lemma *min-maxsimpchain-adj*:
⟦ *maxsimp x*; *maxsimp y*; $x \sim y$; $x \neq y$ ⟧ $\Longrightarrow$ *min-maxsimpchain* $[x,y]$
⟨*proof*⟩

lemma *min-maxsimpchain-betw-CCons-reduce*:
assumes *min-maxsimpchain* $(w\#x\#ys@[z])$
shows *min-maxsimpchain* $(x\#ys@[z])$
⟨*proof*⟩

lemma *min-maxsimpchain-betw-uniform-length*:
assumes *min-maxsimpchain* $(x\#xs@[y])$ *min-maxsimpchain* $(x\#ys@[y])$
shows *length xs* = *length ys*
⟨*proof*⟩

lemma *not-min-maxsimpchainI-betw*:
⟦ *maxsimpchain* $(x\#ys@[y])$; *length ys* < *length xs* ⟧ $\Longrightarrow$
$\neg$ *min-maxsimpchain* $(x\#xs@[y])$
⟨*proof*⟩

lemma *maxsimpchain-in-subcomplex*:

$\llbracket$ *Subcomplex Y*; *set ys* $\subseteq$ *Y*; *maxsimpchain ys* $\rrbracket \Longrightarrow$
SimplicialComplex.maxsimpchain Y ys
⟨*proof*⟩

end

3.4 Isomorphisms of simplicial complexes

Here we develop the concept of isomorphism of simplicial complexes. Note that we have not bothered to first develop the concept of morphism of simplicial complexes, since every function on the vertex set of a simplicial complex can be considered a morphism of complexes (see lemma *map-is-simplicial-morph* above).

locale *SimplicialComplexIsomorphism* = *SimplicialComplex X*
for *X* :: *'a set set*
+ **fixes** *f* :: *'a* ⇒ *'b*
assumes *inj*: *inj-on f* ($\bigcup$ *X*)
begin

lemmas *morph* = *map-is-simplicial-morph*[*of f*]

lemma *iso-codim-map*:
x ∈ *X* ⟹ *y* ∈ *X* ⟹ *card* (*f'x* − *f'y*) = *card* (*x*−*y*)
⟨*proof*⟩

lemma *maxsimp-im-max*: *maxsimp x* ⟹ *w* ∈ *X* ⟹ *f'x* ⊆ *f'w* ⟹ *f'w* = *f'x*
⟨*proof*⟩

lemma *maxsimp-map*:
maxsimp x ⟹ *SimplicialComplex.maxsimp* (*f*⊢*X*) (*f'x*)
⟨*proof*⟩

lemma *iso-adj-int-im*:
assumes *maxsimp x maxsimp y x*∼*y x*≠*y*
shows (*f'x* ∩ *f'y*) ⊲ *f'x*
⟨*proof*⟩

lemma *iso-adj-map*:
assumes *maxsimp x maxsimp y x*∼*y x*≠*y*
shows *f'x* ∼ *f'y*
⟨*proof*⟩

lemma *pmaxsimpchain-map*:
pmaxsimpchain xs ⟹ *SimplicialComplex.pmaxsimpchain* (*f*⊢*X*) (*f*⊨*xs*)
⟨*proof*⟩

end

3.5 The complex associated to a poset

A simplicial complex is naturally a poset under the subset relation. The following develops the reverse direction: constructing a simplicial complex from a suitable poset.

context *ordering*
begin

definition *PosetComplex* :: *'a set* ⇒ *'a set set*
  where *PosetComplex P* ≡ (⋃*x*∈*P*. { {*y*. *pseudominimal-in* (*P*.≤*x*) *y*} })

lemma *poset-is-SimplicialComplex*:
  assumes ∀ *x*∈*P*. *simplex-like* (*P*.≤*x*)
  shows *SimplicialComplex* (*PosetComplex P*)
⟨*proof*⟩

definition *poset-simplex-map* :: *'a set* ⇒ *'a* ⇒ *'a set*
  where *poset-simplex-map P x* = {*y*. *pseudominimal-in* (*P*.≤*x*) *y*}

lemma *poset-to-PosetComplex-OrderingSetMap*:
  assumes ⋀*x*. *x*∈*P* ⟹ *simplex-like* (*P*.≤*x*)
  shows *OrderingSetMap* (≤) (<) (⊆) (⊂) *P* (*poset-simplex-map P*)
⟨*proof*⟩

end

When a poset affords a simplicial complex, there is a natural morphism of posets from the source poset into the poset of sets in the complex, as above. However, some further assumptions are necessary to ensure that this morphism is an isomorphism. These conditions are collected in the following locale.

locale *ComplexLikePoset* = *ordering less-eq less*
  for *less-eq* :: *'a*⇒*'a*⇒*bool* (**infix** ‹≤› *50*)
  and *less* :: *'a*⇒*'a*⇒*bool* (**infix** ‹<› *50*)
\+ **fixes** *P* :: *'a set*
  assumes *below-in-P-simplex-like*: *x*∈*P* ⟹ *simplex-like* (*P*.≤*x*)
  and *P-has-bottom* : *has-bottom P*
  and *P-has-glbs* : *x*∈*P* ⟹ *y*∈*P* ⟹ ∃ *b*. *glbound-in-of P x y b*
begin

abbreviation *smap* ≡ *poset-simplex-map P*

lemma *smap-onto-PosetComplex*: *smap* ‘ *P* = *PosetComplex P*
  ⟨*proof*⟩

lemma *ordsetmap-smap*: ⟦ *a*∈*P*; *b*∈*P*; *a*≤*b* ⟧ ⟹ *smap a* ⊆ *smap b*
  ⟨*proof*⟩

lemma *inj-on-smap*: *inj-on smap P*
⟨*proof*⟩

lemma *OrderingSetIso-smap*:
OrderingSetIso (≤) (<) (⊆) (⊂) *P smap*
⟨*proof*⟩

lemmas *rev-ordsetmap-smap* =
OrderingSetIso.rev-ordsetmap[*OF OrderingSetIso-smap*]

end

end

4 Chamber complexes

Now we develop the basic theory of chamber complexes, including both thin and thick complexes. Some terminology: a maximal simplex is now called a chamber, and a chain (with respect to adjacency) of chambers is now called a gallery. A gallery in which no chamber appears more than once is called proper, and we use the prefix p as a naming convention to denote proper. Again, we remind the reader that some sources reserve the name gallery for (a slightly weaker notion of) what we are calling a proper gallery, using pregallery to denote an improper gallery.

theory *Chamber*
imports *Algebra Simplicial*

begin

4.1 Locale definition and basic facts

locale *ChamberComplex* = *SimplicialComplex X*
for *X* :: *'a set set*
\+ **assumes** *simplex-in-max* : $y{\in}X \Longrightarrow \exists\, x.\ maxsimp\ x \wedge y{\subseteq}x$
and *maxsimp-connect*: $[\![\ x \neq y;\ maxsimp\ x;\ maxsimp\ y\ ]\!] \Longrightarrow$
$\exists\, xs.\ maxsimpchain\ (x\#xs@[y])$

context *ChamberComplex*
begin

abbreviation *chamber* ≡ *maxsimp*
abbreviation *gallery* ≡ *maxsimpchain*
abbreviation *pgallery* ≡ *pmaxsimpchain*
abbreviation *min-gallery* ≡ *min-maxsimpchain*
abbreviation *supchamber v* ≡ (*SOME C. chamber C* ∧ *v*∈*C*)

lemmas *faces* = *faces*

lemmas *singleton-simplex* = *singleton-simplex*
lemmas *chamberI* = *maxsimpI*
lemmas *chamberD-simplex* = *maxsimpD-simplex*
lemmas *chamberD-maximal* = *maxsimpD-maximal*
lemmas *finite-chamber* = *finite-maxsimp*
lemmas *chamber-nempty* = *maxsimp-nempty*
lemmas *chamber-vertices* = *maxsimp-vertices*
lemmas *gallery-def* = *maxsimpchain-def*
lemmas *gallery-snocI* = *maxsimpchain-snocI*
lemmas *galleryD-chamber* = *maxsimpchainD-maxsimp*
lemmas *galleryD-adj* = *maxsimpchainD-adj*
lemmas *gallery-CConsI* = *maxsimpchain-CConsI*
lemmas *gallery-Cons-reduce* = *maxsimpchain-Cons-reduce*
lemmas *gallery-append-reduce1* = *maxsimpchain-append-reduce1*
lemmas *gallery-append-reduce2* = *maxsimpchain-append-reduce2*
lemmas *gallery-remdup-adj* = *maxsimpchain-remdup-adj*
lemmas *gallery-obtain-pgallery* = *maxsimpchain-obtain-pmaxsimpchain*
lemmas *pgallery-def* = *pmaxsimpchain-def*
lemmas *pgalleryI-gallery* = *pmaxsimpchainI-maxsimpchain*
lemmas *pgalleryD-chamber* = *pmaxsimpchainD-maxsimp*
lemmas *pgalleryD-adj* = *pmaxsimpchainD-adj*
lemmas *pgalleryD-distinct* = *pmaxsimpchainD-distinct*
lemmas *pgallery-Cons-reduce* = *pmaxsimpchain-Cons-reduce*
lemmas *pgallery-append-reduce1* = *pmaxsimpchain-append-reduce1*
lemmas *pgallery* = *pmaxsimpchain*
lemmas *min-gallery-simps* = *min-maxsimpchain.simps*
lemmas *min-galleryI-betw* = *min-maxsimpchainI-betw*
lemmas *min-galleryI-betw-compare* = *min-maxsimpchainI-betw-compare*
lemmas *min-galleryD-min-betw* = *min-maxsimpchainD-min-betw*
lemmas *min-galleryD-gallery* = *min-maxsimpchainD-maxsimpchain*
lemmas *min-gallery-pgallery* = *min-maxsimpchain-pmaxsimpchain*
lemmas *min-gallery-rev* = *min-maxsimpchain-rev*
lemmas *min-gallery-adj* = *min-maxsimpchain-adj*
lemmas *not-min-galleryI-betw* = *not-min-maxsimpchainI-betw*

lemmas *min-gallery-betw-CCons-reduce* =
min-maxsimpchain-betw-CCons-reduce
lemmas *min-gallery-betw-uniform-length* =
min-maxsimpchain-betw-uniform-length
lemmas *vertex-set-int* = *vertex-set-int*[*OF ChamberComplex.axioms*(*1*)]

lemma *chamber-pconnect*:
$\llbracket\ x \neq y;\ chamber\ x;\ chamber\ y\ \rrbracket \Longrightarrow \exists\, xs.\ pgallery\ (x\#xs@[y])$
⟨*proof*⟩

lemma *supchamberD*:
assumes $v \in \bigcup X$
defines $C \equiv supchamber\ v$
shows *chamber C* $v \in C$

⟨*proof*⟩

definition
ChamberSubcomplex Y ≡ *Y* ⊆ *X* ∧ *ChamberComplex Y* ∧
(∀ *C*. *ChamberComplex.chamber Y C* ⟶ *chamber C*)

lemma *ChamberSubcomplexI*:
assumes *Y*⊆*X* *ChamberComplex Y*
⋀*y*. *ChamberComplex.chamber Y y* ⟹ *chamber y*
shows *ChamberSubcomplex Y*
⟨*proof*⟩

lemma *ChamberSubcomplexD-sub*: *ChamberSubcomplex Y* ⟹ *Y* ⊆ *X*
⟨*proof*⟩

lemma *ChamberSubcomplexD-complex*:
ChamberSubcomplex Y ⟹ *ChamberComplex Y*
⟨*proof*⟩

lemma *chambersub-imp-sub*: *ChamberSubcomplex Y* ⟹ *Subcomplex Y*
⟨*proof*⟩

lemma *chamber-in-subcomplex*:
⟦ *ChamberSubcomplex Y*; *C* ∈ *Y*; *chamber C* ⟧ ⟹
ChamberComplex.chamber Y C
⟨*proof*⟩

lemma *subcomplex-chamber*:
ChamberSubcomplex Y ⟹ *ChamberComplex.chamber Y C* ⟹ *chamber C*
⟨*proof*⟩

lemma *gallery-in-subcomplex*:
⟦ *ChamberSubcomplex Y*; *set ys* ⊆ *Y*; *gallery ys* ⟧ ⟹
ChamberComplex.gallery Y ys
⟨*proof*⟩

lemma *subcomplex-gallery*:
ChamberSubcomplex Y ⟹ *ChamberComplex.gallery Y Cs* ⟹ *gallery Cs*
⟨*proof*⟩

lemma *subcomplex-pgallery*:
ChamberSubcomplex Y ⟹ *ChamberComplex.pgallery Y Cs* ⟹ *pgallery Cs*
⟨*proof*⟩

lemma *min-gallery-in-subcomplex*:
assumes *ChamberSubcomplex Y min-gallery Cs set Cs* ⊆ *Y*
shows *ChamberComplex.min-gallery Y Cs*
⟨*proof*⟩

lemma *chamber-card*: *chamber C* $\Longrightarrow$ *chamber D* $\Longrightarrow$ *card C* = *card D*
⟨*proof*⟩

lemma *chamber-facet-is-chamber-facet*:
⟦ *chamber C*; *chamber D*; *z*⊲*C*; *z*⊆*D* ⟧ $\Longrightarrow$ *z*⊲*D*
⟨*proof*⟩

lemma *chamber-adj*:
assumes *chamber C D*∈*X C* ∼ *D*
shows *chamber D*
⟨*proof*⟩

lemma *chambers-share-facet*:
assumes *chamber C chamber* (*insert v z*) *z*⊲*C*
shows *z*⊲*insert v z*
⟨*proof*⟩

lemma *adjacentset-chamber*: *chamber C* $\Longrightarrow$ *D*∈*adjacentset C* $\Longrightarrow$ *chamber D*
⟨*proof*⟩

lemma *chamber-shared-facet*: ⟦ *chamber C*; *z*⊲*C*; *D*∈*X*; *z*⊲*D* ⟧ $\Longrightarrow$ *chamber D*
⟨*proof*⟩

lemma *adjacentset-conv-facetchambersets*:
assumes *X* ≠ {{}} *chamber C*
shows *adjacentset C* = (⋃ *v*∈*C*. {*D*∈*X*. *C*−{*v*}⊲*D*})
⟨*proof*⟩

end

4.2 The system of chambers and distance between chambers

We now examine the system of all chambers in more detail, and explore the distance function on this system provided by lengths of minimal galleries.

context *ChamberComplex*
begin

definition *chamber-system* :: *'a set set*
where *chamber-system* ≡ {*C*. *chamber C*}
abbreviation $\mathcal{C}$ ≡ *chamber-system*

definition *chamber-distance* :: *'a set* ⇒ *'a set* ⇒ *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* ⇒ *'a set* ⇒ *'a set*
where *closest-supchamber F D* =
(*ARG-MIN* (λ*C*. *chamber-distance C D*) *C*.

chamber C ∧ F⊆C)

definition *face-distance F D* ≡ *chamber-distance* (*closest-supchamber F D*) *D*

lemma *chamber-system-simplices*: $\mathcal{C} \subseteq X$
⟨*proof*⟩

lemma *gallery-chamber-system*: *gallery Cs* ⟹ *set Cs* ⊆ $\mathcal{C}$
⟨*proof*⟩

lemmas *pgallery-chamber-system* = *gallery-chamber-system*[*OF pgallery*]

lemma *chamber-distance-le*:
gallery (*C#Cs*@[*D*]) ⟹ *chamber-distance C D* ≤ *Suc* (*length Cs*)
⟨*proof*⟩

lemma *min-gallery-betw-chamber-distance*:
min-gallery (*C#Cs*@[*D*]) ⟹ *chamber-distance C D* = *Suc* (*length Cs*)
⟨*proof*⟩

lemma *min-galleryI-chamber-distance-betw*:
gallery (*C#Cs*@[*D*]) ⟹ *Suc* (*length Cs*) = *chamber-distance C D* ⟹
min-gallery (*C#Cs*@[*D*])
⟨*proof*⟩

lemma *gallery-least-length*:
assumes *chamber C chamber D C≠D*
defines *Cs* ≡ *ARG-MIN length Cs. gallery* (*C#Cs*@[*D*])
shows *gallery* (*C#Cs*@[*D*])
⟨*proof*⟩

lemma *min-gallery-least-length*:
assumes *chamber C chamber D C≠D*
defines *Cs* ≡ *ARG-MIN length Cs. gallery* (*C#Cs*@[*D*])
shows *min-gallery* (*C#Cs*@[*D*])
⟨*proof*⟩

lemma *pgallery-least-length*:
assumes *chamber C chamber D C≠D*
defines *Cs* ≡ *ARG-MIN length Cs. gallery* (*C#Cs*@[*D*])
shows *pgallery* (*C#Cs*@[*D*])
⟨*proof*⟩

lemma *closest-supchamberD*:
assumes *F∈X chamber D*
shows *chamber* (*closest-supchamber F D*) *F* ⊆ *closest-supchamber F D*
⟨*proof*⟩

lemma *closest-supchamber-closest*:

chamber C $\Longrightarrow$ *F*$\subseteq$*C* $\Longrightarrow$
chamber-distance (*closest-supchamber F D*) *D* $\leq$ *chamber-distance C D*
⟨*proof*⟩

lemma *face-distance-le*:
chamber C $\Longrightarrow$ *F*$\subseteq$*C* $\Longrightarrow$ *face-distance F D* $\leq$ *chamber-distance C D*
⟨*proof*⟩

lemma *face-distance-eq-0*: *chamber C* $\Longrightarrow$ *F*$\subseteq$*C* $\Longrightarrow$ *face-distance F C* = *0*
⟨*proof*⟩

end

4.3 Labelling a chamber complex

A labelling of a chamber complex is a function on the vertex set so that each chamber is in bijective correspondence with the label set (chambers all have the same number of vertices).

context *ChamberComplex*
begin

definition *label-wrt* :: *'b set* $\Rightarrow$ (*'a*$\Rightarrow$*'b*) $\Rightarrow$ *bool*
where *label-wrt B f* $\equiv$ ($\forall$ *C*$\in\mathcal{C}$. *bij-betw f C B*)

lemma *label-wrtD*: *label-wrt B f* $\Longrightarrow$ *C*$\in\mathcal{C}$ $\Longrightarrow$ *bij-betw f C B*
⟨*proof*⟩

lemma *label-wrtD'*: *label-wrt B f* $\Longrightarrow$ *chamber C* $\Longrightarrow$ *bij-betw f C B*
⟨*proof*⟩

lemma *label-wrt-adjacent*:
assumes *label-wrt B f chamber C chamber D C*$\sim$*D v*$\in$*C*$-$*D w*$\in$*D*$-$*C*
shows *f v* = *f w*
⟨*proof*⟩

lemma *label-wrt-adjacent-shared-facet*:
⟦ *label-wrt B f*; *chamber* (*insert v z*); *chamber* (*insert w z*); *v*$\notin$*z*; *w*$\notin$*z* ⟧ $\Longrightarrow$
f v = *f w*
⟨*proof*⟩

lemma *label-wrt-elt-image*: *label-wrt B f* $\Longrightarrow$ *v*$\in\bigcup$*X* $\Longrightarrow$ *f v* $\in$ *B*
⟨*proof*⟩

end

4.4 Morphisms of chamber complexes

While any function on the vertex set of a simplicial complex can be considered a morphism of simplicial complexes onto its image, for chamber complexes we require the function send chambers onto chambers of the same cardinality in some chamber complex of the codomain.

4.4.1 Morphism locale and basic facts

locale *ChamberComplexMorphism* = *domain*: *ChamberComplex X* + *codomain*: *ChamberComplex Y*
  for *X* :: *'a set set*
  and *Y* :: *'b set set*
+ **fixes** *f* :: *'a⇒'b*
  assumes *chamber-map*: *domain.chamber C* ⟹ *codomain.chamber* (*f'C*)
  and *dim-map* : *domain.chamber C* ⟹ *card* (*f'C*) = *card C*

lemma (**in** *ChamberComplex*) *trivial-morphism*:
  ChamberComplexMorphism X X id
  ⟨*proof*⟩

lemma (**in** *ChamberComplex*) *inclusion-morphism*:
  assumes *ChamberSubcomplex Y*
  shows *ChamberComplexMorphism Y X id*
  ⟨*proof*⟩

context *ChamberComplexMorphism*
begin

lemmas *domain-complex* = *domain.ChamberComplex-axioms*
lemmas *codomain-complex* = *codomain.ChamberComplex-axioms*

lemmas *simplicialcomplex-image* = *domain.map-is-simplicial-morph*[*of f*]

lemma *cong*: *fun-eq-on g f* (⋃*X*) ⟹ *ChamberComplexMorphism X Y g*
  ⟨*proof*⟩

lemma *comp*:
  assumes *ChamberComplexMorphism Y Z g*
  shows *ChamberComplexMorphism X Z* (*g*∘*f*)
⟨*proof*⟩

lemma *restrict-domain*:
  assumes *domain.ChamberSubcomplex W*
  shows *ChamberComplexMorphism W Y f*
⟨*proof*⟩

lemma *restrict-codomain*:
  assumes *codomain.ChamberSubcomplex Z f⊢X ⊆ Z*

shows *ChamberComplexMorphism X Z f*
⟨*proof*⟩

lemma *inj-on-chamber*: *domain.chamber C* $\Longrightarrow$ *inj-on f C*
⟨*proof*⟩

lemma *bij-betw-chambers*: *domain.chamber C* $\Longrightarrow$ *bij-betw f C* $(f`C)$
⟨*proof*⟩

lemma *card-map*: $x\in X \Longrightarrow$ *card* $(f`x) =$ *card* x
⟨*proof*⟩

lemma *codim-map*:
assumes *domain.chamber C* $y \subseteq C$
shows *card* $(f`C - f`y) =$ *card* $(C-y)$
⟨*proof*⟩

lemma *simplex-map*: $x\in X \Longrightarrow f`x\in Y$
⟨*proof*⟩

lemma *simplices-map*: $f\vdash X \subseteq Y$
⟨*proof*⟩

lemma *vertex-map*: $x \in \bigcup X \Longrightarrow f\ x \in \bigcup Y$
⟨*proof*⟩

lemma *facet-map*: *domain.chamber C* $\Longrightarrow z\lhd C \Longrightarrow f`z \lhd f`C$
⟨*proof*⟩

lemma *adj-int-im*:
assumes *domain.chamber C domain.chamber D* $C \sim D$ $f`C \neq f`D$
shows $(f`C \cap f`D) \lhd f`C$
⟨*proof*⟩

lemma *adj-map'*:
assumes *domain.chamber C domain.chamber D* $C \sim D$ $f`C \neq f`D$
shows $f`C \sim f`D$
⟨*proof*⟩

lemma *adj-map*:
⟦ *domain.chamber C*; *domain.chamber D*; $C \sim D$ ⟧ $\Longrightarrow f`C \sim f`D$
⟨*proof*⟩

lemma *chamber-vertex-outside-facet-image*:
assumes $v\notin z$ *domain.chamber* (*insert v z*)
shows $f\ v \notin f`z$
⟨*proof*⟩

lemma *expand-codomain*:

assumes *ChamberComplex Z ChamberComplex.ChamberSubcomplex Z Y*
shows *ChamberComplexMorphism X Z f*
⟨*proof*⟩

end

4.4.2 Action on pregalleries and galleries

context *ChamberComplexMorphism*
begin

lemma *gallery-map*: *domain.gallery Cs* ⟹ *codomain.gallery* (*f*⊨*Cs*)
⟨*proof*⟩

lemma *gallery-betw-map*:
domain.gallery (*C*#*Cs*@[*D*]) ⟹ *codomain.gallery* (*f*'*C* # *f*⊨*Cs* @ [*f*'*D*])
⟨*proof*⟩

end

4.4.3 Properties of the image

context *ChamberComplexMorphism*
begin

lemma *subcomplex-image*: *codomain.Subcomplex* (*f*⊢*X*)
⟨*proof*⟩

lemmas *chamber-in-image* = *codomain.max-in-subcomplex*[*OF subcomplex-image*]

lemma *maxsimp-map-into-image*:
assumes *domain.chamber x*
shows *SimplicialComplex.maxsimp* (*f*⊢*X*) (*f*'*x*)
⟨*proof*⟩

lemma *maxsimp-preimage*:
assumes *C*∈*X SimplicialComplex.maxsimp* (*f*⊢*X*) (*f*'*C*)
shows *domain.chamber C*
⟨*proof*⟩

lemma *chamber-preimage*:
C∈*X* ⟹ *codomain.chamber* (*f*'*C*) ⟹ *domain.chamber C*
⟨*proof*⟩

lemma *chambercomplex-image*: *ChamberComplex* (*f*⊢*X*)
⟨*proof*⟩

lemma *chambersubcomplex-image*: *codomain.ChamberSubcomplex* (*f*⊢*X*)
⟨*proof*⟩

lemma *restrict-codomain-to-image*: *ChamberComplexMorphism X* ($f \vdash X$) *f*
⟨*proof*⟩

end

4.4.4 Action on the chamber system

context *ChamberComplexMorphism*
begin

lemma *chamber-system-into*: $f \vdash domain.\mathcal{C} \subseteq codomain.\mathcal{C}$
⟨*proof*⟩

lemma *chamber-system-image*: $f \vdash domain.\mathcal{C} = codomain.\mathcal{C} \cap (f \vdash X)$
⟨*proof*⟩

lemma *image-chamber-system*: *ChamberComplex*.$\mathcal{C}$ ($f \vdash X$) = $f \vdash domain.\mathcal{C}$
⟨*proof*⟩

lemma *image-chamber-system-image*:
ChamberComplex.$\mathcal{C}$ ($f \vdash X$) = *codomain*.$\mathcal{C}$ ∩ ($f \vdash X$)
⟨*proof*⟩

lemma *face-distance-eq-chamber-distance-map*:
assumes *domain.chamber C domain.chamber D* $C \neq D$ $z \subseteq C$
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*)
⟨*proof*⟩

lemma *face-distance-eq-chamber-distance-min-gallery-betw-map*:
assumes *domain.chamber C domain.chamber D* $C \neq D$ $z \subseteq C$
codomain.face-distance (*f'z*) (*f'D*) = *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*]))
⟨*proof*⟩

end

4.4.5 Isomorphisms

locale *ChamberComplexIsomorphism* = *ChamberComplexMorphism X Y f*
for *X* :: *'a set set*
and *Y* :: *'b set set*
and *f* :: *'a⇒'b*
\+ **assumes** *bij-betw-vertices*: *bij-betw f* ($\bigcup X$) ($\bigcup Y$)
and *surj-simplex-map* : $f \vdash X = Y$

lemma (**in** *ChamberComplexIsomorphism*) *inj*: *inj-on f* ($\bigcup X$)
⟨*proof*⟩

sublocale *ChamberComplexIsomorphism* < *SimplicialComplexIsomorphism*
⟨*proof*⟩

lemma (**in** *ChamberComplex*) *trivial-isomorphism*:
ChamberComplexIsomorphism X X id
⟨*proof*⟩

lemma (**in** *ChamberComplexMorphism*) *isoI-inverse*:
assumes *ChamberComplexMorphism Y X g*
fixespointwise ($g \circ f$) ($\bigcup X$) *fixespointwise* ($f \circ g$) ($\bigcup Y$)
shows *ChamberComplexIsomorphism X Y f*
⟨*proof*⟩

context *ChamberComplexIsomorphism*
begin

lemmas *domain-complex* = *domain-complex*
lemmas *chamber-map* = *chamber-map*
lemmas *dim-map* = *dim-map*
lemmas *gallery-map* = *gallery-map*
lemmas *simplex-map* = *simplex-map*
lemmas *chamber-preimage* = *chamber-preimage*

lemma *chamber-morphism*: *ChamberComplexMorphism X Y f* ⟨*proof*⟩

lemma *pgallery-map*: *domain.pgallery Cs* ⟹ *codomain.pgallery* ($f \models Cs$)
⟨*proof*⟩

lemma *iso-cong*:
assumes *fun-eq-on g f* ($\bigcup X$)
shows *ChamberComplexIsomorphism X Y g*
⟨*proof*⟩

lemma *iso-comp*:
assumes *ChamberComplexIsomorphism Y Z g*
shows *ChamberComplexIsomorphism X Z* ($g \circ f$)
⟨*proof*⟩

lemma *inj-on-chamber-system*: *inj-on* ((') *f*) *domain.*$\mathcal{C}$
⟨*proof*⟩

lemma *inv*: *ChamberComplexIsomorphism Y X* (*the-inv-into* ($\bigcup X$) *f*)
⟨*proof*⟩

lemma *chamber-distance-map*:
assumes *domain.chamber C domain.chamber D*

shows *codomain.chamber-distance* (f‘C) (f‘D) =
domain.chamber-distance C D
⟨*proof*⟩

lemma *face-distance-map*:
assumes *domain.chamber C F*∈*X*
shows *codomain.face-distance* (f‘F) (f‘C) = *domain.face-distance F C*
⟨*proof*⟩

end

4.4.6 Endomorphisms

locale *ChamberComplexEndomorphism* = *ChamberComplexMorphism X X f*
for *X* :: *'a set set*
and *f* :: *'a*⇒*'a*
+ **assumes** *trivial-outside* : $v \notin \bigcup X \Longrightarrow f\ v = v$
— to facilitate uniqueness arguments

lemma (**in** *ChamberComplex*) *trivial-endomorphism*:
ChamberComplexEndomorphism X id
⟨*proof*⟩

context *ChamberComplexEndomorphism*
begin

abbreviation *ChamberSubcomplex* ≡ *domain.ChamberSubcomplex*
abbreviation *Subcomplex* ≡ *domain.Subcomplex*
abbreviation *chamber* ≡ *domain.chamber*
abbreviation *gallery* ≡ *domain.gallery*
abbreviation $\mathcal{C}$ ≡ *domain.chamber-system*
abbreviation *label-wrt* ≡ *domain.label-wrt*

lemmas *dim-map* = *dim-map*
lemmas *simplex-map* = *simplex-map*
lemmas *vertex-map* = *vertex-map*
lemmas *chamber-map* = *chamber-map*
lemmas *adj-map* = *adj-map*
lemmas *facet-map* = *facet-map*
lemmas *bij-betw-chambers* = *bij-betw-chambers*
lemmas *chamber-system-into* = *chamber-system-into*
lemmas *chamber-system-image* = *chamber-system-image*
lemmas *image-chamber-system* = *image-chamber-system*
lemmas *chambercomplex-image* = *chambercomplex-image*
lemmas *chambersubcomplex-image* = *chambersubcomplex-image*

lemmas *face-distance-eq-chamber-distance-map* =
face-distance-eq-chamber-distance-map
lemmas *face-distance-eq-chamber-distance-min-gallery-betw-map* =

face-distance-eq-chamber-distance-min-gallery-betw-map
lemmas *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*∘*f*)
⟨*proof*⟩

lemma *restrict-endo*:
assumes *ChamberSubcomplex Y f*⊢*Y* ⊆ *Y*
shows *ChamberComplexEndomorphism Y* (*restrict1 f* (⋃ *Y*))
⟨*proof*⟩

lemma *funpower-endomorphism*:
ChamberComplexEndomorphism X (*f*^^*n*)
⟨*proof*⟩

end

lemma (**in** *ChamberComplex*) *fold-chamber-complex-endomorph-list*:
∀ *x*∈*set xs*. *ChamberComplexEndomorphism X* (*f x*) ⟹
ChamberComplexEndomorphism X (*fold f xs*)
⟨*proof*⟩

context *ChamberComplexEndomorphism*
begin

lemma *split-gallery*:
⟦ *C*∈*f*⊢𝒞; *D*∈𝒞−*f*⊢𝒞; *gallery* (*C*#*Cs*@[*D*]) ⟧ ⟹
∃ *As A B Bs*. *A*∈*f*⊢𝒞 ∧ *B*∈𝒞−*f*⊢𝒞 ∧ *C*#*Cs*@[*D*] = *As*@*A*#*B*#*Bs*
⟨*proof*⟩

lemma *respects-labels-adjacent*:
  assumes *label-wrt B* φ *chamber C chamber D* $C \sim D$ $\forall v \in C.\ \varphi\ (f\ v) = \varphi\ v$
  shows $\forall v \in D.\ \varphi\ (f\ v) = \varphi\ v$
⟨*proof*⟩

lemma *respects-labels-gallery*:
  assumes *label-wrt B* φ $\forall v \in C.\ \varphi\ (f\ v) = \varphi\ v$
  shows *gallery* $(C\#Cs@[D]) \Longrightarrow \forall v \in D.\ \varphi\ (f\ v) = \varphi\ v$
⟨*proof*⟩

lemma *respect-label-fix-chamber-imp-fun-eq-on*:
  assumes *label* : *label-wrt B* φ
  and *chamber*: *chamber C f'C = g'C*
  and *respect*: $\forall v \in C.\ \varphi\ (f\ v) = \varphi\ v$ $\forall v \in C.\ \varphi\ (g\ v) = \varphi\ v$
  shows *fun-eq-on f g C*
⟨*proof*⟩

lemmas *respects-label-fixes-chamber-imp-fixespointwise* =
  respect-label-fix-chamber-imp-fun-eq-on[*of - - - id*, *simplified*]

end

4.4.7 Automorphisms

locale *ChamberComplexAutomorphism* = *ChamberComplexIsomorphism X X f*
  for *X* :: *'a set set*
  and *f* :: $'a \Rightarrow 'a$
+ **assumes** *trivial-outside* : $v \notin \bigcup X \Longrightarrow f\ v = v$
  — to facilitate uniqueness arguments

sublocale *ChamberComplexAutomorphism* < *ChamberComplexEndomorphism*
  ⟨*proof*⟩

lemma (**in** *ChamberComplex*) *trivial-automorphism*:
  ChamberComplexAutomorphism X id
  ⟨*proof*⟩

context *ChamberComplexAutomorphism*
begin

lemmas *facet-map* = *facet-map*
lemmas *chamber-map* = *chamber-map*
lemmas *chamber-morphism* = *chamber-morphism*
lemmas *bij-betw-vertices* = *bij-betw-vertices*
lemmas *surj-simplex-map* = *surj-simplex-map*

lemma *bij*: *bij f*
⟨*proof*⟩

lemma *comp*:
assumes *ChamberComplexAutomorphism X g*
shows *ChamberComplexAutomorphism X* (*g*∘*f*)
⟨*proof*⟩

lemma *equality*:
assumes *ChamberComplexAutomorphism X g fun-eq-on f g* ($\bigcup$ *X*)
shows *f* = *g*
⟨*proof*⟩

end

4.4.8 Retractions

A retraction of a chamber complex is an endomorphism that is the identity on its image.

locale *ChamberComplexRetraction* = *ChamberComplexEndomorphism X f*
for *X* :: *'a set set*
and *f* :: *'a*⇒*'a*
\+ **assumes** *retraction*: *v*∈$\bigcup$ *X* ⟹ *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*∈*f*‘($\bigcup$ *X*) ⟹ *f v* = *v*
⟨*proof*⟩

lemma *simplex-retraction1*: *x*∈*f*⊢*X* ⟹ *fixespointwise f x*
⟨*proof*⟩

lemma *simplex-retraction2*: *x*∈*f*⊢*X* ⟹ *f*‘*x* = *x*
⟨*proof*⟩

lemma *chamber-retraction1*: *C*∈*f*⊢$\mathcal{C}$ ⟹ *fixespointwise f C*
⟨*proof*⟩

lemma *chamber-retraction2*: *C*∈*f*⊢$\mathcal{C}$ ⟹ *f*‘*C* = *C*
⟨*proof*⟩

lemma *respects-labels*:
assumes *label-wrt B* φ *v*∈($\bigcup$ *X*)
shows φ (*f v*) = φ *v*
⟨*proof*⟩

end

4.4.9 Foldings of chamber complexes

A folding of a chamber complex is a retraction that literally folds the complex in half, in that each chamber in the image is the image of precisely two chambers: itself (since a folding is a retraction) and a unique chamber outside the image.

Locale definition Here we define the locale and collect some lemmas inherited from the *ChamberComplexRetraction* locale.

locale *ChamberComplexFolding* = *ChamberComplexRetraction X f*
 for *X* :: *'a set set*
 and *f* :: *'a⇒'a*
\+ **assumes** *folding*:
  chamber C ⟹ *C*∈*f*⊢*X* ⟹
   ∃!*D*. *chamber D* ∧ *D*∉*f*⊢*X* ∧ *f*'*D* = *C*
begin

lemmas *folding-ex* = *ex1-implies-ex*[*OF folding*]
lemmas *chamber-system-into* = *chamber-system-into*
lemmas *gallery-map* = *gallery-map*
lemmas *chamber-retraction1* = *chamber-retraction1*
lemmas *chamber-retraction2* = *chamber-retraction2*

end

Decomposition into half chamber systems and half apartments Here we describe how a folding splits the chamber system of the complex into its image and the complement of its image. The chamber subcomplex consisting of all simplices contained in a chamber of a given half of the chamber system is called a half-apartment.

context *ChamberComplexFolding*
begin

definition *opp-half-apartment* :: *'a set set*
 where *opp-half-apartment* ≡ {*x*∈*X*. ∃ *C*∈𝒞−*f*⊢𝒞. *x*⊆*C*}
abbreviation *Y* ≡ *opp-half-apartment*

lemma *opp-half-apartment-subset-complex*: *Y*⊆*X*
 ⟨*proof*⟩

lemma *simplicialcomplex-opp-half-apartment*: *SimplicialComplex Y*
⟨*proof*⟩

lemma *subcomplex-opp-half-apartment*: *Subcomplex Y*
 ⟨*proof*⟩

lemma *opp-half-apartmentI*: $[\![x{\in}X;\ C{\in}\mathcal{C}{-}f{\vdash}\mathcal{C};\ x{\subseteq}C]\!] \Longrightarrow x{\in}Y$
⟨*proof*⟩

lemma *opp-chambers-subset-opp-half-apartment*: $\mathcal{C}{-}f{\vdash}\mathcal{C} \subseteq Y$
⟨*proof*⟩

lemma *maxsimp-in-opp-half-apartment*:
assumes *SimplicialComplex.maxsimp Y C*
shows $C \in \mathcal{C}{-}f{\vdash}\mathcal{C}$
⟨*proof*⟩

lemma *chamber-in-opp-half-apartment*:
SimplicialComplex.maxsimp Y C $\Longrightarrow$ *chamber C*
⟨*proof*⟩

end

Mapping between half chamber systems for foldings Since each chamber in the image of the folding is the image of a unique chamber in the complement of the image, we obtain well-defined functions from one half chamber system to the other.

context *ChamberComplexFolding*
begin

abbreviation *opp-chamber C* $\equiv$ *THE D.* $D{\in}\mathcal{C}{-}f{\vdash}\mathcal{C} \wedge f`D = C$
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* $D{\notin}f{\vdash}X$ $f`C = f`D$
shows $C{=}D$
⟨*proof*⟩

lemma *inj-on-opp-chambers″*:
$[\![C \in \mathcal{C}{-}f{\vdash}\mathcal{C};\ D \in \mathcal{C}{-}f{\vdash}\mathcal{C};\ f`C = f`D]\!] \Longrightarrow C{=}D$
⟨*proof*⟩

lemma *inj-on-opp-chambers*: *inj-on* $((`)\ f)\ (\mathcal{C}{-}f{\vdash}\mathcal{C})$
⟨*proof*⟩

lemma *opp-chambers-surj*: $f{\vdash}(\mathcal{C}{-}(f{\vdash}\mathcal{C})) = f{\vdash}\mathcal{C}$
⟨*proof*⟩

lemma *opp-chambers-bij*: *bij-betw* $((`)\ f)\ (\mathcal{C}{-}(f{\vdash}\mathcal{C}))\ (f{\vdash}\mathcal{C})$
⟨*proof*⟩

lemma *folding′*:
assumes $C{\in}f{\vdash}\mathcal{C}$
shows $\exists!D{\in}\mathcal{C}{-}f{\vdash}\mathcal{C}.\ f`D = C$

⟨*proof*⟩

lemma *opp-chambers-distinct-map*:
set Cs ⊆ $\mathcal{C}$−*f*⊢$\mathcal{C}$ ⟹ *distinct Cs* ⟹ *distinct* (*f*⊨*Cs*)
⟨*proof*⟩

lemma *opp-chamberD1*: *C*∈*f*⊢$\mathcal{C}$ ⟹ *opp-chamber C* ∈ $\mathcal{C}$−*f*⊢$\mathcal{C}$
⟨*proof*⟩

lemma *opp-chamberD2*: *C*∈*f*⊢$\mathcal{C}$ ⟹ *f*‘(*opp-chamber C*) = *C*
⟨*proof*⟩

lemma *opp-chamber-reverse*: *C*∈$\mathcal{C}$−*f*⊢$\mathcal{C}$ ⟹ *opp-chamber* (*f*‘*C*) = *C*
⟨*proof*⟩

lemma *f-opp-chamber-list*:
set Cs ⊆ *f*⊢$\mathcal{C}$ ⟹ *f*⊨(*map opp-chamber Cs*) = *Cs*
⟨*proof*⟩

lemma *flop-chamber*: *chamber C* ⟹ *chamber* (*flop C*)
⟨*proof*⟩

end

4.5 Thin chamber complexes

A thin chamber complex is one in which every facet is a facet in exactly two chambers. Slightly more generally, we first consider the case of a chamber complex in which every facet is a facet of at most two chambers. One of the main results obtained at this point is the so-called standard uniqueness argument, which essentially states that two morphisms on a thin chamber complex that agree on a particular chamber must in fact agree on the entire complex. Following that, foldings of thin chamber complexes are investigated. In particular, we are interested in pairs of opposed foldings.

4.5.1 Locales and basic facts

locale *ThinishChamberComplex* = *ChamberComplex X*
for *X* :: ′*a set set*
\+ **assumes** *thinish*:
⟦ *chamber C*; *z*⊲*C*; ∃ *D*∈*X*−{*C*}. *z*⊲*D* ⟧ ⟹ ∃!*D*∈*X*−{*C*}. *z*⊲*D*
— being adjacent to a chamber, such a *D* would also be a chamber (see lemma *chamber-adj*)
begin

lemma *facet-unique-other-chamber*:
⟦ *chamber B*; *z*⊲*B*; *chamber C*; *z*⊲*C*; *chamber D*; *z*⊲*D*; *C*≠*B*; *D*≠*B* ⟧
⟹ *C*=*D*

⟨*proof*⟩

lemma *finite-adjacentset*:
assumes *chamber C*
shows *finite* (*adjacentset C*)
⟨*proof*⟩

lemma *label-wrt-eq-on-adjacent-vertex*:
fixes $v\ v' :: 'a$
and $z\ z' :: 'a\ set$
defines $D : D \equiv insert\ v\ z$
and D': $D' \equiv insert\ v'\ z'$
assumes *label* : *label-wrt B f f v* = *f v'*
and *chambers*: *chamber C chamber D chamber D'* $z \lhd C\ z' \lhd C\ D \neq C\ D' \neq C$
shows $D = D'$
⟨*proof*⟩

lemma *face-distance-eq-chamber-distance-compare-other-chamber*:
assumes *chamber C chamber D* $z \lhd C\ z \lhd D\ C \neq D$
chamber-distance C E ≤ *chamber-distance D E*
shows *face-distance z E* = *chamber-distance C E*
⟨*proof*⟩

end

lemma (**in** *ChamberComplexIsomorphism*) *thinish-image-shared-facet*:
assumes *dom*: *domain.chamber C domain.chamber D* $z \lhd C\ z \lhd D\ C \neq D$
and *cod*: *ThinishChamberComplex Y codomain.chamber D'* $f`z \lhd D'$
$D' \neq f`C$
shows $f`D = D'$
⟨*proof*⟩

locale *ThinChamberComplex* = *ChamberComplex X*
for $X :: 'a\ set\ set$
\+ **assumes** *thin*: *chamber C* $\Longrightarrow z \lhd C \Longrightarrow \exists! D \in X - \{C\}.\ z \lhd D$

sublocale *ThinChamberComplex* < *ThinishChamberComplex*
⟨*proof*⟩

context *ThinChamberComplex*
begin

lemma *thinish*: *ThinishChamberComplex X* ⟨*proof*⟩

lemmas *face-distance-eq-chamber-distance-compare-other-chamber* =
face-distance-eq-chamber-distance-compare-other-chamber

abbreviation *the-adj-chamber C z* ≡ *THE D*. $D \in X - \{C\} \wedge z \lhd D$

lemma *the-adj-chamber-simplex*:
chamber C $\Longrightarrow$ *z* $\lhd$ *C* $\Longrightarrow$ *the-adj-chamber C z* $\in$ *X*
⟨*proof*⟩

lemma *the-adj-chamber-facet*: *chamber C* $\Longrightarrow$ *z*$\lhd$*C* $\Longrightarrow$ *z* $\lhd$ *the-adj-chamber C z*
⟨*proof*⟩

lemma *the-adj-chamber-is-adjacent*:
chamber C $\Longrightarrow$ *z*$\lhd$*C* $\Longrightarrow$ *C* $\sim$ *the-adj-chamber C z*
⟨*proof*⟩

lemma *the-adj-chamber*:
chamber C $\Longrightarrow$ *z* $\lhd$ *C* $\Longrightarrow$ *chamber* (*the-adj-chamber C z*)
⟨*proof*⟩

lemma *the-adj-chamber-neq*:
chamber C $\Longrightarrow$ *z* $\lhd$ *C* $\Longrightarrow$ *the-adj-chamber C z* $\neq$ *C*
⟨*proof*⟩

lemma *the-adj-chamber-adjacentset*:
chamber C $\Longrightarrow$ *z*$\lhd$*C* $\Longrightarrow$ *the-adj-chamber C z* $\in$ *adjacentset C*
⟨*proof*⟩

end

lemmas (**in** *ChamberComplexIsomorphism*) *thin-image-shared-facet* =
thinish-image-shared-facet[*OF - - - - - ThinChamberComplex.thinish*]

4.5.2 The standard uniqueness argument for chamber morphisms of thin chamber complexes

context *ThinishChamberComplex*
begin

lemma *standard-uniqueness-dbl*:
assumes *morph* : *ChamberComplexMorphism W X f*
ChamberComplexMorphism W X g
and *chambers*: *ChamberComplex.chamber W C*
ChamberComplex.chamber W D
C$\sim$*D f‘D* $\neq$ *f‘C g‘D* $\neq$ *g‘C chamber* (*g‘D*)
and *funeq* : *fun-eq-on f g C*
shows *fun-eq-on f g D*
⟨*proof*⟩

lemma *standard-uniqueness-pgallery-betw*:
assumes *morph* : *ChamberComplexMorphism W X f*
ChamberComplexMorphism W X g
and *chambers*: *fun-eq-on f g C ChamberComplex.gallery W* (*C*#*Cs*@[*D*])
pgallery (*f*$\models$(*C*#*Cs*@[*D*])) *pgallery* (*g*$\models$(*C*#*Cs*@[*D*]))

shows *fun-eq-on f g D*
⟨*proof*⟩

lemma *standard-uniqueness*:
assumes *morph* : *ChamberComplexMorphism W X f*
ChamberComplexMorphism W X g
and *chamber* : *ChamberComplex.chamber W C fun-eq-on f g C*
and *map-gals*:
$\bigwedge$*Cs*. *ChamberComplex.min-gallery W* (*C#Cs*) $\Longrightarrow$ *pgallery* (*f*⊨(*C#Cs*))
$\bigwedge$*Cs*. *ChamberComplex.min-gallery W* (*C#Cs*) $\Longrightarrow$ *pgallery* (*g*⊨(*C#Cs*))
shows *fun-eq-on f g* ($\bigcup$ *W*)
⟨*proof*⟩

lemma *standard-uniqueness-isomorphs*:
assumes *ChamberComplexIsomorphism W X f*
ChamberComplexIsomorphism W X g
ChamberComplex.chamber W C fun-eq-on f g C
shows *fun-eq-on f g* ($\bigcup$ *W*)
⟨*proof*⟩

lemma *standard-uniqueness-automorphs*:
assumes *ChamberComplexAutomorphism X f*
ChamberComplexAutomorphism X g
chamber C fun-eq-on f g C
shows *f=g*
⟨*proof*⟩

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* :: *'a⇒'a*
begin

abbreviation *opp-chamber* ≡ *folding.opp-chamber*

lemma *adjacent-half-chamber-system-image*:
  assumes *chambers*: $C \in f\vdash\mathcal{C}$ $D \in \mathcal{C}-f\vdash\mathcal{C}$
  and *adjacent*: $C\sim D$
  shows $f`D = C$
⟨*proof*⟩

lemma *adjacent-half-chamber-system-image-reverse*:
  $[\![\ C \in f\vdash\mathcal{C};\ D \in \mathcal{C}-f\vdash\mathcal{C};\ C\sim D\ ]\!] \Longrightarrow$ *opp-chamber* $C = D$
  ⟨*proof*⟩

lemma *chamber-image-closer*:
  assumes $D\in\mathcal{C}-f\vdash\mathcal{C}$ $B\in f\vdash\mathcal{C}$ $B\neq f`D$ *gallery* $(B\#Ds@[D])$
  shows $\exists$ *Cs*. *gallery* $(B\#Cs@[f`D]) \wedge$ *length Cs* $<$ *length Ds*
⟨*proof*⟩

lemma *chamber-image-subset*:
  assumes *D*: $D\in\mathcal{C}-f\vdash\mathcal{C}$
  defines *C*: $C \equiv f`D$
  defines *closerToC* $\equiv$ {$B\in\mathcal{C}$. *chamber-distance B C* $<$ *chamber-distance B D*}
  shows $f\vdash\mathcal{C} \subseteq$ *closerToC*
⟨*proof*⟩

lemma *gallery-double-cross-not-minimal-Cons1*:
  $[\![\ B\in f\vdash\mathcal{C};\ C\in\mathcal{C}-f\vdash\mathcal{C};\ D\in f\vdash\mathcal{C};$ *gallery* $(B\#C\#Cs@[D])\ ]\!] \Longrightarrow$
    $\neg$ *min-gallery* $(B\#C\#Cs@[D])$
  ⟨*proof*⟩

lemma *gallery-double-cross-not-minimal1*:
  $[\![\ B\in f\vdash\mathcal{C};\ C\in\mathcal{C}-f\vdash\mathcal{C};\ D\in f\vdash\mathcal{C};$ *gallery* $(B\#Bs@C\#Cs@[D])\ ]\!] \Longrightarrow$
    $\neg$ *min-gallery* $(B\#Bs@C\#Cs@[D])$
⟨*proof*⟩

end

locale *ThinChamberComplexFolding* =
  ThinChamberComplex X + *folding*: *ChamberComplexFolding X f*
  for *X* :: *'a set set*
  and *f* :: *'a⇒'a*

sublocale *ThinChamberComplexFolding* < *ThinishChamberComplexFolding* ⟨*proof*⟩

context *ThinChamberComplexFolding*
begin

abbreviation *flop* ≡ *folding.flop*

lemmas *adjacent-half-chamber-system-image* = *adjacent-half-chamber-system-image*
lemmas *gallery-double-cross-not-minimal1* = *gallery-double-cross-not-minimal1*

lemmas *gallery-double-cross-not-minimal-Cons1* =
gallery-double-cross-not-minimal-Cons1

lemma *adjacent-preimage*:
assumes *chambers*: $C \in \mathcal{C}-f\vdash\mathcal{C}$ $D \in \mathcal{C}-f\vdash\mathcal{C}$
and *adjacent*: $f`C \sim f`D$
shows $C \sim D$
⟨*proof*⟩

lemma *adjacent-opp-chamber*:
$[\![C{\in}f{\vdash}\mathcal{C};\ D{\in}f{\vdash}\mathcal{C};\ C{\sim}D]\!] \Longrightarrow$ *opp-chamber* $C \sim$ *opp-chamber* D
⟨*proof*⟩

lemma *adjacentchain-preimage*:
set $Cs \subseteq \mathcal{C}-f\vdash\mathcal{C} \Longrightarrow$ *adjacentchain* $(f{\models}Cs) \Longrightarrow$ *adjacentchain* Cs
⟨*proof*⟩

lemma *gallery-preimage*: *set* $Cs \subseteq \mathcal{C}-f\vdash\mathcal{C} \Longrightarrow$ *gallery* $(f{\models}Cs) \Longrightarrow$ *gallery* Cs
⟨*proof*⟩

lemma *chambercomplex-opp-half-apartment*: *ChamberComplex folding.Y*
⟨*proof*⟩

lemma *flop-adj*:
assumes *chamber* C *chamber* D $C{\sim}D$
shows *flop* $C \sim$ *flop* D
⟨*proof*⟩

lemma *flop-gallery*: *gallery* $Cs \Longrightarrow$ *gallery* (*map flop* Cs)
⟨*proof*⟩

lemma *morphism-half-apartments*: *ChamberComplexMorphism folding.Y* $(f{\vdash}X)$ f
⟨*proof*⟩

lemma *chamber-image-complement-closer*:
$[\![D{\in}\mathcal{C}-f{\vdash}\mathcal{C};\ B{\in}\mathcal{C}-f{\vdash}\mathcal{C};\ B{\neq}D;$ *gallery* $(B\#Cs@[f`D])]\!] \Longrightarrow$
$\exists Ds.$ *gallery* $(B\#Ds@[D]) \wedge$ *length* $Ds <$ *length* Cs
⟨*proof*⟩

lemma *chamber-image-complement-subset*:
assumes D: $D{\in}\mathcal{C}-f{\vdash}\mathcal{C}$
defines C: $C \equiv f`D$
defines *closerToD* $\equiv \{B{\in}\mathcal{C}.$ *chamber-distance* B $D <$ *chamber-distance* B $C\}$
shows $\mathcal{C}-f\vdash\mathcal{C} \subseteq$ *closerToD*
⟨*proof*⟩

lemma *chamber-image-and-complement*:
assumes D: $D{\in}\mathcal{C}-f{\vdash}\mathcal{C}$
defines C: $C \equiv f`D$

defines *closerToC* ≡ {*B*∈𝒞. *chamber-distance B C* < *chamber-distance B D*}
and *closerToD* ≡ {*B*∈𝒞. *chamber-distance B D* < *chamber-distance B C*}
shows *f*⊢𝒞 = *closerToC* 𝒞−*f*⊢𝒞 = *closerToD*
⟨*proof*⟩

end

4.6.2 Pairs of opposed foldings

A pair of foldings of a thin chamber complex are opposed or opposite if there is a corresponding pair of adjacent chambers, where each folding sends its corresponding chamber to the other chamber.

locale *OpposedThinChamberComplexFoldings* =
ThinChamberComplex X
+ *folding-f*: *ChamberComplexFolding X f*
+ *folding-g*: *ChamberComplexFolding X g*
for *X* :: ′*a set set*
and *f* :: ′*a*⇒′*a*
and *g* :: ′*a*⇒′*a*
+ **fixes** *C0* :: ′*a set*
assumes *chambers*: *chamber C0 C0*∼*g*‘*C0 C0*≠*g*‘*C0 f*‘*g*‘*C0* = *C0*
begin

abbreviation *D0* ≡ *g*‘*C0*

lemmas *chamber-D0* = *folding-g.chamber-map*[*OF chambers*(*1*)]

lemma *ThinChamberComplexFolding-f*: *ThinChamberComplexFolding X f* ⟨*proof*⟩
lemma *ThinChamberComplexFolding-g*: *ThinChamberComplexFolding X g* ⟨*proof*⟩

lemmas *foldf* = *ThinChamberComplexFolding-f*
lemmas *foldg* = *ThinChamberComplexFolding-g*

lemma *fg-symmetric*: *OpposedThinChamberComplexFoldings X g f D0*
⟨*proof*⟩

lemma *basechambers-half-chamber-systems*: *C0*∈*f*⊢𝒞 *D0*∈*g*⊢𝒞
⟨*proof*⟩

lemmas *basech-halfchsys* =
basechambers-half-chamber-systems

lemma *f-trivial-C0*: *v*∈*C0* ⟹ *f v* = *v*
⟨*proof*⟩

lemmas *g-trivial-D0* =
OpposedThinChamberComplexFoldings.f-trivial-C0[*OF fg-symmetric*]

lemma *double-fold-D0*:

assumes $v \in D0 - C0$
shows $g\ (f\ v) = v$
⟨*proof*⟩

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*⟩

lemmas *flopped-half-chamber-systems-gf* =
OpposedThinChamberComplexFoldings.flopped-half-chamber-systems-fg[
OF fg-symmetric
]

lemma *flopped-half-apartments-fg*: *folding-f.opp-half-apartment* = $g\vdash X$
⟨*proof*⟩

lemmas *flopped-half-apartments-gf* =
OpposedThinChamberComplexFoldings.flopped-half-apartments-fg[
OF fg-symmetric
]

lemma *vertex-set-split*: $\bigcup X = f`(\bigcup X) \cup g`(\bigcup X)$
— f and g will both be the identity on the intersection
⟨*proof*⟩

lemma *half-chamber-system-disjoint-union*:
$\mathcal{C} = f\vdash\mathcal{C} \cup g\vdash\mathcal{C}\ (f\vdash\mathcal{C}) \cap (g\vdash\mathcal{C}) = \{\}$
⟨*proof*⟩

lemmas *halfchsys-decomp* =
half-chamber-system-disjoint-union

lemma *chamber-in-other-half-fg*: *chamber* $C \Longrightarrow C\notin f\vdash\mathcal{C} \Longrightarrow C\in g\vdash\mathcal{C}$
⟨*proof*⟩

lemma *adjacent-half-chamber-system-image-fg*:
$C\in f\vdash\mathcal{C} \Longrightarrow D\in g\vdash\mathcal{C} \Longrightarrow C\sim D \Longrightarrow f`D = C$
⟨*proof*⟩

lemmas *adjacent-half-chamber-system-image-gf* =
OpposedThinChamberComplexFoldings.adjacent-half-chamber-system-image-fg[
OF fg-symmetric
]

lemmas *adjhalfchsys-image-gf* =
adjacent-half-chamber-system-image-gf

lemma *switch-basechamber*:
  assumes $C \in f \vdash \mathcal{C}$ $C \sim g`C$
  shows *OpposedThinChamberComplexFoldings* $X\ f\ g\ C$
⟨*proof*⟩

lemma *unique-half-chamber-system-f*:
  assumes *OpposedThinChamberComplexFoldings* $X\ f'\ g'\ C0$ $g'`C0 = D0$
  shows $f' \vdash \mathcal{C} = f \vdash \mathcal{C}$
⟨*proof*⟩

lemma *unique-half-chamber-system-g*:
  OpposedThinChamberComplexFoldings $X\ f'\ g'\ C0 \Longrightarrow g'`C0 = D0 \Longrightarrow$
    $g' \vdash \mathcal{C} = g \vdash \mathcal{C}$
  ⟨*proof*⟩

lemma *split-gallery-fg*:
  $[\![\ C \in f \vdash \mathcal{C};\ D \in g \vdash \mathcal{C};\ gallery\ (C\#Cs@[D])\ ]\!] \Longrightarrow$
    $\exists\ As\ A\ B\ Bs.\ A \in f \vdash \mathcal{C} \wedge B \in g \vdash \mathcal{C} \wedge C\#Cs@[D] = As@A\#B\#Bs$
  ⟨*proof*⟩

lemmas *split-gallery-gf* =
  OpposedThinChamberComplexFoldings.split-gallery-fg[*OF fg-symmetric*]

end

4.6.3 The automorphism induced by a pair of opposed foldings

Recall that a folding of a chamber complex is a special kind of chamber complex retraction, and so is the identity on its image. Hence a pair of opposed foldings will be the identity on the intersection of their images and so we can stitch them together to create an automorphism of the chamber complex, by allowing each folding to act on the complement of its image. This automorphism will be of order two, and will be the unique automorphism of the chamber complex that fixes pointwise the facet shared by the pair of adjacent chambers associated to the opposed foldings.

context *OpposedThinChamberComplexFoldings*
begin

definition *induced-automorphism* :: $'a \Rightarrow 'a$
  where *induced-automorphism* $v \equiv$
      $if\ v \in f`(\bigcup X)\ then\ g\ v\ else\ if\ v \in g`(\bigcup X)\ then\ f\ v\ else\ v$
— f and g will both be the identity on the intersection of their images
abbreviation s ≡ *induced-automorphism*

lemma *induced-automorphism-fg-symmetric*:
  s = *OpposedThinChamberComplexFoldings*.s $X\ g\ f$
  ⟨*proof*⟩

lemma *induced-automorphism-on-simplices-fg*: $x\in f\vdash X \Longrightarrow v\in x \Longrightarrow \mathrm{s}\ v = g\ v$
⟨*proof*⟩

lemma *induced-automorphism-eq-foldings-on-chambers-fg*:
$C\in f\vdash\mathcal{C} \Longrightarrow$ *fun-eq-on* s g C
⟨*proof*⟩

lemmas *indaut-eq-foldch-fg* =
induced-automorphism-eq-foldings-on-chambers-fg

lemma *induced-automorphism-eq-foldings-on-chambers-gf*:
$C\in g\vdash\mathcal{C} \Longrightarrow$ *fun-eq-on* s f C
⟨*proof*⟩

lemma *induced-automorphism-on-chamber-vertices-f*:
chamber $C \Longrightarrow v\in C \Longrightarrow \mathrm{s}\ v = (\mathit{if}\ C\in f\vdash\mathcal{C}\ \mathit{then}\ g\ v\ \mathit{else}\ f\ v)$
⟨*proof*⟩

lemma *induced-automorphism-simplex-image*:
$C\in f\vdash\mathcal{C} \Longrightarrow x\subseteq C \Longrightarrow \mathrm{s}`x = g`x\ C\in g\vdash\mathcal{C} \Longrightarrow x\subseteq C \Longrightarrow \mathrm{s}`x = f`x$
⟨*proof*⟩

lemma *induced-automorphism-chamber-list-image-fg*:
set $Cs \subseteq f\vdash\mathcal{C} \Longrightarrow \mathrm{s}\models Cs = g\models Cs$
⟨*proof*⟩

lemma *induced-automorphism-chamber-image-fg*:
chamber $C \Longrightarrow \mathrm{s}`C = (\mathit{if}\ C\in f\vdash\mathcal{C}\ \mathit{then}\ g`C\ \mathit{else}\ f`C)$
⟨*proof*⟩

lemma *induced-automorphism-C0*: $\mathrm{s}`C0 = D0$
⟨*proof*⟩

lemma *induced-automorphism-fixespointwise-C0-int-D0*:
fixespointwise s $(C0\cap D0)$
⟨*proof*⟩

lemmas *indaut-fixes-fundfacet* =
induced-automorphism-fixespointwise-C0-int-D0

lemma *induced-automorphism-adjacent-half-chamber-system-image-fg*:
$[\![\ C\in f\vdash\mathcal{C};\ D\in g\vdash\mathcal{C};\ C\sim D\ ]\!] \Longrightarrow \mathrm{s}`D = C$
⟨*proof*⟩

lemmas *indaut-adj-halfchsys-im-fg* =
induced-automorphism-adjacent-half-chamber-system-image-fg

lemma *induced-automorphism-chamber-map*: *chamber* $C \Longrightarrow$ *chamber* $(\mathrm{s}`C)$
⟨*proof*⟩

lemmas *indaut-chmap* = *induced-automorphism-chamber-map*

lemma *induced-automorphism-ntrivial*: s ≠ *id*
⟨*proof*⟩

lemma *induced-automorphism-bij-between-half-chamber-systems-f*:
bij-betw ((') s) (𝒞−*f*⊢𝒞) (*f*⊢𝒞)
⟨*proof*⟩

lemmas *indaut-bij-btw-halfchsys-f* =
induced-automorphism-bij-between-half-chamber-systems-f

lemma *induced-automorphism-bij-between-half-chamber-systems-g*:
bij-betw ((') s) (𝒞−*g*⊢𝒞) (*g*⊢𝒞)
⟨*proof*⟩

lemma *induced-automorphism-halfmorphism-fopp-to-fimage*:
ChamberComplexMorphism folding-f.opp-half-apartment (*f*⊢*X*) s
⟨*proof*⟩

lemmas *indaut-halfmorph-fopp-fim* =
induced-automorphism-halfmorphism-fopp-to-fimage

lemma *induced-automorphism-half-chamber-system-gallery-map-f*:
set Cs ⊆ *f*⊢𝒞 ⟹ *gallery Cs* ⟹ *gallery* (s⊨*Cs*)
⟨*proof*⟩

lemma *induced-automorphism-half-chamber-system-pgallery-map-f*:
set Cs ⊆ *f*⊢𝒞 ⟹ *pgallery Cs* ⟹ *pgallery* (s⊨*Cs*)
⟨*proof*⟩

lemmas *indaut-halfchsys-pgal-map-f* =
induced-automorphism-half-chamber-system-pgallery-map-f

lemma *induced-automorphism-half-chamber-system-pgallery-map-g*:
set Cs ⊆ *g*⊢𝒞 ⟹ *pgallery Cs* ⟹ *pgallery* (s⊨*Cs*)
⟨*proof*⟩

lemma *induced-automorphism-halfmorphism-fimage-to-fopp*:
ChamberComplexMorphism (*f*⊢*X*) *folding-f.opp-half-apartment* s
⟨*proof*⟩

lemma *induced-automorphism-selfcomp-halfmorphism-f*:
ChamberComplexMorphism (*f*⊢*X*) (*f*⊢*X*) (s∘s)
⟨*proof*⟩

lemma *induced-automorphism-selfcomp-halftrivial-f*: *fixespointwise* (s∘s) (⋃(*f*⊢*X*))
⟨*proof*⟩

lemmas *indaut-selfcomp-halftriv-f* =
induced-automorphism-selfcomp-halftrivial-f

lemma *induced-automorphism-selfcomp-halftrivial-g*: *fixespointwise* (s∘s) (⋃ (g⊢X))
⟨*proof*⟩

lemma *induced-automorphism-trivial-outside*:
assumes $v \notin \bigcup X$
shows s $v = v$
⟨*proof*⟩

lemma *induced-automorphism-morphism*: *ChamberComplexEndomorphism X* s
⟨*proof*⟩

lemmas *indaut-morph* = *induced-automorphism-morphism*

lemma *induced-automorphism-morphism-order2*: s∘s = *id*
⟨*proof*⟩

lemmas *indaut-order2* = *induced-automorphism-morphism-order2*

lemmas *induced-automorphism-bij* =
o-bij[*OF*
induced-automorphism-morphism-order2
induced-automorphism-morphism-order2
]

lemma *induced-automorphism-surj-on-vertexset*: s‘(⋃ X) = ⋃ X
⟨*proof*⟩

lemma *induced-automorphism-bij-betw-vertexset*: *bij-betw* s (⋃ X) (⋃ X)
⟨*proof*⟩

lemma *induced-automorphism-surj-on-simplices*: s⊢X = X
⟨*proof*⟩

lemma *induced-automorphism-automorphism*:
ChamberComplexAutomorphism X s
⟨*proof*⟩

lemmas *indaut-aut* = *induced-automorphism-automorphism*

lemma *induced-automorphism-unique-automorphism'*:
assumes *ChamberComplexAutomorphism X s s≠id fixespointwise s* (*C0*∩*D0*)
shows *fun-eq-on s* s *C0*
⟨*proof*⟩

lemma *induced-automorphism-unique-automorphism*:

⟦ *ChamberComplexAutomorphism X s*; *s*≠*id*; *fixespointwise s* (*C0*∩*D0*) ⟧
  ⟹ *s* = s
⟨*proof*⟩

lemmas *indaut-uniq-aut* =
  induced-automorphism-unique-automorphism

lemma *induced-automorphism-unique*:
  OpposedThinChamberComplexFoldings X f′ g′ C0 ⟹ *g′‘C0* = *g‘C0* ⟹
    OpposedThinChamberComplexFoldings.induced-automorphism X f′ g′ = s
  ⟨*proof*⟩

lemma *induced-automorphism-sym*:
  OpposedThinChamberComplexFoldings.induced-automorphism X g f = s
  ⟨*proof*⟩

lemma *induced-automorphism-respects-labels*:
  assumes *label-wrt B φ v*∈(⋃*X*)
  shows φ (s *v*) = φ *v*
⟨*proof*⟩

lemmas *indaut-resplabels* =
  induced-automorphism-respects-labels

end

4.6.4 Walls

A pair of opposed foldings of a thin chamber complex defines a decomposition of the chamber system into the two disjoint chamber system images. Call such a decomposition a wall, as we image that disjointness erects a wall between the two half chamber systems. By considering the collection of all possible opposed folding pairs, and their associated walls, we can obtain information about minimality of galleries by considering the walls they cross.

context *ThinChamberComplex*
begin

definition *foldpairs* :: ((′*a*⇒′*a*) × (′*a*⇒′*a*)) *set*
  where *foldpairs* ≡ {(*f*,*g*). ∃ *C*. *OpposedThinChamberComplexFoldings X f g C*}

abbreviation *walls* ≡ ⋃(*f*,*g*)∈*foldpairs*. {{*f*⊢𝒞,*g*⊢𝒞}}
abbreviation *the-wall-betw C D* ≡
    THE-default {} (λ*H*. *H*∈*walls* ∧ *separated-by H C D*)

definition *walls-betw* :: ′*a set* ⇒ ′*a set* ⇒ ′*a set set set set*
  where *walls-betw C D* ≡ {*H*∈*walls*. *separated-by H C D*}

fun *wall-crossings* :: *'a set list* ⇒ *'a set set set list*
  where *wall-crossings* [] = []
  | *wall-crossings* [*C*] = []
  | *wall-crossings* (*B*#*C*#*Cs*) = *the-wall-betw B C* # *wall-crossings* (*C*#*Cs*)

lemma *foldpairs-sym*: (*f*,*g*)∈*foldpairs* ⟹ (*g*,*f*)∈*foldpairs*
  ⟨*proof*⟩

lemma *not-self-separated-by-wall*: *H*∈*walls* ⟹ ¬ *separated-by H C C*
  ⟨*proof*⟩

lemma *the-wall-betw-nempty*:
  assumes *the-wall-betw C D* ≠ {}
  shows *the-wall-betw C D* ∈ *walls separated-by* (*the-wall-betw C D*) *C D*
⟨*proof*⟩

lemma *the-wall-betw-self-empty*: *the-wall-betw C C* = {}
⟨*proof*⟩

lemma *length-wall-crossings*: *length* (*wall-crossings Cs*) = *length Cs* − *1*
  ⟨*proof*⟩

lemma *wall-crossings-snoc*:
  wall-crossings (*Cs*@[*D*,*E*]) = *wall-crossings* (*Cs*@[*D*]) @ [*the-wall-betw D E*]
  ⟨*proof*⟩

lemma *wall-crossings-are-walls*:
  H∈*set* (*wall-crossings Cs*) ⟹ *H*≠{} ⟹ *H*∈*walls*
⟨*proof*⟩

lemma *in-set-wall-crossings-decomp*:
  H∈*set* (*wall-crossings Cs*) ⟹
    ∃ *As A B Bs*. *Cs* = *As*@[*A*,*B*]@*Bs* ∧ *H* = *the-wall-betw A B*
⟨*proof*⟩

end

context *OpposedThinChamberComplexFoldings*
begin

lemma *foldpair*: (*f*,*g*)∈*foldpairs*
  ⟨*proof*⟩

lemma *separated-by-this-wall-fg*:
  separated-by {*f*⊢𝒞,*g*⊢𝒞} *C D* ⟹ *C*∈*f*⊢𝒞 ⟹ *D*∈*g*⊢𝒞
  ⟨*proof*⟩

lemmas *separated-by-this-wall-gf* =
  OpposedThinChamberComplexFoldings.separated-by-this-wall-fg[

OF fg-symmetric
]

lemma *induced-automorphism-this-wall-vertex*:
assumes *C*∈*f*⊢*𝒞* *D*∈*g*⊢*𝒞* *v*∈*C*∩*D*
shows s *v* = *v*
⟨*proof*⟩

lemmas *indaut-wallvertex* =
induced-automorphism-this-wall-vertex

lemma *unique-wall*:
assumes *opp′* : *OpposedThinChamberComplexFoldings X f′ g′ C′*
and *chambers*: *A*∈*f*⊢*𝒞* *A*∈*f′*⊢*𝒞* *B*∈*g*⊢*𝒞* *B*∈*g′*⊢*𝒞* *A*∼*B*
shows {*f*⊢*𝒞*,*g*⊢*𝒞*} = {*f′*⊢*𝒞*,*g′*⊢*𝒞*}
⟨*proof*⟩

end

context *ThinChamberComplex*
begin

lemma *separated-by-wall-ex-foldpair*:
assumes *H*∈*walls separated-by H C D*
shows ∃(*f*,*g*)∈*foldpairs*. *H* = {*f*⊢*𝒞*,*g*⊢*𝒞*} ∧ *C*∈*f*⊢*𝒞* ∧ *D*∈*g*⊢*𝒞*
⟨*proof*⟩

lemma *not-separated-by-wall-ex-foldpair*:
assumes *chambers*: *chamber C chamber D*
and *wall* : *H*∈*walls* ¬ *separated-by H C D*
shows ∃(*f*,*g*)∈*foldpairs*. *H* = {*f*⊢*𝒞*,*g*⊢*𝒞*} ∧ *C*∈*f*⊢*𝒞* ∧ *D*∈*f*⊢*𝒞*
⟨*proof*⟩

lemma *adj-wall-imp-ex1-wall*:
assumes *adj* : *C*∼*D*
and *wall*: *H0*∈*walls separated-by H0 C D*
shows ∃!*H*∈*walls*. *separated-by H C D*
⟨*proof*⟩

end

context *OpposedThinChamberComplexFoldings*
begin

lemma *this-wall-betwI*:
assumes *C*∈*f*⊢*𝒞* *D*∈*g*⊢*𝒞* *C*∼*D*
shows *the-wall-betw C D* = {*f*⊢*𝒞*,*g*⊢*𝒞*}
⟨*proof*⟩

lemma *this-wall-betw-basechambers*:
the-wall-betw C0 D0 = {*f*⊢𝒞,*g*⊢𝒞}
⟨*proof*⟩

lemma *this-wall-in-crossingsI-fg*:
defines *H*: *H* ≡ {*f*⊢𝒞,*g*⊢𝒞}
assumes *D*: *D*∈*g*⊢𝒞
shows *C*∈*f*⊢𝒞 ⟹ *gallery* (*C*#*Cs*@[*D*]) ⟹ *H* ∈ *set* (*wall-crossings* (*C*#*Cs*@[*D*]))
⟨*proof*⟩

end

lemma (**in** *ThinChamberComplex*) *walls-betw-subset-wall-crossings*:
assumes *gallery* (*C*#*Cs*@[*D*])
shows *walls-betw C D* ⊆ *set* (*wall-crossings* (*C*#*Cs*@[*D*]))
⟨*proof*⟩

context *OpposedThinChamberComplexFoldings*
begin

lemma *same-side-this-wall-wall-crossings-not-distinct-f*:
gallery (*C*#*Cs*@[*D*]) ⟹ *C*∈*f*⊢𝒞 ⟹ *D*∈*f*⊢𝒞 ⟹
{*f*⊢𝒞,*g*⊢𝒞}∈*set* (*wall-crossings* (*C*#*Cs*@[*D*])) ⟹
¬ *distinct* (*wall-crossings* (*C*#*Cs*@[*D*]))
⟨*proof*⟩

lemmas *sside-wcrossings-ndistinct-f* =
same-side-this-wall-wall-crossings-not-distinct-f

lemma *separated-by-this-wall-chain3-fg*:
assumes *B*∈*f*⊢𝒞 *chamber C chamber D*
separated-by {*f*⊢𝒞,*g*⊢𝒞} *B C separated-by* {*f*⊢𝒞,*g*⊢𝒞} *C D*
shows *C*∈*g*⊢𝒞 *D*∈*f*⊢𝒞
⟨*proof*⟩

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*]))
∀ *H*∈*set* (*wall-crossings* (*C*#*Cs*@[*D*])). *separated-by H C D*
shows *min-gallery* (*C*#*Cs*@[*D*])
⟨*proof*⟩

lemma *ex-nonseparating-wall-imp-wall-crossings-not-distinct*:
assumes *gal* : *gallery* (*C*#*Cs*@[*D*])
and *wall*: *H*∈*set* (*wall-crossings* (*C*#*Cs*@[*D*])) *H*≠{}
¬ *separated-by H C D*
shows ¬ *distinct* (*wall-crossings* (*C*#*Cs*@[*D*]))
⟨*proof*⟩

lemma *not-min-gallery-double-crosses-wall*:
assumes *gallery Cs* ¬ *min-gallery Cs* {} ∉ *set* (*wall-crossings Cs*)
shows ¬ *distinct* (*wall-crossings Cs*)
⟨*proof*⟩

lemma *not-distinct-crossings-split-gallery*:
⟦ *gallery Cs*; {} ∉ *set* (*wall-crossings Cs*); ¬ *distinct* (*wall-crossings Cs*) ⟧ ⟹
∃*f g As A B Bs E F Fs.*
(*f*,*g*)∈*foldpairs* ∧ *A*∈*f*⊢𝒞 ∧ *B*∈*g*⊢𝒞 ∧ *E*∈*g*⊢𝒞 ∧ *F*∈*f*⊢𝒞 ∧
(*Cs* = *As*@[*A*,*B*,*F*]@*Fs* ∨ *Cs* = *As*@[*A*,*B*]@*Bs*@[*E*,*F*]@*Fs*)
⟨*proof*⟩

lemma *not-min-gallery-double-split*:
⟦ *gallery Cs*; ¬ *min-gallery Cs*; {} ∉ *set* (*wall-crossings Cs*) ⟧ ⟹
∃*f g As A B Bs E F Fs.*
(*f*,*g*)∈*foldpairs* ∧ *A*∈*f*⊢𝒞 ∧ *B*∈*g*⊢𝒞 ∧ *E*∈*g*⊢𝒞 ∧ *F*∈*f*⊢𝒞 ∧
(*Cs* = *As*@[*A*,*B*,*F*]@*Fs* ∨ *Cs* = *As*@[*A*,*B*]@*Bs*@[*E*,*F*]@*Fs*)
⟨*proof*⟩

end

4.7 Thin chamber complexes with many foldings

Here we begin to examine thin chamber complexes in which every pair of adjacent chambers affords a pair of opposed foldings of the complex. This condition will ultimately be shown to be sufficient to ensure that a thin chamber complex is isomorphic to some Coxeter complex.

4.7.1 Locale definition and basic facts

locale *ThinChamberComplexManyFoldings* = *ThinChamberComplex X*
for *X* :: *'a set set*
+ **fixes** *C0* :: *'a set*
assumes *fundchamber*: *chamber C0*
and *ex-walls* :
⟦ *chamber C*; *chamber D*; *C*∼*D*; *C*≠*D* ⟧ ⟹
∃*f g*. *OpposedThinChamberComplexFoldings X f g C* ∧ *D*=*g*‘*C*

lemma (**in** *ThinChamberComplex*) *ThinChamberComplexManyFoldingsI*:
assumes *chamber C0*
and ⋀*C D*. ⟦ *chamber C*; *chamber D*; *C*∼*D*; *C*≠*D* ⟧ ⟹

$\exists f\ g.\ OpposedThinChamberComplexFoldings\ X\ f\ g\ C \wedge D{=}g`C$
shows *ThinChamberComplexManyFoldings X C0*
⟨*proof*⟩

lemma (**in** *ThinChamberComplexManyFoldings*) *wall-crossings-subset-walls-betw*:
assumes *min-gallery* (*C#Cs@[D]*)
shows *set* (*wall-crossings* (*C#Cs@[D]*)) ⊆ *walls-betw C D*
⟨*proof*⟩

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⇒'a)×('a⇒'a)) *set*
where *fundfoldpairs* ≡ {(*f*,*g*). *OpposedThinChamberComplexFoldings X f g C0*}

abbreviation *fundadjset* ≡ *adjacentset C0* − {*C0*}

abbreviation *induced-automorph* :: ('a⇒'a) ⇒ ('a⇒'a) ⇒ ('a⇒'a)
where *induced-automorph f g* ≡
OpposedThinChamberComplexFoldings.induced-automorphism X f g

abbreviation *Abs-induced-automorph* :: ('a⇒'a) ⇒ ('a⇒'a) ⇒ 'a *permutation*
where *Abs-induced-automorph f g* ≡ *Abs-permutation* (*induced-automorph f g*)

abbreviation $S \equiv \bigcup(f,g){\in}fundfoldpairs.\ \{Abs\text{-}induced\text{-}automorph\ f\ g\}$
abbreviation $W \equiv \langle S\rangle$

lemma *fundfoldpairs-induced-autormorph-bij*:
(*f*,*g*) ∈ *fundfoldpairs* ⟹ *bij* (*induced-automorph f g*)
⟨*proof*⟩

lemmas *permutation-conv-induced-automorph* =
Abs-permutation-inverse[*OF CollectI*, *OF fundfoldpairs-induced-autormorph-bij*]

lemma *fundfoldpairs-induced-autormorph-order2*:
(*f*,*g*) ∈ *fundfoldpairs* ⟹ *induced-automorph f g* ∘ *induced-automorph f g* = *id*
⟨*proof*⟩

lemma *fundfoldpairs-induced-autormorph-ntrivial*:

$(f,g) \in$ *fundfoldpairs* $\Longrightarrow$ *induced-automorph f g* $\neq$ *id*
⟨*proof*⟩

lemma *fundfoldpairs-fundchamber-image*:
$(f,g)\in$*fundfoldpairs* $\Longrightarrow$ *Abs-induced-automorph f g* ‘→ *C0* = *g*‘*C0*
⟨*proof*⟩

lemma *fundfoldpair-fundchamber-in-half-chamber-system-f*:
$(f,g)\in$*fundfoldpairs* $\Longrightarrow$ *C0*∈*f*⊢$\mathcal{C}$
⟨*proof*⟩

lemma *fundfoldpair-unique-half-chamber-system-f*:
assumes $(f,g)\in$*fundfoldpairs* $(f',g')\in$*fundfoldpairs*
Abs-induced-automorph f′ g′ = *Abs-induced-automorph f g*
shows *f′*⊢$\mathcal{C}$ = *f*⊢$\mathcal{C}$
⟨*proof*⟩

lemma *fundfoldpair-unique-half-chamber-systems-chamber-ng-f*:
assumes $(f,g)\in$*fundfoldpairs* $(f',g')\in$*fundfoldpairs*
Abs-induced-automorph f′ g′ = *Abs-induced-automorph f g*
chamber C C∉*g*⊢$\mathcal{C}$
shows *C*∈*f′*⊢$\mathcal{C}$
⟨*proof*⟩

lemma *the-wall-betw-adj-fundchamber*:
$(f,g)\in$*fundfoldpairs* $\Longrightarrow$
the-wall-betw C0 (*Abs-induced-automorph f g* ‘→ *C0*) = {*f*⊢$\mathcal{C}$,*g*⊢$\mathcal{C}$}
⟨*proof*⟩

lemma *zero-notin-S*: *0*∉*S*
⟨*proof*⟩

lemma *S-order2-add*: *s*∈*S* $\Longrightarrow$ *s* + *s* = *0*
⟨*proof*⟩

lemma *S-add-order2*:
assumes *s*∈*S*
shows *add-order s* = *2*
⟨*proof*⟩

lemmas *S-uminus* = *minus-unique*[*OF S-order2-add*]

lemma *S-sym*: *uminus* ‘ *S* ⊆ *S*
⟨*proof*⟩

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∈*S* ⟹ *ChamberComplexEndomorphism X* (*permutation s*)
⟨*proof*⟩

lemma *S-list-endomorphism*:
ss∈*lists S* ⟹ *ChamberComplexEndomorphism X* (*permutation* (*sum-list ss*))
⟨*proof*⟩

lemma *W-endomorphism*:
w∈*W* ⟹ *ChamberComplexEndomorphism X* (*permutation w*)
⟨*proof*⟩

lemma *S-automorphism*:
s∈*S* ⟹ *ChamberComplexAutomorphism X* (*permutation s*)
⟨*proof*⟩

lemma *S-list-automorphism*:
ss∈*lists S* ⟹ *ChamberComplexAutomorphism X* (*permutation* (*sum-list ss*))
⟨*proof*⟩

lemma *W-automorphism*:
w∈*W* ⟹ *ChamberComplexAutomorphism X* (*permutation w*)
⟨*proof*⟩

lemma *S-respects-labels*: ⟦ *label-wrt B φ*; *s*∈*S*; *v*∈(⋃*X*) ⟧ ⟹ *φ* (*s* → *v*) = *φ v*
⟨*proof*⟩

lemma *S-list-respects-labels*:
⟦ *label-wrt B φ*; *ss*∈*lists S*; *v*∈(⋃*X*) ⟧ ⟹ *φ* (*sum-list ss* → *v*) = *φ v*
⟨*proof*⟩

lemma *W-respects-labels*:
⟦ *label-wrt B φ*; *w*∈*W*; *v*∈(⋃*X*) ⟧ ⟹ *φ* (*w*→*v*) = *φ v*
⟨*proof*⟩

end

4.7.3 Action of the group of automorphisms on the chamber system

Now we examine the action of the group W on the chamber system. In particular, we show that the action is transitive.

context *ThinChamberComplexManyFoldings*
begin

lemma *fundchamber-S-chamber*: *s*∈*S* ⟹ *chamber* (*s*‘→*C0*)
⟨*proof*⟩

lemma *fundchamber-W-image-chamber*:
w∈*W* ⟹ *chamber* (*w*‘→*C0*)

⟨*proof*⟩

lemma *fundchamber-S-adjacent*: $s{\in}S \Longrightarrow C0 \sim (s\text{‘}{\rightarrow}C0)$
⟨*proof*⟩

lemma *fundchamber-WS-image-adjacent*:
$w{\in}W \Longrightarrow s{\in}S \Longrightarrow (w\text{‘}{\rightarrow}C0) \sim ((w{+}s)\text{‘}{\rightarrow}C0)$
⟨*proof*⟩

lemma *fundchamber-S-image-neq-fundchamber*: $s{\in}S \Longrightarrow s\text{‘}{\rightarrow}C0 \neq C0$
⟨*proof*⟩

lemma *fundchamber-next-WS-image-neq*:
assumes $s{\in}S$
shows $(w{+}s)$ ‘$\rightarrow$ $C0 \neq w$ ‘$\rightarrow$ $C0$
⟨*proof*⟩

lemma *fundchamber-S-fundadjset*: $s{\in}S \Longrightarrow s\text{‘}{\rightarrow}C0 \in \mathit{fundadjset}$
⟨*proof*⟩

lemma *fundadjset-eq-S-image*: $D{\in}\mathit{fundadjset} \Longrightarrow \exists\, s{\in}S.\ D = s\text{‘}{\rightarrow}C0$
⟨*proof*⟩

lemma *S-fixespointwise-fundchamber-image-int*:
assumes $s{\in}S$
shows *fixespointwise* $((\rightarrow)\ s)\ (C0{\cap}s\text{‘}{\rightarrow}C0)$
⟨*proof*⟩

lemma *S-fixes-fundchamber-image-int*:
$s{\in}S \Longrightarrow s\text{‘}{\rightarrow}(C0{\cap}s\text{‘}{\rightarrow}C0) = C0{\cap}s\text{‘}{\rightarrow}C0$
⟨*proof*⟩

lemma *fundfacets*:
assumes $s{\in}S$
shows $C0{\cap}s\text{‘}{\rightarrow}C0 \lhd C0\ C0{\cap}s\text{‘}{\rightarrow}C0 \lhd s\text{‘}{\rightarrow}C0$
⟨*proof*⟩

lemma *fundadjset-ex1-eq-S-image*:
assumes $D{\in}\mathit{fundadjset}$
shows $\exists!s{\in}S.\ D = s\text{‘}{\rightarrow}C0$
⟨*proof*⟩

lemma *fundchamber-S-image-inj-on*: *inj-on* $(\lambda s.\ s\text{‘}{\rightarrow}C0)\ S$
⟨*proof*⟩

lemma *S-list-image-gallery*:
$ss{\in}\mathit{lists}\ S \Longrightarrow \mathit{gallery}\ (\mathit{map}\ (\lambda w.\ w\text{‘}{\rightarrow}C0)\ (\mathit{sums}\ ss))$
⟨*proof*⟩

lemma *pgallery-last-eq-W-image*:
pgallery $(C0\#Cs@[C]) \Longrightarrow \exists\, w\in W.\ C = w`\rightarrow C0$
⟨*proof*⟩

lemma *chamber-eq-W-image*:
assumes *chamber C*
shows $\exists\, w\in W.\ C = w`\rightarrow C0$
⟨*proof*⟩

lemma *S-list-image-crosses-walls*:
$ss \in lists\ S \Longrightarrow \{\} \notin set\ (wall\text{-}crossings\ (map\ (\lambda w.\ w`\rightarrow C0)\ (sums\ ss)))$
⟨*proof*⟩

end

4.7.4 A labelling by the vertices of the fundamental chamber

Here we show that by repeatedly applying the composition of all the elements in the collection S of fundamental automorphisms, we can retract the entire chamber complex onto the fundamental chamber. This retraction provides a means of labelling the chamber complex, using the vertices of the fundamental chamber as labels.

context *ThinChamberComplexManyFoldings*
begin

definition *Spair* :: $'a\ permutation \Rightarrow ('a\Rightarrow'a)\times('a\Rightarrow'a)$
where *Spair s* ≡
SOME fg. fg ∈ *fundfoldpairs* ∧ *s* = *case-prod Abs-induced-automorph fg*

lemma *Spair-fundfoldpair*: *s*∈*S* ⟹ *Spair s* ∈ *fundfoldpairs*
⟨*proof*⟩

lemma *Spair-induced-automorph*:
s∈*S* ⟹ *s* = *case-prod Abs-induced-automorph* (*Spair s*)
⟨*proof*⟩

lemma *S-list-pgallery-decomp1*:
assumes *ss*: *set ss* = *S* **and** *gal*: *Cs*≠[] *pgallery* (*C0#Cs*)
shows ∃ *s*∈*set ss*. ∃ *C*∈*set Cs*. ∀ (*f*,*g*)∈*fundfoldpairs*.
s = *Abs-induced-automorph f g* ⟶ *C* ∈ *g*⊢𝒞
⟨*proof*⟩

lemma *S-list-pgallery-decomp2*:
assumes *set ss* = *S Cs*≠[] *pgallery* (*C0#Cs*)
shows
∃ *rs s ts*. *ss* = *rs*@*s*#*ts* ∧
(∃ *C*∈*set Cs*. ∀ (*f*,*g*)∈*fundfoldpairs*.
s = *Abs-induced-automorph f g* ⟶ *C* ∈ *g*⊢𝒞) ∧

(∀ *r*∈*set rs*. ∀ *C*∈*set Cs*. ∀ (*f*,*g*)∈*fundfoldpairs*.
r = *Abs-induced-automorph f g* ⟶ *C*∈*f*⊢𝒞)
⟨*proof*⟩

lemma *S-list-pgallery-decomp3*:
assumes *set ss* = *S Cs*≠[] *pgallery* (*C0*#*Cs*)
shows
∃ *rs s ts As B Bs*. *ss* = *rs*@*s*#*ts* ∧ *Cs* = *As*@*B*#*Bs* ∧
(∀ (*f*,*g*)∈*fundfoldpairs*. *s* = *Abs-induced-automorph f g* ⟶ *B*∈*g*⊢𝒞) ∧
(∀ *A*∈*set As*. ∀ (*f*,*g*)∈*fundfoldpairs*.
s = *Abs-induced-automorph f g* ⟶ *A*∈*f*⊢𝒞) ∧
(∀ *r*∈*set rs*. ∀ *C*∈*set Cs*. ∀ (*f*,*g*)∈*fundfoldpairs*.
r = *Abs-induced-automorph f g* ⟶ *C*∈*f*⊢𝒞)
⟨*proof*⟩

lemma *fundfold-trivial-f𝒞*:
r∈*S* ⟹ ∀ (*f*,*g*)∈*fundfoldpairs*. *r* = *Abs-induced-automorph f g* ⟶ *C*∈*f*⊢𝒞 ⟹
fst (*Spair r*) ' *C* = *C*
⟨*proof*⟩

lemma *fundfold-comp-trivial-f𝒞*:
set rs ⊆ *S* ⟹
∀ *r*∈*set rs*. ∀ (*f*,*g*)∈*fundfoldpairs*.
r = *Abs-induced-automorph f g* ⟶ *C*∈*f*⊢𝒞 ⟹
fold fst (*map Spair rs*) ' *C* = *C*
⟨*proof*⟩

lemma *fundfold-trivial-f𝒞-list*:
r∈*S* ⟹
∀ *C*∈*set Cs*. ∀ (*f*,*g*)∈*fundfoldpairs*.
r = *Abs-induced-automorph f g* ⟶ *C*∈*f*⊢𝒞 ⟹
fst (*Spair r*) ⊨ *Cs* = *Cs*
⟨*proof*⟩

lemma *fundfold-comp-trivial-f𝒞-list*:
set rs ⊆ *S* ⟹
∀ *r*∈*set rs*. ∀ *C*∈*set Cs*. ∀ (*f*,*g*)∈*fundfoldpairs*.
r = *Abs-induced-automorph f g* ⟶ *C*∈*f*⊢𝒞 ⟹
fold fst (*map Spair rs*) ⊨ *Cs* = *Cs*
⟨*proof*⟩

lemma *fundfold-gallery-map*:
s∈*S* ⟹ *gallery Cs* ⟹ *gallery* (*fst* (*Spair s*) ⊨ *Cs*)
⟨*proof*⟩

lemma *fundfold-comp-gallery-map*:
assumes *pregal*: *gallery Cs*
shows *set ss* ⊆ *S* ⟹ *gallery* (*fold fst* (*map Spair ss*) ⊨ *Cs*)
⟨*proof*⟩

lemma *fundfold-comp-pgallery-ex-funpow*:
assumes *ss*: *set ss* = *S*
shows *pgallery* (*C0*#*Cs*@[*C*]) $\Longrightarrow$
$\exists$ *n*. (*fold fst* (*map Spair ss*) ^^ *n*) ‘ *C* = *C0*
⟨*proof*⟩

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*⟩

lemma *fundfold-comp-fixespointwise-C0*:
assumes *set ss* $\subseteq$ *S*
shows *fixespointwise* (*fold fst* (*map Spair ss*)) *C0*
⟨*proof*⟩

lemma *fundfold-comp-endomorphism*:
assumes *set ss* $\subseteq$ *S*
shows *ChamberComplexEndomorphism X* (*fold fst* (*map Spair ss*))
⟨*proof*⟩

lemma *finite-S*: *finite S*
⟨*proof*⟩

lemma *ex-label-retraction*: $\exists$ φ. *label-wrt C0* φ $\wedge$ *fixespointwise* φ *C0*
⟨*proof*⟩

lemma *ex-label-map*: $\exists$ φ. *label-wrt C0* φ
⟨*proof*⟩

end

4.7.5 More on the action of the group of automorphisms on chambers

Recall that we have already verified that W acts transitively on the chamber system. We now use the labelling of the chamber complex examined in the previous section to show that this action is simply transitive.

context *ThinChamberComplexManyFoldings*
begin

lemma *fundchamber-W-image-ker*:
assumes *w*$\in$*W w*‘$\rightarrow$*C0* = *C0*
shows *w* = *0*
⟨*proof*⟩

lemma *fundchamber-W-image-inj-on*:

inj-on (*λw. w‘→C0*) *W*
⟨*proof*⟩

end

4.7.6 A bijection between the fundamental chamber and the set of generating automorphisms

Removing a single vertex from the fundamental chamber determines a facet, a facet in the fundamental chamber determines an adjacent chamber (since our complex is thin), and a chamber adjacent to the fundamental chamber determines an automorphism (via some pair of opposed foldings) in our generating set *S*. Here we show that this correspondence is bijective.

context *ThinChamberComplexManyFoldings*
begin

definition *fundantivertex* :: *'a permutation ⇒ 'a*
where *fundantivertex s ≡* (*THE v. v ∈ C0−s‘→C0*)

abbreviation *fundantipermutation ≡ the-inv-into S fundantivertex*

lemma *fundantivertex*: *s∈S ⟹ fundantivertex s ∈ C0−s‘→C0*
⟨*proof*⟩

lemma *fundantivertex-fundchamber-decomp*:
s∈S ⟹ C0 = insert (*fundantivertex s*) (*C0∩s‘→C0*)
⟨*proof*⟩

lemma *fundantivertex-unstable*:
s∈S ⟹ s → fundantivertex s ≠ fundantivertex s
⟨*proof*⟩

lemma *fundantivertex-inj-on*: *inj-on fundantivertex S*
⟨*proof*⟩

lemma *fundantivertex-surj-on*: *fundantivertex ‘ S = C0*
⟨*proof*⟩

lemma *fundantivertex-bij-betw*: *bij-betw fundantivertex S C0*
⟨*proof*⟩

lemma *card-S-fundchamber*: *card S = card C0*
⟨*proof*⟩

lemma *card-S-chamber*:
chamber C ⟹ card C = card S
⟨*proof*⟩

lemma *fundantipermutation1*:
  $v \in C0 \Longrightarrow$ *fundantipermutation* $v \in S$
  ⟨*proof*⟩

end

4.8 Thick chamber complexes

A thick chamber complex is one in which every facet is a facet of at least three chambers.

locale *ThickChamberComplex* = *ChamberComplex X*
  for X :: *'a set set*
+ **assumes** *thick*:
    chamber $C \Longrightarrow z \lhd C \Longrightarrow$
      $\exists D\ E.\ D \in X - \{C\} \wedge z \lhd D \wedge E \in X - \{C,D\} \wedge z \lhd E$
begin

definition *some-third-chamber* :: *'a set* ⇒ *'a set* ⇒ *'a set* ⇒ *'a set*
  where *some-third-chamber C D z* ≡ *SOME E*. $E \in X - \{C,D\} \wedge z \lhd E$

lemma *facet-ex-third-chamber*: *chamber* $C \Longrightarrow z \lhd C \Longrightarrow \exists E \in X - \{C,D\}.\ z \lhd E$
  ⟨*proof*⟩

lemma *some-third-chamberD-facet*:
  chamber $C \Longrightarrow z \lhd C \Longrightarrow z \lhd$ *some-third-chamber C D z*
  ⟨*proof*⟩

lemma *some-third-chamberD-simplex*:
  chamber $C \Longrightarrow z \lhd C \Longrightarrow$ *some-third-chamber C D z* $\in X$
  ⟨*proof*⟩

lemma *some-third-chamberD-adj*:
  chamber $C \Longrightarrow z \lhd C \Longrightarrow C \sim$ *some-third-chamber C D z*
  ⟨*proof*⟩

lemma *chamber-some-third-chamber*:
  chamber $C \Longrightarrow z \lhd C \Longrightarrow$ *chamber* (*some-third-chamber C D z*)
  ⟨*proof*⟩

lemma *some-third-chamberD-ne*:
  assumes *chamber C* $z \lhd C$
  shows *some-third-chamber C D z* $\neq C$ *some-third-chamber C D z* $\neq D$
  ⟨*proof*⟩

end

end

5 Coxeter systems and complexes

A Coxeter system is a group that affords a presentation, where each generator is of order two, and each relator is an alternating word of even length in two generators.

theory *Coxeter*
imports *Chamber*

begin

5.1 Coxeter-like systems

First we work in a group generated by elements of order two.

5.1.1 Locale definition and basic facts

locale *PreCoxeterSystem* =
  fixes *S* :: *'w*::*group-add set*
  assumes *genset-order2*: *s*∈*S* ⟹ *add-order s* = *2*
begin

abbreviation *W* ≡ ⟨*S*⟩
abbreviation *S-length* ≡ *word-length S*
abbreviation *S-reduced-for* ≡ *reduced-word-for S*
abbreviation *S-reduced* ≡ *reduced-word S*
abbreviation *relfun* ≡ λ*s t*. *add-order* (*s*+*t*)

lemma *no-zero-genset*: *0*∉*S*
⟨*proof*⟩

lemma *genset-order2-add*: *s*∈*S* ⟹ *s* + *s* = *0*
  ⟨*proof*⟩

lemmas *genset-uminus* = *minus-unique*[*OF genset-order2-add*]

lemma *relfun-S*: *s*∈*S* ⟹ *relfun s s* = *1*
  ⟨*proof*⟩

lemma *relfun-eq1*: ⟦ *s*∈*S*; *relfun s t* = *1* ⟧ ⟹ *t*=*s*
  ⟨*proof*⟩

lemma *S-relator-list*: *s*∈*S* ⟹ *pair-relator-list s s* = [*s*,*s*]
  ⟨*proof*⟩

lemma *S-sym*: *T* ⊆ *S* ⟹ *uminus* ‘ *T* ⊆ *T*
  ⟨*proof*⟩

lemmas *special-subgroup-eq-sum-list* =

genby-sym-eq-sum-lists[*OF S-sym*]
lemmas *genby-S-reduced-word-for-arg-min* =
reduced-word-for-genby-sym-arg-min[*OF S-sym*]
lemmas *in-genby-S-reduced-letter-set* =
in-genby-sym-imp-in-reduced-letter-set[*OF S-sym*]

end

5.1.2 Special cosets

From a Coxeter system we will eventually construct an associated chamber complex. To do so, we will consider the collection of special cosets: left cosets of subgroups generated by subsets of the generating set S. This collection forms a poset under the supset relation that, under a certain extra assumption, can be used to form a simplicial complex whose poset of simplices is isomorphic to this poset of special cosets. In the literature, groups generated by subsets of S are often referred to as parabolic subgroups of W, and their cosets as parabolic cosets, but following Garrett [2] we have opted for the names special subgroups and special cosets.

context *PreCoxeterSystem*
begin

definition *special-cosets* :: *'w set set*
where *special-cosets* ≡ (⋃ *T*∈*Pow S*. (⋃ *w*∈*W*. { *w* +*o* ⟨*T*⟩ }))
abbreviation 𝒫 ≡ *special-cosets*

lemma *special-cosetsI*: *T*∈*Pow S* ⟹ *w*∈*W* ⟹ *w* +*o* ⟨*T*⟩ ∈ 𝒫
⟨*proof*⟩

lemma *special-coset-singleton*: *w*∈*W* ⟹ {*w*}∈𝒫
⟨*proof*⟩

lemma *special-coset-nempty*: *X*∈𝒫 ⟹ *X* ≠ {}
⟨*proof*⟩

lemma *special-subgroup-special-coset*: *T*∈*Pow S* ⟹ ⟨*T*⟩ ∈ 𝒫
⟨*proof*⟩

lemma *special-cosets-lcoset-closed*: *w*∈*W* ⟹ *X*∈𝒫 ⟹ *w* +*o* *X* ∈ 𝒫
⟨*proof*⟩

lemma *special-cosets-lcoset-shift*: *w*∈*W* ⟹ ((+*o*) *w*) ' 𝒫 = 𝒫
⟨*proof*⟩

lemma *special-cosets-has-bottom*: *supset-has-bottom* 𝒫
⟨*proof*⟩

lemma *special-cosets-bottom*: *supset-bottom* 𝒫 = *W*

⟨*proof*⟩

end

5.1.3 Transfer from the free group over generators

We form a set of relators and show that it and S form a *GroupWithGeneratorsRelators*. The associated quotient group G maps surjectively onto W. In the *CoxeterSystem* locale below, this correspondence will be assumed to be injective as well.

context *PreCoxeterSystem*
begin

abbreviation R :: *'w list set* **where** $R \equiv (\bigcup s{\in}S.\ \bigcup t{\in}S.\ \{$*pair-relator-list s t*$\})$
abbreviation $P \equiv$ *map* (*charpair* S) ‘ R
abbreviation $P' \equiv$ *GroupWithGeneratorsRelators.P′ S R*
abbreviation $Q \equiv$ *GroupWithGeneratorsRelators.Q S R*
abbreviation $G \equiv$ *GroupWithGeneratorsRelators.G S R*
abbreviation *relator-freeword* ≡
GroupWithGeneratorsRelators.relator-freeword S
abbreviation *pair-relator-freeword* :: *'w ⇒ 'w ⇒ 'w freeword*
where *pair-relator-freeword s t* ≡ *Abs-freelist* (*pair-relator-list s t*)

abbreviation *freeliftid* ≡ *freeword-funlift id*

abbreviation *induced-id* :: *'w freeword set permutation ⇒ 'w*
where *induced-id* ≡ *GroupWithGeneratorsRelators.induced-id S R*

lemma *S-relator-freeword*: *s∈S* ⟹ *pair-relator-freeword s s* = *s*[+]*s*
⟨*proof*⟩

lemma *map-charpair-map-pairtrue-R*:
s∈S ⟹ *t∈S* ⟹
map (*charpair S*) (*pair-relator-list s t*) = *map pairtrue* (*pair-relator-list s t*)
⟨*proof*⟩

lemma *relator-freeword*:
s∈S ⟹ *t∈S* ⟹
pair-relator-freeword s t = *relator-freeword* (*pair-relator-list s t*)
⟨*proof*⟩

lemma *relator-freewords*: *Abs-freelist* ‘ $R = P'$
⟨*proof*⟩

lemma *GroupWithGeneratorsRelators-S-R*: *GroupWithGeneratorsRelators S R*
⟨*proof*⟩

lemmas *GroupByPresentation-S-P* =

GroupWithGeneratorsRelators.GroupByPresentation-S-P[
  OF GroupWithGeneratorsRelators-S-R
]

lemmas *Q-FreeS* = *GroupByPresentation.Q-FreeS*[*OF GroupByPresentation-S-P*]

lemma *relator-freeword-Q*: $s \in S \Longrightarrow t \in S \Longrightarrow$ *pair-relator-freeword s t* $\in Q$
⟨*proof*⟩

lemmas *P′-FreeS* =
GroupWithGeneratorsRelators.P′-FreeS[
  OF GroupWithGeneratorsRelators-S-R
]

lemmas *GroupByPresentationInducedFun-S-P-id* =
GroupWithGeneratorsRelators.GroupByPresentationInducedFun-S-P-id[
  OF GroupWithGeneratorsRelators-S-R
]

lemma *rconj-relator-freeword*:
⟦ $s \in S$; $t \in S$; *proper-signed-list xs*; *fst ‘ set xs* $\subseteq S$ ⟧ $\Longrightarrow$
  rconjby (*Abs-freeword xs*) (*pair-relator-freeword s t*) $\in Q$
⟨*proof*⟩

lemma *lconjby-Abs-freelist-relator-freeword*:
⟦ $s \in S$; $t \in S$; *xs*∈*lists S* ⟧ $\Longrightarrow$
  lconjby (*Abs-freelist xs*) (*pair-relator-freeword s t*) $\in Q$
⟨*proof*⟩

lemma *Abs-freelist-rev-append-alternating-list-in-Q*:
assumes $s \in S$ $t \in S$
shows *Abs-freelist* (*rev* (*alternating-list n s t*) @ *alternating-list n s t*) $\in Q$
⟨*proof*⟩

lemma *Abs-freeword-freelist-uminus-add-in-Q*:
proper-signed-list xs $\Longrightarrow$ *fst ‘ set xs* $\subseteq S \Longrightarrow$
  − *Abs-freelistfst xs* + *Abs-freeword xs* $\in Q$
⟨*proof*⟩

lemma *Q-freelist-freeword′*:
⟦ *proper-signed-list xs*; *fst ‘ set xs* $\subseteq S$; *Abs-freelistfst xs* $\in Q$ ⟧ $\Longrightarrow$
  Abs-freeword xs $\in Q$
⟨*proof*⟩

lemma *Q-freelist-freeword*:
$c \in$ *FreeGroup S* $\Longrightarrow$ *Abs-freelist* (*map fst* (*freeword c*)) $\in Q \Longrightarrow c \in Q$
⟨*proof*⟩

Here we show that the lift of the identity map to the free group on S is really just summation.

lemma *freeliftid-Abs-freeword-conv-sum-list*:
proper-signed-list xs $\Longrightarrow$ *fst ‘ set xs* $\subseteq$ *S* $\Longrightarrow$
freeliftid (*Abs-freeword xs*) = *sum-list* (*map fst xs*)
⟨*proof*⟩

end

5.1.4 Words in generators containing alternating subwords

Besides cancelling subwords equal to relators, the primary algebraic manipulation in seeking to reduce a word in generators in a Coxeter system is to reverse the order of alternating subwords of half the length of the associated relator, in order to create adjacent repeated letters that can be cancelled. Here we detail the mechanics of such manipulations.

context *PreCoxeterSystem*
begin

lemma *sum-list-pair-relator-halflist-flip*:
s$\in$*S* $\Longrightarrow$ *t*$\in$*S* $\Longrightarrow$
sum-list (*pair-relator-halflist s t*) = *sum-list* (*pair-relator-halflist t s*)
⟨*proof*⟩

definition *flip-altsublist-adjacent* :: *'w list* $\Rightarrow$ *'w list* $\Rightarrow$ *bool*
where *flip-altsublist-adjacent ss ts*
$\equiv$ $\exists$ *s t as bs*. *ss* = *as* @ (*pair-relator-halflist s t*) @ *bs* $\wedge$
ts = *as* @ (*pair-relator-halflist t s*) @ *bs*
abbreviation *flip-altsublist-chain* $\equiv$ *binrelchain flip-altsublist-adjacent*

lemma *flip-altsublist-adjacentI*:
ss = *as* @ (*pair-relator-halflist s t*) @ *bs* $\Longrightarrow$
ts = *as* @ (*pair-relator-halflist t s*) @ *bs* $\Longrightarrow$
flip-altsublist-adjacent ss ts
⟨*proof*⟩

lemma *flip-altsublist-adjacent-Cons-grow*:
assumes *flip-altsublist-adjacent ss ts*
shows *flip-altsublist-adjacent* (*a*#*ss*) (*a*#*ts*)
⟨*proof*⟩

lemma *flip-altsublist-chain-map-Cons-grow*:
flip-altsublist-chain tss $\Longrightarrow$ *flip-altsublist-chain* (*map* ((#) *t*) *tss*)
⟨*proof*⟩

lemma *flip-altsublist-adjacent-refl*:
ss $\neq$ [] $\Longrightarrow$ *ss*$\in$*lists S* $\Longrightarrow$ *flip-altsublist-adjacent ss ss*
⟨*proof*⟩

lemma *flip-altsublist-adjacent-sym*:
flip-altsublist-adjacent ss ts ⟹ *flip-altsublist-adjacent ts ss*
⟨*proof*⟩

lemma *rev-flip-altsublist-chain*:
flip-altsublist-chain xss ⟹ *flip-altsublist-chain* (*rev xss*)
⟨*proof*⟩

lemma *flip-altsublist-adjacent-set*:
assumes *ss*∈*lists S flip-altsublist-adjacent ss ts*
shows *set ts* = *set ss*
⟨*proof*⟩

lemma *flip-altsublist-adjacent-set-ball*:
∀ *ss*∈*lists S*. ∀ *ts*. *flip-altsublist-adjacent ss ts* ⟶ *set ts* = *set ss*
⟨*proof*⟩

lemma *flip-altsublist-adjacent-lists*:
ss ∈ *lists S* ⟹ *flip-altsublist-adjacent ss ts* ⟹ *ts* ∈ *lists S*
⟨*proof*⟩

lemma *flip-altsublist-adjacent-lists-ball*:
∀ *ss*∈*lists S*. ∀ *ts*. *flip-altsublist-adjacent ss ts* ⟶ *ts* ∈ *lists S*
⟨*proof*⟩

lemma *flip-altsublist-chain-lists*:
ss ∈ *lists S* ⟹ *flip-altsublist-chain* (*ss*#*xss*@[*ts*]) ⟹ *ts* ∈ *lists S*
⟨*proof*⟩

lemmas *flip-altsublist-chain-funcong-Cons-snoc* =
binrelchain-setfuncong-Cons-snoc[*OF flip-altsublist-adjacent-lists-ball*]

lemmas *flip-altsublist-chain-set* =
flip-altsublist-chain-funcong-Cons-snoc[
OF flip-altsublist-adjacent-set-ball
]

lemma *flip-altsublist-adjacent-length*:
flip-altsublist-adjacent ss ts ⟹ *length ts* = *length ss*
⟨*proof*⟩

lemmas *flip-altsublist-chain-length* =
binrelchain-funcong-Cons-snoc[
of flip-altsublist-adjacent length, *OF flip-altsublist-adjacent-length*, *simplified*
]

lemma *flip-altsublist-adjacent-sum-list*:
assumes *ss* ∈ *lists S flip-altsublist-adjacent ss ts*
shows *sum-list ts* = *sum-list ss*

⟨*proof*⟩

lemma *flip-altsublist-adjacent-sum-list-ball*:
$\forall$ *ss*$\in$*lists S*. $\forall$ *ts*. *flip-altsublist-adjacent ss ts* $\longrightarrow$ *sum-list ts* = *sum-list ss*
⟨*proof*⟩

lemma *S-reduced-forI-flip-altsublist-adjacent*:
S-reduced-for w ss $\Longrightarrow$ *flip-altsublist-adjacent ss ts* $\Longrightarrow$ *S-reduced-for w ts*
⟨*proof*⟩

lemma *flip-altsublist-adjacent-in-Q'*:
fixes *as bs s t*
defines *xs*: *xs* $\equiv$ *as* @ *pair-relator-halflist s t* @ *bs*
and *ys*: *ys* $\equiv$ *as* @ *pair-relator-halflist t s* @ *bs*
assumes *Axs*: *Abs-freelist xs* $\in$ *Q*
shows *Abs-freelist ys* $\in$ *Q*
⟨*proof*⟩

lemma *flip-altsublist-adjacent-in-Q*:
Abs-freelist ss $\in$ *Q* $\Longrightarrow$ *flip-altsublist-adjacent ss ts* $\Longrightarrow$ *Abs-freelist ts* $\in$ *Q*
⟨*proof*⟩

lemma *flip-altsublist-chain-G-in-Q*:
⟦ *Abs-freelist ss* $\in$ *Q*; *flip-altsublist-chain* (*ss*#*xss*@[*ts*]) ⟧ $\Longrightarrow$ *Abs-freelist ts* $\in$ *Q*
⟨*proof*⟩

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*⟩

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 n s t* @ *cs*)
alternating-list n s t @ *cs* = *xs*@[*x*]@*ys S-reduced-for w* (*t*#*xs*@*ys*)
shows *length xs* $\geq$ *n*
⟨*proof*⟩

end

5.1.5 Preliminary facts on the word problem

The word problem seeks criteria for determining whether two words over the generator set represent the same element in *W*. Here we establish one direction of the word problem, as well as a preliminary step toward the other direction.

context *PreCoxeterSystem*
begin

lemmas *flip-altsublist-chain-sum-list* =
flip-altsublist-chain-funcong-Cons-snoc[*OF flip-altsublist-adjacent-sum-list-ball*]
— This lemma represents one direction in the word problem: if a word in generators can be transformed into another by a sequence of manipulations, each of which consists of replacing a half-relator subword by its reversal, then the two words sum to the same element of *W*.

lemma *reduced-word-problem-eq-hd-step*:
assumes *step*: $\bigwedge y\ ss\ ts.$ ⟦
S-length y < *S-length w*; *y*≠*0*; *S-reduced-for y ss*; *S-reduced-for y ts*
⟧ ⟹ ∃ *xss*. *flip-altsublist-chain* (*ss* # *xss* @ [*ts*])
and *set-up*: *S-reduced-for w* (*a*#*ss*) *S-reduced-for w* (*a*#*ts*)
shows ∃ *xss*. *flip-altsublist-chain* ((*a*#*ss*) # *xss* @ [*a*#*ts*])
⟨*proof*⟩

end

5.1.6 Preliminary facts related to the deletion condition

The deletion condition states that in a Coxeter system, every non-reduced word in the generating set can be shortened to an equivalent word by deleting some particular pair of letters. This condition is both necessary and sufficient for a group generated by elements of order two to be a Coxeter system. Here we establish some facts related to the deletion condition that are true in any group generated by elements of order two.

context *PreCoxeterSystem*
begin

abbreviation $\mathcal{H}$ ≡ (⋃*w*∈*W*. *lconjby w* ' *S*) — the set of reflections

abbreviation *lift-signed-lconjperm* ≡ *freeword-funlift signed-lconjpermutation*

lemma *lconjseq-reflections*: *ss*∈*lists S* ⟹ *set* (*lconjseq ss*) ⊆ $\mathcal{H}$
⟨*proof*⟩

lemma *deletion'*:
ss ∈ *lists S* ⟹ ¬ *distinct* (*lconjseq ss*) ⟹
∃ *a b as bs cs*. *ss* = *as* @ [*a*] @ *bs* @ [*b*] @ *cs* ∧
sum-list ss = *sum-list* (*as*@*bs*@*cs*)
⟨*proof*⟩

lemma *S-reduced-imp-distinct-lconjseq'*:
assumes *ss* ∈ *lists S* ¬ *distinct* (*lconjseq ss*)
shows ¬ *S-reduced ss*
⟨*proof*⟩

lemma *S-reduced-imp-distinct-lconjseq*: *S-reduced ss* ⟹ *distinct* (*lconjseq ss*)

⟨*proof*⟩

lemma *permutation-lift-signed-lconjperm-eq-signed-list-lconjaction′*:
proper-signed-list xs ⟹ *fst* ‘ *set xs* ⊆ *S* ⟹
permutation (*lift-signed-lconjperm* (*Abs-freeword xs*)) =
signed-list-lconjaction (*map fst xs*)
⟨*proof*⟩

lemma *permutation-lift-signed-lconjperm-eq-signed-list-lconjaction*:
x ∈ *FreeGroup S* ⟹
permutation (*lift-signed-lconjperm x*) =
signed-list-lconjaction (*map fst* (*freeword x*))
⟨*proof*⟩

lemma *even-count-lconjseq-rev-relator*:
s∈*S* ⟹ *t*∈*S* ⟹ *even* (*count-list* (*lconjseq* (*rev* (*pair-relator-list s t*))) *x*)
⟨*proof*⟩

lemma *GroupByPresentationInducedFun-S-R-signed-lconjaction*:
GroupByPresentationInducedFun S P signed-lconjpermutation
⟨*proof*⟩

end

5.2 Coxeter-like systems with deletion

Here we add the so-called deletion condition as an assumption, and explore its consequences.

5.2.1 Locale definition

locale *PreCoxeterSystemWithDeletion* = *PreCoxeterSystem S*
for *S* :: ′*w*::*group-add set*
\+ **assumes** *deletion*:
ss ∈ *lists S* ⟹ ¬ *reduced-word S ss* ⟹
∃ *a b as bs cs*. *ss* = *as* @ [*a*] @ *bs* @ [*b*] @ *cs* ∧
sum-list ss = *sum-list* (*as*@*bs*@*cs*)

5.2.2 Consequences of the deletion condition

context *PreCoxeterSystemWithDeletion*
begin

lemma *deletion-reduce*:
ss ∈ *lists S* ⟹ ∃ *ts*. *ts* ∈ *ssubseqs ss* ∩ *reduced-words-for S* (*sum-list ss*)
⟨*proof*⟩

lemma *deletion-reduce′*:
ss ∈ *lists S* ⟹ ∃ *ts*∈*reduced-words-for S* (*sum-list ss*). *set ts* ⊆ *set ss*

⟨*proof*⟩

end

5.2.3 The exchange condition

The exchange condition states that, given a reduced word in the generators, if prepending a letter to the word does not remain reduced, then the new word can be shortened to a word equivalent to the original one by deleting some letter other than the prepended one. Thus, one able to exchange some letter for the addition of a desired letter at the beginning of a word, without changing the elemented represented.

context *PreCoxeterSystemWithDeletion*
begin

lemma *exchange*:
  assumes $s \in S$ *S-reduced-for w ss* $\neg$ *S-reduced* $(s\#ss)$
  shows $\exists\, t\ as\ bs.\ ss = as@t\#bs \wedge$ *reduced-word-for S w* $(s\#as@bs)$
⟨*proof*⟩

lemma *reduced-head-imp-exchange*:
  assumes *reduced-word-for S w* $(s\#as)$ *reduced-word-for S w cs*
  shows $\exists\, a\ ds\ es.\ cs = ds@[a]@es \wedge$ *reduced-word-for S w* $(s\#ds@es)$
⟨*proof*⟩

end

5.2.4 More on words in generators containing alternating subwords

Here we explore more of the mechanics of manipulating words over S that contain alternating subwords, in preparation of the word problem.

context *PreCoxeterSystemWithDeletion*
begin

lemma *two-reduced-heads-imp-reduced-alt-step*:
  assumes $s \neq t$ *reduced-word-for S w* $(t\#bs)$ $n <$ *relfun s t* $\vee$ *relfun s t* $= 0$
    reduced-word-for S w (*alternating-list n s t* @ *cs*)
  shows $\exists\, ds.$ *reduced-word-for S w* (*alternating-list* (*Suc n*) *t s* @ *ds*)
⟨*proof*⟩

lemma *two-reduced-heads-imp-reduced-alt'*:
  assumes $s \neq t$ *reduced-word-for S w* $(s\#as)$ *reduced-word-for S w* $(t\#bs)$
  shows $n \leq$ *relfun s t* $\vee$ *relfun s t* $= 0 \Longrightarrow (\exists\, cs.$
    reduced-word-for S w (*alternating-list n s t* @ *cs*) $\vee$
    reduced-word-for S w (*alternating-list n t s* @ *cs*)
)

⟨*proof*⟩

lemma *two-reduced-heads-imp-reduced-alt*:
assumes $s \neq t$ *reduced-word-for S w* ($s\#as$) *reduced-word-for S w* ($t\#bs$)
shows $\exists$ *cs. reduced-word-for S w* (*pair-relator-halflist s t* @ *cs*)
⟨*proof*⟩

lemma *two-reduced-heads-imp-nzero-relfun*:
assumes $s \neq t$ *reduced-word-for S w* ($s\#as$) *reduced-word-for S w* ($t\#bs$)
shows *relfun s t* $\neq$ *0*
⟨*proof*⟩

end

5.2.5 The word problem

Here we establish the other direction of the word problem for reduced words.

context *PreCoxeterSystemWithDeletion*
begin

lemma *reduced-word-problem-ConsCons-step*:
assumes $\bigwedge$*y ss ts.* ⟦ *S-length y* < *S-length w*; $y \neq 0$; *reduced-word-for S y ss*;
reduced-word-for S y ts ⟧ $\Longrightarrow$ $\exists$ *xss. flip-altsublist-chain* (*ss* # *xss* @ [*ts*])
reduced-word-for S w ($a\#as$) *reduced-word-for S w* ($b\#bs$) $a \neq b$
shows $\exists$ *xss. flip-altsublist-chain* (($a\#as$)#*xss*@[$b\#bs$])
⟨*proof*⟩

lemma *reduced-word-problem*:
⟦ $w \neq 0$; *reduced-word-for S w ss*; *reduced-word-for S w ts* ⟧ $\Longrightarrow$
$\exists$ *xss. flip-altsublist-chain* (*ss*#*xss*@[*ts*])
⟨*proof*⟩

lemma *reduced-word-letter-set*:
assumes *S-reduced-for w ss*
shows *reduced-letter-set S w* = *set ss*
⟨*proof*⟩

end

5.2.6 Special subgroups and cosets

Recall that special subgroups are those generated by subsets of the generating set S. Here we show that the presence of the deletion condition guarantees that the collection of special subgroups and their left cosets forms a poset under reverse inclusion that satisfies the necessary properties to ensure that the poset of simplices in the associated simplicial complex is isomorphic to this poset of special cosets.

context *PreCoxeterSystemWithDeletion*

begin

lemma *special-subgroup-int-S*:
assumes $T \in Pow\ S$
shows $\langle T\rangle \cap S = T$
⟨*proof*⟩

lemma *special-subgroup-inj*: *inj-on genby* (*Pow S*)
⟨*proof*⟩

lemma *special-subgroup-genby-subset-ordering-iso*:
subset-ordering-iso (*Pow S*) *genby*
⟨*proof*⟩

lemmas *special-subgroup-genby-rev-mono*
= *OrderingSetIso.rev-ordsetmap*[*OF special-subgroup-genby-subset-ordering-iso*]

lemma *special-subgroup-word-length*:
assumes $T \in Pow\ S$ $w \in \langle T\rangle$
shows *word-length T w* = *S-length w*
⟨*proof*⟩

lemma *S-subset-reduced-imp-S-reduced*:
$T \in Pow\ S \Longrightarrow$ *reduced-word T ts* $\Longrightarrow$ *S-reduced ts*
⟨*proof*⟩

lemma *smallest-genby*: $T \in Pow\ S \Longrightarrow w \in \langle T\rangle \Longrightarrow$ *reduced-letter-set S w* $\subseteq T$
⟨*proof*⟩

lemma *special-cosets-below-in*:
assumes $w \in W$ $T \in Pow\ S$
shows $\mathcal{P}.\supseteq(w +o \langle T\rangle) = (\bigcup R\in(Pow\ S).\supseteq T.\ \{w +o \langle R\rangle\})$
⟨*proof*⟩

lemmas *special-coset-inj*
= *comp-inj-on*[*OF special-subgroup-inj*, *OF inj-inj-on*, *OF lcoset-inj-on*]

lemma *special-coset-eq-imp-eq-gensets*:
$[\![\ T1 \in Pow\ S;\ T2 \in Pow\ S;\ w1\ +o\ \langle T1\rangle = w2\ +o\ \langle T2\rangle\ ]\!] \Longrightarrow T1 = T2$
⟨*proof*⟩

lemma *special-subgroup-special-coset-subset-ordering-iso*:
subset-ordering-iso (*genby* ‘ *Pow S*) ((+*o*) *w*)
⟨*proof*⟩

lemma *special-coset-subset-ordering-iso*:
subset-ordering-iso (*Pow S*) ((+*o*) *w* ∘ *genby*)
⟨*proof*⟩

lemmas *special-coset-subset-rev-mono* =
OrderingSetIso.rev-ordsetmap[*OF special-coset-subset-ordering-iso*]

lemma *special-coset-below-in-subset-ordering-iso*:
subset-ordering-iso ((*Pow S*).$\supseteq T$) ((+*o*) *w* ∘ *genby*)
⟨*proof*⟩

lemma *special-coset-below-in-supset-ordering-iso*:
OrderingSetIso (⊇) (⊃) (⊇) (⊃) ((*Pow S*).$\supseteq T$) ((+*o*) *w* ∘ *genby*)
⟨*proof*⟩

lemma *special-coset-pseudominimals*:
assumes *supset-pseudominimal-in* $\mathcal{P}$ *X*
shows $\exists w\ s.\ w \in W \wedge s \in S \wedge X = w +o \langle S - \{s\} \rangle$
⟨*proof*⟩

lemma *special-coset-pseudominimal-in-below-in*:
assumes $w \in W$ $T \in Pow\ S$ *supset-pseudominimal-in* ($\mathcal{P}$.$\supseteq$(w +*o* ⟨T⟩)) *X*
shows $\exists s \in S - T.\ X = w +o \langle S - \{s\} \rangle$
⟨*proof*⟩

lemma *exclude-one-is-pseudominimal*:
assumes $w \in W$ $t \in S$
shows *supset-pseudominimal-in* $\mathcal{P}$ (w +*o* ⟨$S - \{t\}$⟩)
⟨*proof*⟩

lemma *exclude-one-is-pseudominimal-in-below-in*:
⟦ $w \in W$; $T \in Pow\ S$; $s \in S - T$ ⟧ ⟹
supset-pseudominimal-in ($\mathcal{P}$.$\supseteq$(w +*o* ⟨T⟩)) (w +*o* ⟨$S - \{s\}$⟩)
⟨*proof*⟩

lemma *glb-special-subset-coset*:
assumes wTT': $w \in W$ $T \in Pow\ S$ $T' \in Pow\ S$
defines U: $U \equiv T \cup T' \cup$ *reduced-letter-set S w*
shows *supset-glbound-in-of* $\mathcal{P}$ ⟨T⟩ (w +*o* ⟨T'⟩) ⟨U⟩
⟨*proof*⟩

lemma *glb-special-subset-coset-ex*:
assumes $w \in W$ $T \in Pow\ S$ $T' \in Pow\ S$
shows $\exists B.$ *supset-glbound-in-of* $\mathcal{P}$ ⟨T⟩ (w +*o* ⟨T'⟩) *B*
⟨*proof*⟩

lemma *special-cosets-have-glbs*:
assumes $X \in \mathcal{P}$ $Y \in \mathcal{P}$
shows $\exists B.$ *supset-glbound-in-of* $\mathcal{P}$ *X Y B*
⟨*proof*⟩

end

5.3 Coxeter systems

5.3.1 Locale definition and transfer from the associated free group

Now we consider groups generated by elements of order two with an additional assumption to ensure that the natural correspondence between the group W and the group presentation on the generating set S and its relations is bijective. Below, such groups will be shown to satisfy the deletion condition.

locale *CoxeterSystem* = *PreCoxeterSystem S*
  for *S* :: *'w::group-add set*
+ **assumes** *induced-id-inj*: *inj-on induced-id G*

lemma (**in** *PreCoxeterSystem*) *CoxeterSystemI*:
  assumes $\bigwedge g.\ g \in G \Longrightarrow \textit{induced-id}\ g = 0 \Longrightarrow g=0$
  shows *CoxeterSystem S*
⟨*proof*⟩

context *CoxeterSystem*
begin

abbreviation *inv-induced-id* ≡ *GroupPresentation.inv-induced-id S R*

lemma *GroupPresentation-S-R*: *GroupPresentation S R*
  ⟨*proof*⟩

lemmas *inv-induced-id-sum-list* =
  GroupPresentation.inv-induced-id-sum-list-S[*OF GroupPresentation-S-R*]

end

5.3.2 The deletion condition is necessary

Call an element of W a reflection if it is a conjugate of a generating element (and so is also of order two). Here we use the action of words over S on such reflections to show that Coxeter systems satisfy the deletion condition.

context *CoxeterSystem*
begin

abbreviation *induced-signed-lconjperm* ≡
  GroupByPresentationInducedFun.induced-hom S P signed-lconjpermutation

definition *flipped-reflections* :: *'w* ⇒ *'w set*
  where *flipped-reflections w* ≡
    {$t \in \mathcal{H}$. *induced-signed-lconjperm* (*inv-induced-id* $(-w)$) →
      (*t*, *True*) = (*rconjby w t*, *False*)}

lemma *induced-signed-lconjperm-inv-induced-id-sum-list*:

ss ∈ *lists S* ⟹ *induced-signed-lconjperm* (*inv-induced-id* (*sum-list ss*)) =
sum-list (*map signed-lconjpermutation ss*)
⟨*proof*⟩

lemma *induced-signed-eq-lconjpermutation*:
ss ∈ *lists S* ⟹
permutation (*induced-signed-lconjperm* (*inv-induced-id* (*sum-list ss*))) =
signed-list-lconjaction ss
⟨*proof*⟩

lemma *flipped-reflections-odd-lconjseq*:
assumes *ss*∈*lists S*
shows *flipped-reflections* (*sum-list ss*) = {*t*∈$\mathcal{H}$. *odd* (*count-list* (*lconjseq ss*) *t*)}
⟨*proof*⟩

lemma *flipped-reflections-in-lconjseq*:
ss∈*lists S* ⟹ *flipped-reflections* (*sum-list ss*) ⊆ *set* (*lconjseq ss*)
⟨*proof*⟩

lemma *flipped-reflections-distinct-lconjseq-eq-lconjseq*:
assumes *ss*∈*lists S distinct* (*lconjseq ss*)
shows *flipped-reflections* (*sum-list ss*) = *set* (*lconjseq ss*)
⟨*proof*⟩

lemma *flipped-reflections-reduced-eq-lconjseq*:
S-reduced ss ⟹ *flipped-reflections* (*sum-list ss*) = *set* (*lconjseq ss*)
⟨*proof*⟩

lemma *card-flipped-reflections*:
assumes *w*∈*W*
shows *card* (*flipped-reflections w*) = *S-length w*
⟨*proof*⟩

end

sublocale *CoxeterSystem* < *PreCoxeterSystemWithDeletion*
⟨*proof*⟩

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 ∈ *lists S* ⟹ ¬ *S-reduced ss* ⟹
∃ *xss as t bs*. *flip-altsublist-chain* (*ss* # *xss* @ [*as*@[*t*,*t*]@*bs*])

⟨*proof*⟩

lemma *freeliftid-kernel′*:
ss ∈ *lists S* ⟹ *sum-list ss* = *0* ⟹ *Abs-freelist ss* ∈ *Q*
⟨*proof*⟩

lemma *freeliftid-kernel*:
assumes *c* ∈ *FreeGroup S freeliftid c* = *0*
shows *c*∈*Q*
⟨*proof*⟩

lemma *induced-id-kernel*:
c ∈ *FreeGroup S* ⟹ *induced-id* (⌈*FreeGroup S*|*c*|*Q*⌉) = *0* ⟹ *c*∈*Q*
⟨*proof*⟩

theorem *CoxeterSystem*: *CoxeterSystem S*
⟨*proof*⟩

end

5.3.4 The Coxeter system associated to a thin chamber complex with many foldings

We now show that the fundamental automorphisms in a thin chamber complex with many foldings satisfy the deletion condition, and hence form a Coxeter system.

context *ThinChamberComplexManyFoldings*
begin

lemma *not-reduced-word-not-min-gallery*:
assumes *ss* ∈ *lists S* ¬ *reduced-word S ss*
shows ¬ *min-gallery* (*map* (λ*w*. *w*‘→*C0*) (*sums ss*))
⟨*proof*⟩

lemma *S-list-not-min-gallery-double-split*:
assumes *ss* ∈ *lists S ss*≠[] ¬ *min-gallery* (*map* (λ*w*. *w*‘→*C0*) (*sums ss*))
shows
∃*f g as s bs t cs*.
(*f*,*g*)∈*foldpairs* ∧
sum-list as ‘→ *C0* ∈ *f*⊢𝒞 ∧
sum-list (*as*@[*s*]) ‘→ *C0* ∈ *g*⊢𝒞 ∧
sum-list (*as*@[*s*]@*bs*) ‘→ *C0* ∈ *g*⊢𝒞 ∧
sum-list (*as*@[*s*]@*bs*@[*t*]) ‘→ *C0* ∈ *f*⊢𝒞 ∧
ss = *as*@[*s*]@*bs*@[*t*]@*cs*
⟨*proof*⟩

lemma *fold-end-sum-chain-fg*:
fixes *f g* :: ′*a*⇒′*a*

defines s : s ≡ *induced-automorph f g*
assumes *fg* : $(f,g) \in$ *foldpairs*
and *as* : *as* ∈ *lists S*
and *s* : *s*∈*S*
and *sep*: *sum-list as* ‘→ *C0* ∈ *f*⊢*𝒞 sum-list* (*as*@[*s*]) ‘→ *C0* ∈ *g*⊢*𝒞*
shows *bs* ∈*lists S* ⟹
s ‘ *sum-list* (*as*@[*s*]@*bs*) ‘→ *C0* = *sum-list* (*as*@*bs*) ‘→ *C0*
⟨*proof*⟩

lemma *fold-end-sum-chain-gf*:
fixes *f g* :: ′*a*⇒′*a*
defines s ≡ *induced-automorph f g*
assumes *fg* : $(f,g) \in$ *foldpairs*
and *as* ∈ *lists S s*∈*S bs* ∈*lists S*
sum-list as ‘→ *C0* ∈ *g*⊢*𝒞*
sum-list (*as*@[*s*]) ‘→ *C0* ∈ *f*⊢*𝒞*
shows s ‘ *sum-list* (*as*@[*s*]@*bs*) ‘→ *C0* = *sum-list* (*as*@*bs*) ‘→ *C0*
⟨*proof*⟩

lemma *fold-middle-sum-chain*:
assumes *fg* : $(f,g) \in$ *foldpairs*
and *S* : *as* ∈ *lists S s*∈*S bs* ∈ *lists S t*∈*S cs* ∈*lists S*
and *sep*: *sum-list as* ‘→ *C0* ∈ *f*⊢*𝒞*
sum-list (*as*@[*s*]) ‘→ *C0* ∈ *g*⊢*𝒞*
sum-list (*as*@[*s*]@*bs*) ‘→ *C0* ∈ *g*⊢*𝒞 sum-list* (*as*@[*s*]@*bs*@[*t*]) ‘→ *C0*
∈ *f*⊢*𝒞*
shows *sum-list* (*as*@[*s*]@*bs*@[*t*]@*cs*) ‘→ *C0* = *sum-list* (*as*@*bs*@*cs*) ‘→ *C0*
⟨*proof*⟩

lemma *S-list-not-min-gallery-deletion*:
fixes *ss* :: ′*a permutation list*
defines *w* : *w* ≡ *sum-list ss*
assumes *ss*: *ss*∈*lists S ss*≠[] ¬ *min-gallery* (*map* (λ*w*. *w*‘→*C0*) (*sums ss*))
shows ∃ *a b as bs cs*. *ss* = *as*@[*a*]@*bs*@[*b*]@*cs* ∧ *w* = *sum-list* (*as*@*bs*@*cs*)
⟨*proof*⟩

lemma *deletion*:
ss ∈ *lists S* ⟹ ¬ *reduced-word S ss* ⟹
∃ *a b as bs cs*. *ss* = *as*@[*a*]@*bs*@[*b*]@*cs* ∧ *sum-list ss* = *sum-list* (*as*@*bs*@*cs*)
⟨*proof*⟩

lemma *PreCoxeterSystemWithDeletion*: *PreCoxeterSystemWithDeletion S*
⟨*proof*⟩

lemma *CoxeterSystem*: *CoxeterSystem S*
⟨*proof*⟩

end

5.4 Coxeter complexes

5.4.1 Locale and complex definitions

Now we add in the assumption that the generating set is finite, and construct the associated Coxeter complex from the poset of special cosets.

locale *CoxeterComplex* = *CoxeterSystem S*
  for *S* :: *'w::group-add set*
+ **assumes** *finite-genset*: *finite S*
begin

definition *TheComplex* :: *'w set set set*
  where *TheComplex* ≡ *ordering.PosetComplex* (⊇) (⊃) $\mathcal{P}$
abbreviation Σ ≡ *TheComplex*

end

5.4.2 As a simplicial complex

Here we record the fact that the Coxeter complex associated to a Coxeter system is a simplicial complex, and note that the poset of special cosets is complex-like. This last fact allows us to reason about the complex by reasoning about the poset, via the poset isomorphism *ComplexLikePoset.smap*.

context *CoxeterComplex*
begin

lemma *simplex-like-special-cosets*:
  assumes $X \in \mathcal{P}$
  shows *supset-simplex-like* ($\mathcal{P}$.⊇X)
⟨*proof*⟩

lemma *SimplicialComplex-Σ*: *SimplicialComplex* Σ
  ⟨*proof*⟩

lemma *ComplexLikePoset-special-cosets*: *ComplexLikePoset* (⊇) (⊃) $\mathcal{P}$
  ⟨*proof*⟩

abbreviation *smap* ≡ *ordering.poset-simplex-map* (⊇) (⊃) $\mathcal{P}$

lemmas *smap-def* = *ordering.poset-simplex-map-def*[*OF supset-poset*, *of* $\mathcal{P}$]

lemma *ordsetmap-smap*: ⟦ $X \in \mathcal{P}$; $Y \in \mathcal{P}$; $X \supseteq Y$ ⟧ ⟹ *smap X* ⊆ *smap Y*
  ⟨*proof*⟩

lemma *rev-ordsetmap-smap*: ⟦ $X \in \mathcal{P}$; $Y \in \mathcal{P}$; *smap X* ⊆ *smap Y* ⟧ ⟹ $X \supseteq Y$
  ⟨*proof*⟩

lemma *smap-onto-PosetComplex*: *smap* ‘ $\mathcal{P}$ = Σ

⟨*proof*⟩

lemmas *simplices-conv-special-cosets* = *smap-onto-PosetComplex*[*THEN sym*]

lemma *smap-into-PosetComplex*: $X \in \mathcal{P} \Longrightarrow smap\ X \in \Sigma$
⟨*proof*⟩

lemma *smap-pseudominimal*:
$w \in W \Longrightarrow s \in S \Longrightarrow smap\ (w +o \langle S-\{s\}\rangle) = \{w +o \langle S-\{s\}\rangle\}$
⟨*proof*⟩

lemma *exclude-one-notin-smap-singleton*:
$s \in S \Longrightarrow w +o \langle S-\{s\}\rangle \notin smap\ (w +o \langle\{s\}\rangle)$
⟨*proof*⟩

lemma *maxsimp-vertices*: $w \in W \Longrightarrow s \in S \Longrightarrow w +o \langle S-\{s\}\rangle \in smap\ \{w\}$
⟨*proof*⟩

lemma *maxsimp-singleton*:
assumes $w \in W$
shows *SimplicialComplex.maxsimp* Σ (*smap* $\{w\}$)
⟨*proof*⟩

lemma *maxsimp-is-singleton*:
assumes *SimplicialComplex.maxsimp* $\Sigma\ x$
shows $\exists w \in W.\ smap\ \{w\} = x$
⟨*proof*⟩

lemma *maxsimp-vertex-conv-special-coset*:
$w \in W \Longrightarrow X \in smap\ \{w\} \Longrightarrow \exists s \in S.\ X = w +o \langle S-\{s\}\rangle$
⟨*proof*⟩

lemma *vertices*: $w \in W \Longrightarrow s \in S \Longrightarrow w +o \langle S-\{s\}\rangle \in \bigcup \Sigma$
⟨*proof*⟩

lemma *smap0-conv-special-subgroups*:
$smap\ 0 = (\lambda s.\ \langle S - \{s\}\rangle)$ ‘ S
⟨*proof*⟩

lemma *S-bij-betw-chamber0*: *bij-betw* $(\lambda s.\ \langle S-\{s\}\rangle)\ S\ (smap\ 0)$
⟨*proof*⟩

lemma *smap-singleton-conv-W-image*:
$w \in W \Longrightarrow smap\ \{w\} = ((+o)\ w)$ ‘ $(smap\ 0)$
⟨*proof*⟩

lemma *W-lcoset-bij-betw-singletons*:
assumes $w \in W$
shows *bij-betw* $((+o)\ w)\ (smap\ 0)\ (smap\ \{w\})$

⟨*proof*⟩

lemma *facets*:
assumes *w*∈ *W* *s*∈*S*
shows *smap* (*w* +*o* ⟨{*s*}⟩) ◁ *smap* {*w*}
⟨*proof*⟩

lemma *facets'*: *w*∈ *W* ⟹ *s*∈*S* ⟹ *smap* {*w*,*w*+*s*} ◁ *smap* {*w*}
⟨*proof*⟩

lemma *adjacent*: *w*∈ *W* ⟹ *s*∈*S* ⟹ *smap* {*w*+*s*} ∼ *smap* {*w*}
⟨*proof*⟩

lemma *singleton-adjacent-0*: *s*∈*S* ⟹ *smap* {*s*} ∼ *smap* *0*
⟨*proof*⟩

end

5.4.3 As a chamber complex

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

context *CoxeterComplex*
begin

abbreviation *chamber* ≡ *SimplicialComplex.maxsimp* Σ
abbreviation *gallery* ≡ *SimplicialComplex.maxsimpchain* Σ

lemmas *chamber-singleton* = *maxsimp-singleton*
lemmas *chamber-vertex-conv-special-coset* = *maxsimp-vertex-conv-special-coset*
lemmas *chamber-vertices* = *maxsimp-vertices*
lemmas *chamber-is-singleton* = *maxsimp-is-singleton*

lemmas *faces* = *SimplicialComplex.faces* [*OF SimplicialComplex-*Σ]
lemmas *gallery-def* = *SimplicialComplex.maxsimpchain-def* [*OF SimplicialComplex-*Σ]
lemmas *gallery-rev* = *SimplicialComplex.maxsimpchain-rev* [*OF SimplicialComplex-*Σ]

lemmas *chamberD-simplex* =
SimplicialComplex.maxsimpD-simplex[*OF SimplicialComplex-*Σ]
lemmas *gallery-CConsI* =
SimplicialComplex.maxsimpchain-CConsI[*OF SimplicialComplex-*Σ]

lemmas *gallery-overlap-join* =
SimplicialComplex.maxsimpchain-overlap-join[*OF SimplicialComplex-*Σ]

lemma *word-gallery-to-0*:
ss ≠ [] ⟹ *ss*∈ *lists S* ⟹ ∃ *xs*. *gallery* (*smap* {*sum-list ss*} # *xs* @ [*smap 0*])
⟨*proof*⟩

lemma *gallery-to-0*:
  assumes $w \in W$ $w \neq 0$
  shows $\exists xs.$ *gallery* (*smap* $\{w\}$ # *xs* @ [*smap* *0*])
⟨*proof*⟩

lemma *ChamberComplex-Σ*: *ChamberComplex* Σ
⟨*proof*⟩

lemma *card-chamber*: *chamber* $x \Longrightarrow$ *card* x = *card* S
  ⟨*proof*⟩

lemma *vertex-conv-special-coset*:
  $X \in \bigcup \Sigma \Longrightarrow \exists\, w\ s.\ w \in W \wedge s \in S \wedge X = w +o \langle S - \{s\} \rangle$
  ⟨*proof*⟩

end

5.4.4 The Coxeter complex associated to a thin chamber complex with many foldings

Having previously verified that the fundamental automorphisms in a thin chamber complex with many foldings form a Coxeter system, we now record the existence of a chamber complex isomorphism onto the associated Coxeter complex.

context *ThinChamberComplexManyFoldings*
begin

lemma *CoxeterComplex*: *CoxeterComplex* *S*
  ⟨*proof*⟩

abbreviation Σ ≡ *CoxeterComplex.TheComplex* *S*

lemma *S-list-not-min-gallery-not-reduced*:
  assumes *ss*≠[] ¬ *min-gallery* (*map* (λ*w*. *w*‘→*C0*) (*sums* *ss*))
  shows ¬ *reduced-word* *S* *ss*
⟨*proof*⟩

lemma *reduced-S-list-min-gallery*:
  ss≠[] ⟹ *reduced-word* *S* *ss* ⟹ *min-gallery* (*map* (λ*w*. *w*‘→*C0*) (*sums* *ss*))
  ⟨*proof*⟩

lemma *fundchamber-vertex-stabilizer1*:
  fixes *t*
  defines *v*: *v* ≡ *fundantivertex* *t*
  assumes *tw*: *t*∈*S* *w*∈*W* *w*→*v* = *v*
  shows $w \in \langle S - \{t\} \rangle$
⟨*proof*⟩

lemma *fundchamber-vertex-stabilizer2*:
assumes *s*: $s \in S$
defines *v*: $v \equiv$ *fundantivertex s*
shows $w \in \langle S - \{s\} \rangle \Longrightarrow w \rightarrow v = v$
⟨*proof*⟩

lemma *label-wrt-special-coset1*:
assumes *label-wrt C0* φ *fixespointwise* φ *C0* $w0 \in W$ $s \in S$
defines $v \equiv$ *fundantivertex s*
shows $\{w \in W.\ w \rightarrow \varphi\ (w0 \rightarrow v) = w0 \rightarrow v\} = w0\ +o\ \langle S - \{s\} \rangle$
⟨*proof*⟩

lemma *label-wrt-special-coset1'*:
assumes *label-wrt C0* φ *fixespointwise* φ *C0* $w0 \in W$ $v \in C0$
defines $s \equiv$ *fundantipermutation v*
shows $\{w \in W.\ w \rightarrow \varphi\ (w0 \rightarrow v) = w0 \rightarrow v\} = w0\ +o\ \langle S - \{s\} \rangle$
⟨*proof*⟩

lemma *label-wrt-special-coset2'*:
assumes *label-wrt C0* φ *fixespointwise* φ *C0* $w0 \in W$ $v \in w0$‘$\rightarrow C0$
defines $s \equiv$ *fundantipermutation* $(\varphi\ v)$
shows $\{w \in W.\ w \rightarrow \varphi\ v = v\} = w0\ +o\ \langle S - \{s\} \rangle$
⟨*proof*⟩

lemma *label-stab-map-W-fundchamber-image*:
assumes *label-wrt C0* φ *fixespointwise* φ *C0* $w0 \in W$
defines $\psi \equiv \lambda v.\ \{w \in W.\ w \rightarrow (\varphi\ v) = v\}$
shows ψ‘$(w0$‘$\rightarrow C0)$ = *CoxeterComplex.smap S* $\{w0\}$
⟨*proof*⟩

lemma *label-stab-map-chamber-map*:
assumes φ: *label-wrt C0* φ *fixespointwise* φ *C0*
and *C*: *chamber C*
defines ψ: $\psi \equiv \lambda v.\ \{w \in W.\ w \rightarrow (\varphi\ v) = v\}$
shows *CoxeterComplex.chamber S* $(\psi$‘$C)$
⟨*proof*⟩

lemma *label-stab-map-inj-on-vertices*:
assumes φ: *label-wrt C0* φ *fixespointwise* φ *C0*
defines ψ: $\psi \equiv \lambda v.\ \{w \in W.\ w \rightarrow (\varphi\ v) = v\}$
shows *inj-on* ψ $(\bigcup X)$
⟨*proof*⟩

lemma *label-stab-map-surj-on-vertices*:
assumes *label-wrt C0* φ *fixespointwise* φ *C0*
defines $\psi \equiv \lambda v.\ \{w \in W.\ w \rightarrow (\varphi\ v) = v\}$
shows ψ‘$(\bigcup X) = \bigcup \Sigma$
⟨*proof*⟩

lemma *label-stab-map-bij-betw-vertices*:
  assumes *label-wrt C0 φ fixespointwise φ C0*
  defines $\psi \equiv \lambda v.\ \{w{\in}W.\ w{\rightarrow}(\varphi\ v) = v\}$
  shows *bij-betw* ψ $(\bigcup X)$ $(\bigcup \Sigma)$
  ⟨*proof*⟩

lemma *label-stab-map-bij-betw-W-chambers*:
  assumes *label-wrt C0 φ fixespointwise φ C0 w0*$\in W$
  defines $\psi \equiv \lambda v.\ \{w{\in}W.\ w{\rightarrow}(\varphi\ v) = v\}$
  shows *bij-betw* ψ (*w0*‘→*C0*) (*CoxeterComplex.smap S* {*w0*})
  ⟨*proof*⟩

lemma *label-stab-map-surj-on-simplices*:
  assumes φ: *label-wrt C0 φ fixespointwise φ C0*
  defines ψ: $\psi \equiv \lambda v.\ \{w{\in}W.\ w{\rightarrow}(\varphi\ v) = v\}$
  shows $\psi \vdash X = \Sigma$
⟨*proof*⟩

lemma *label-stab-map-iso-to-coxeter-complex*:
  assumes *label-wrt C0 φ fixespointwise φ C0*
  defines $\psi \equiv \lambda v.\ \{w{\in}W.\ w{\rightarrow}(\varphi\ v) = v\}$
  shows *ChamberComplexIsomorphism X Σ ψ*
⟨*proof*⟩

lemma *ex-iso-to-coxeter-complex′*:
  ∃ψ. *ChamberComplexIsomorphism X* (*CoxeterComplex.TheComplex S*) ψ
  ⟨*proof*⟩

lemma *ex-iso-to-coxeter-complex*:
  ∃*S*::′*a permutation set. CoxeterComplex S* ∧
    (∃ψ. *ChamberComplexIsomorphism X* (*CoxeterComplex.TheComplex S*) ψ)
  ⟨*proof*⟩

end

end

6 Buildings

In this section we collect the axioms for a (thick) building in a locale, and prove that apartments in a building are uniformly Coxeter.

theory *Building*
imports *Coxeter*

begin

6.1 Apartment systems

First we describe and explore the basic structure of apartment systems. An apartment system is a collection of isomorphic thin chamber subcomplexes with certain intersection properties.

6.1.1 Locale and basic facts

locale *ChamberComplexWithApartmentSystem* = *ChamberComplex X*
for X :: *'a set set*
+ **fixes** $\mathcal{A}$ :: *'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* :
$[\![A \in \mathcal{A};\ A' \in \mathcal{A};\ x \in A \cap A';\ C \in A \cap A';$ *chamber C* $]\!] \Longrightarrow$
$\exists f.$ *ChamberComplexIsomorphism A A' 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* [*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* ≡ (*SOME A. A*∈𝒜 ∧ *C*∈*A* ∧ *D*∈*A*)

lemma *supapartmentD*:
assumes *CD*: *chamber C chamber D*
defines *A* : *A* ≡ *supapartment C D*
shows *A*∈𝒜 *C*∈*A* *D*∈*A*
⟨*proof*⟩

lemma *iso-fixespointwise-chamber-in-int-apartments*:
assumes *apartments*: *A* ∈ 𝒜 *A′* ∈ 𝒜
and *chamber* : *chamber C C*∈*A*∩*A′*
and *iso* : *ChamberComplexIsomorphism A A′ f fixespointwise f C*
shows *fixespointwise f* (⋃(*A*∩*A′*))
⟨*proof*⟩

lemma *strong-intersecttwo*:
⟦ *A*∈𝒜; *A′*∈𝒜; *chamber C*; *C* ∈ *A*∩*A′* ⟧ ⟹
∃*f*. *ChamberComplexIsomorphism A A′ f* ∧ *fixespointwise f* (⋃(*A*∩*A′*))
⟨*proof*⟩

end

6.1.2 Isomorphisms between apartments

By standard uniqueness, the isomorphism between overlapping apartments guaranteed by the axiom *intersecttwo* is unique.

context *ChamberComplexWithApartmentSystem*
begin

lemma *ex1-apartment-iso*:
  assumes $A \in \mathcal{A}$ $A' \in \mathcal{A}$ *chamber* C $C \in A \cap A'$
  shows $\exists! f.$ *ChamberComplexIsomorphism* A A' f $\wedge$
    fixespointwise f $(\bigcup (A \cap A'))$ $\wedge$ *fixespointwise* f $(-\bigcup A)$
— The third clause in the conjunction is to facilitate uniqueness.
⟨*proof*⟩

definition *the-apartment-iso* :: $'a$ *set set* $\Rightarrow$ $'a$ *set set* $\Rightarrow$ $('a \Rightarrow 'a)$
  where *the-apartment-iso* A $A' \equiv$
    (*THE* f. *ChamberComplexIsomorphism* A A' f $\wedge$
      fixespointwise f $(\bigcup (A \cap A'))$ $\wedge$ *fixespointwise* f $(-\bigcup A)$)

lemma *the-apartment-isoD*:
  assumes $A \in \mathcal{A}$ $A' \in \mathcal{A}$ *chamber* C $C \in A \cap A'$
  defines $f \equiv$ *the-apartment-iso* A A'
  shows *ChamberComplexIsomorphism* A A' f *fixespointwise* f $(\bigcup (A \cap A'))$
    fixespointwise f $(-\bigcup A)$
  ⟨*proof*⟩

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*:
  ⟦ $A \in \mathcal{A}$; $B \in \mathcal{A}$; *chamber* C; $C \in A \cap B$; *chamber* D; $D \in A$ ⟧ $\Longrightarrow$
    chamber (*the-apartment-iso* A B ‘ D)
  ⟨*proof*⟩

lemma *the-apartment-iso-comp*:
  assumes *apartments*: $A \in \mathcal{A}$ $A' \in \mathcal{A}$ $A'' \in \mathcal{A}$
  and *chamber* : *chamber* C $C \in A \cap A' \cap A''$
  defines $f \equiv$ *the-apartment-iso* A A'
  and $g \equiv$ *the-apartment-iso* A' A''
  and $h \equiv$ *the-apartment-iso* A A''
  defines $gf \equiv$ *restrict1* $(g \circ f)$ $(\bigcup A)$
  shows $h = gf$
⟨*proof*⟩

lemma *the-apartment-iso-int-im*:
  assumes $A \in \mathcal{A}$ $A' \in \mathcal{A}$ *chamber* C $C \in A \cap A'$ $x \in A \cap A'$
  defines $f \equiv$ *the-apartment-iso* A A'
  shows $f`x = x$
  ⟨*proof*⟩

end

6.1.3 Retractions onto apartments

Since the isomorphism between overlapping apartments is the identity on their intersection, starting with a fixed chamber in a fixed apartment, we can construct a retraction onto that apartment as follows. Given a vertex in the complex, that vertex is contained a chamber, and that chamber lies in a common apartment with the fixed chamber. We then apply to the vertex the apartment isomorphism from that common apartment to the fixed apartment. It turns out that the image of the vertex does not depend on the containing chamber and apartment chosen, and so since the isomorphisms between apartments used are unique, such a retraction onto an apartment is canonical.

context *ChamberComplexWithApartmentSystem*
begin

definition *canonical-retraction* :: $'a$ *set set* $\Rightarrow$ $'a$ *set* $\Rightarrow$ $('a \Rightarrow 'a)$
  where *canonical-retraction* A C =
    restrict1 ($\lambda v.$ *the-apartment-iso* (*supapartment* (*supchamber* v) C) A v)
     ($\bigcup X$)

lemma *canonical-retraction-retraction*:
  assumes $A \in \mathcal{A}$ *chamber* C $C \in A$ $v \in \bigcup A$
  shows *canonical-retraction* A C $v = v$
⟨*proof*⟩

lemma *canonical-retraction-simplex-retraction1*:
  ⟦ $A \in \mathcal{A}$; *chamber* C; $C \in A$; $a \in A$ ⟧ $\Longrightarrow$
   fixespointwise (*canonical-retraction* A C) a
  ⟨*proof*⟩

lemma *canonical-retraction-simplex-retraction2*:
  ⟦ $A \in \mathcal{A}$; *chamber* C; $C \in A$; $a \in A$ ⟧ $\Longrightarrow$ *canonical-retraction* A C ` $a = a$
  ⟨*proof*⟩

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) ($\bigcup B$)
⟨*proof*⟩

lemma *canonical-retraction-uniform-im*:
  ⟦ $A{\in}\mathcal{A}$; $B{\in}\mathcal{A}$; *chamber* C; $C{\in}A{\cap}B$; $x{\in}B$ ⟧ $\Longrightarrow$
    canonical-retraction A C ‘ x = *the-apartment-iso* B A ‘ x
  ⟨*proof*⟩

lemma *canonical-retraction-simplex-im*:
  assumes $A{\in}\mathcal{A}$ *chamber* C $C{\in}A$
  shows *canonical-retraction* A C ⊢ $X = A$
⟨*proof*⟩

lemma *canonical-retraction-vertex-im*:
  ⟦ $A{\in}\mathcal{A}$; *chamber* C; $C{\in}A$ ⟧ $\Longrightarrow$ *canonical-retraction* A C ‘ $\bigcup X = \bigcup A$
  ⟨*proof*⟩

lemma *canonical-retraction*:
  assumes $A{\in}\mathcal{A}$ *chamber* C $C{\in}A$
  shows *ChamberComplexRetraction* X (*canonical-retraction* A C)
⟨*proof*⟩

lemma *canonical-retraction-comp-endomorphism*:
  ⟦ $A{\in}\mathcal{A}$; $B{\in}\mathcal{A}$; *chamber* C; *chamber* D; $C{\in}A$; $D{\in}B$ ⟧ $\Longrightarrow$
    ChamberComplexEndomorphism X
      (*canonical-retraction* A C ∘ *canonical-retraction* B D)
  ⟨*proof*⟩

lemma *canonical-retraction-comp-simplex-im-subset*:
  ⟦ $A{\in}\mathcal{A}$; $B{\in}\mathcal{A}$; *chamber* C; *chamber* D; $C{\in}A$; $D{\in}B$ ⟧ $\Longrightarrow$
      (*canonical-retraction* A C ∘ *canonical-retraction* B D) ⊢ $X \subseteq A$
  ⟨*proof*⟩

lemma *canonical-retraction-comp-apartment-endomorphism*:
  ⟦ $A{\in}\mathcal{A}$; $B{\in}\mathcal{A}$; *chamber* C; *chamber* D; $C{\in}A$; $D{\in}B$ ⟧ $\Longrightarrow$
    ChamberComplexEndomorphism A
      (*restrict1* (*canonical-retraction* A C ∘ *canonical-retraction* B D) ($\bigcup A$))
  ⟨*proof*⟩

end

6.1.4 Distances in apartments

Here we examine distances between chambers and between a facet and a chamber, especially with respect to canonical retractions onto an apartment. Note that a distance measured within an apartment is equal to the distance measured between the same objects in the wider chamber complex. In other words, the shortest distance between chambers can always be achieved within an apartment.

context *ChamberComplexWithApartmentSystem*

begin

lemma *apartment-chamber-distance*:
  assumes $A{\in}\mathcal{A}$ *chamber C chamber D* $C{\in}A$ $D{\in}A$
  shows *ChamberComplex.chamber-distance A C D = chamber-distance C D*
⟨*proof*⟩

lemma *apartment-min-gallery*:
  assumes $A{\in}\mathcal{A}$ *ChamberComplex.min-gallery A Cs*
  shows *min-gallery Cs*
⟨*proof*⟩

lemma *apartment-face-distance*:
  assumes $A{\in}\mathcal{A}$ *chamber C* $C{\in}A$ $F{\in}A$
  shows *ChamberComplex.face-distance A F C = face-distance F C*
⟨*proof*⟩

lemma *apartment-face-distance-eq-chamber-distance-compare-other-chamber*:
  assumes $A{\in}\mathcal{A}$ *chamber C chamber D chamber E* $C{\in}A$ $D{\in}A$ $E{\in}A$
    $z{\lhd}C$ $z{\lhd}D$ $C{\neq}D$ *chamber-distance C E* $\leq$ *chamber-distance D E*
  shows *face-distance z E = chamber-distance C E*
  ⟨*proof*⟩

lemma *canonical-retraction-face-distance-map*:
  assumes $A{\in}\mathcal{A}$ *chamber C chamber D* $C{\in}A$ $F{\subseteq}C$
  shows *face-distance F (canonical-retraction A C ‘ D) = face-distance F D*
⟨*proof*⟩

end

6.1.5 Special situation: a triangle of apartments and chambers

To facilitate proving that apartments in buildings have sufficient foldings to be Coxeter, we explore the situation of three chambers sharing a common facet, along with three apartments, each of which contains two of the chambers. A folding of one of the apartments is constructed by composing two apartment retractions, and by symmetry we automatically obtain an opposed folding.

locale *ChamberComplexApartmentSystemTriangle* =
  ChamberComplexWithApartmentSystem X $\mathcal{A}$
  for *X* :: *'a set set*
  and $\mathcal{A}$ :: *'a set set set*
\+ **fixes** *A B B'* :: *'a set set*
  and *C D E z* :: *'a set*
  assumes *apartments* : $A{\in}\mathcal{A}$ $B{\in}\mathcal{A}$ $B'{\in}\mathcal{A}$
  and *chambers* : *chamber C chamber D chamber E*
  and *facet* : $z{\lhd}C$ $z{\lhd}D$ $z{\lhd}E$
  and *in-apartments*: $C{\in}A{\cap}B$ $D{\in}A{\cap}B'$ $E{\in}B{\cap}B'$

and *chambers-ne* : $D \neq C$ $E \neq D$ $C \neq E$
begin

abbreviation *fold-A* ≡ *canonical-retraction A D* ∘ *canonical-retraction B C*
abbreviation *res-fold-A* ≡ *restrict1 fold-A* (⋃ *A*)
abbreviation *opp-fold-A* ≡ *canonical-retraction A C* ∘ *canonical-retraction B′ D*
abbreviation *res-opp-fold-A* ≡ *restrict1 opp-fold-A* (⋃ *A*)

lemma *rotate*: *ChamberComplexApartmentSystemTriangle X 𝒜 B′ A B D E C z*
⟨*proof*⟩

lemma *reflect*: *ChamberComplexApartmentSystemTriangle X 𝒜 A B′ B D C E z*
⟨*proof*⟩

lemma *facet-in-chambers*: $z \subseteq C$ $z \subseteq D$ $z \subseteq E$
⟨*proof*⟩

lemma *A-chambers*:
ChamberComplex.chamber A C ChamberComplex.chamber A D
⟨*proof*⟩

lemma *res-fold-A-A-chamber-image*:
ChamberComplex.chamber A F ⟹ *res-fold-A* ‘ *F* = *fold-A* ‘ *F*
⟨*proof*⟩

lemma *the-apartment-iso-middle-im*: *the-apartment-iso A B* ‘ *D* = *E*
⟨*proof*⟩

lemma *inside-canonical-retraction-chamber-images*:
canonical-retraction B C ‘ *C* = *C*
canonical-retraction B C ‘ *D* = *E*
canonical-retraction B C ‘ *E* = *E*
⟨*proof*⟩

lemmas *in-canretract-chimages* =
inside-canonical-retraction-chamber-images

lemma *outside-canonical-retraction-chamber-images*:
canonical-retraction A D ‘ *C* = *C*
canonical-retraction A D ‘ *D* = *D*
canonical-retraction A D ‘ *E* = *C*
⟨*proof*⟩

lemma *fold-A-chamber-images*:
fold-A ‘ *C* = *C fold-A* ‘ *D* = *C fold-A* ‘ *E* = *C*
⟨*proof*⟩

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*
⟨*proof*⟩

lemmas *res-opp-fold-A-chamber-images* =
ChamberComplexApartmentSystemTriangle.res-fold-A-chamber-images[*OF reflect*]

lemma *fold-A-fixespointwise1*: *fixespointwise fold-A C*
⟨*proof*⟩

lemmas *opp-fold-A-fixespointwise2* =
ChamberComplexApartmentSystemTriangle.fold-A-fixespointwise1[*OF reflect*]

lemma *fold-A-facet-im*: *fold-A* ' *z* = *z*
⟨*proof*⟩

lemma *fold-A-endo-X*: *ChamberComplexEndomorphism X fold-A*
⟨*proof*⟩

lemma *res-fold-A-endo-A*: *ChamberComplexEndomorphism A res-fold-A*
⟨*proof*⟩

lemmas *opp-res-fold-A-endo-A* =
ChamberComplexApartmentSystemTriangle.res-fold-A-endo-A[*OF reflect*]

lemma *fold-A-morph-A-A*: *ChamberComplexMorphism A A fold-A*
⟨*proof*⟩

lemmas *opp-fold-A-morph-A-A* =
ChamberComplexApartmentSystemTriangle.fold-A-morph-A-A[*OF reflect*]

lemma *res-fold-A-A-im-fold-A-A-im*: *res-fold-A* ⊢ *A* = *fold-A* ⊢ *A*
⟨*proof*⟩

lemmas *res-opp-fold-A-A-im-opp-fold-A-A-im* =
ChamberComplexApartmentSystemTriangle.res-fold-A-A-im-fold-A-A-im[
OF reflect
]

lemma *res-fold-A-𝒞-A-im-fold-A-𝒞-A-im*:
res-fold-A ⊢ (*ChamberComplex.chamber-system A*) =
fold-A ⊢ (*ChamberComplex.chamber-system A*)
⟨*proof*⟩

lemmas *res-opp-fold-A-𝒞-A-im-opp-fold-A-𝒞-A-im* =
ChamberComplexApartmentSystemTriangle.res-fold-A-𝒞-A-im-fold-A-𝒞-A-im[
OF reflect
]

lemma *chambercomplex-fold-A-im*: *ChamberComplex* (*fold-A* ⊢ *A*)
⟨*proof*⟩

lemmas *chambercomplex-opp-fold-A-im* =
ChamberComplexApartmentSystemTriangle.chambercomplex-fold-A-im[
OF reflect
]

lemma *chambersubcomplex-fold-A-im*:
ChamberComplex.ChamberSubcomplex A (*fold-A* ⊢ *A*)
⟨*proof*⟩

lemmas *chambersubcomplex-opp-fold-A-im* =
ChamberComplexApartmentSystemTriangle.chambersubcomplex-fold-A-im[
OF reflect
]

lemma *fold-A-facet-distance-map*:
chamber F ⟹ *face-distance z* (*fold-A*'*F*) = *face-distance z F*
⟨*proof*⟩

lemma *fold-A-min-gallery-betw-map*:
assumes *chamber F chamber G z*⊆*F*
face-distance z G = *chamber-distance F G min-gallery* (*F*#*Fs*@[*G*])
shows *min-gallery* (*fold-A*⊨(*F*#*Fs*@[*G*]))
⟨*proof*⟩

lemma *fold-A-chamber-system-image-fixespointwise*′:
defines $\mathcal{C}$-*A* : $\mathcal{C}$-*A* ≡ *ChamberComplex.*$\mathcal{C}$ *A*
defines *f*$\mathcal{C}$-*A*: *f*$\mathcal{C}$-*A* ≡ {*F*∈$\mathcal{C}$-*A*. *face-distance z F* = *chamber-distance C F*}
assumes *F* : *F*∈*f*$\mathcal{C}$-*A*
shows *fixespointwise fold-A F*
⟨*proof*⟩

lemma *fold-A-chamber-system-image*:
defines $\mathcal{C}$-*A* : $\mathcal{C}$-*A* ≡ *ChamberComplex.*$\mathcal{C}$ *A*
defines *f*$\mathcal{C}$-*A*: *f*$\mathcal{C}$-*A* ≡ {*F*∈$\mathcal{C}$-*A*. *face-distance z F* = *chamber-distance C F*}
shows *fold-A* ⊢ $\mathcal{C}$-*A* = *f*$\mathcal{C}$-*A*
⟨*proof*⟩

lemmas *opp-fold-A-chamber-system-image* =
ChamberComplexApartmentSystemTriangle.fold-A-chamber-system-image[
OF reflect
]

lemma *fold-A-chamber-system-image-fixespointwise*:
F ∈ *ChamberComplex.*$\mathcal{C}$ *A* ⟹ *fixespointwise fold-A* (*fold-A*'*F*)
⟨*proof*⟩

lemmas *fold-A-chsys-imfix* = *fold-A-chamber-system-image-fixespointwise*

lemmas *opp-fold-A-chamber-system-image-fixespointwise* =
  ChamberComplexApartmentSystemTriangle.fold-A-chsys-imfix[
    OF reflect
  ]

lemma *chamber-in-fold-A-im*:
  chamber F $\Longrightarrow$ *F* $\in$ *fold-A* $\vdash$ *A* $\Longrightarrow$ *F* $\in$ *fold-A* $\vdash$ *ChamberComplex.*$\mathcal{C}$ *A*
  ⟨*proof*⟩

lemmas *chamber-in-opp-fold-A-im* =
  ChamberComplexApartmentSystemTriangle.chamber-in-fold-A-im[*OF reflect*]

lemma *simplex-in-fold-A-im-image*:
  assumes *x* $\in$ *fold-A* $\vdash$ *A*
  shows *fold-A* ' *x* = *x*
⟨*proof*⟩

lemma *chamber1-notin-rfold-im*: *C* $\notin$ *opp-fold-A* $\vdash$ *A*
  ⟨*proof*⟩

lemma *fold-A-min-gallery-from1-map*:
  ⟦ *chamber F*; *F* $\in$ *fold-A* $\vdash$ *A*; *min-gallery* (*C#Fs*@[*F*]) ⟧ $\Longrightarrow$
    min-gallery (*C* # *fold-A* $\models$ *Fs* @ [*F*])
  ⟨*proof*⟩

lemma *fold-A-min-gallery-from2-map*:
  ⟦ *chamber F*; *F* $\in$ *opp-fold-A* $\vdash$ *A*; *min-gallery* (*D#Fs*@[*F*]) ⟧ $\Longrightarrow$
    min-gallery (*C* # *fold-A* $\models$ (*Fs*@[*F*]))
  ⟨*proof*⟩

lemma *fold-A-min-gallery-to2-map*:
  assumes *chamber F F* $\in$ *opp-fold-A* $\vdash$ *A min-gallery* (*F#Fs*@[*D*])
  shows *min-gallery* (*fold-A* $\models$ (*F#Fs*) @ [*C*])
  ⟨*proof*⟩

lemmas *opp-fold-A-min-gallery-from1-map* =
  ChamberComplexApartmentSystemTriangle.fold-A-min-gallery-from2-map[
    OF reflect
  ]

lemmas *opp-fold-A-min-gallery-to1-map* =
  ChamberComplexApartmentSystemTriangle.fold-A-min-gallery-to2-map[
    OF reflect
  ]

lemma *closer-to-chamber1-not-in-rfold-im-chamber-system*:
  assumes *chamber-distance C F* $\leq$ *chamber-distance D F*

shows $F \notin$ *ChamberComplex*.$\mathcal{C}$ (*opp-fold-A* $\vdash$ *A*)
⟨*proof*⟩

lemmas *clsrch1-nin-rfold-im-chsys* =
closer-to-chamber1-not-in-rfold-im-chamber-system

lemmas *closer-to-chamber2-not-in-fold-im-chamber-system* =
ChamberComplexApartmentSystemTriangle.clsrch1-nin-rfold-im-chsys[
OF reflect
]

lemma *fold-A-opp-fold-A-chamber-systems*:
ChamberComplex.$\mathcal{C}$ *A* =
(*ChamberComplex*.$\mathcal{C}$ (*fold-A* $\vdash$ *A*)) $\cup$ (*ChamberComplex*.$\mathcal{C}$ (*opp-fold-A* $\vdash$ *A*))
(*ChamberComplex*.$\mathcal{C}$ (*fold-A* $\vdash$ *A*)) $\cap$ (*ChamberComplex*.$\mathcal{C}$ (*opp-fold-A* $\vdash$ *A*)) =
{}
⟨*proof*⟩

lemma *fold-A-im-min-gallery'*:
assumes *ChamberComplex.min-gallery* (*fold-A* $\vdash$ *A*) (*C#Cs*)
shows *ChamberComplex.min-gallery A* (*C#Cs*)
⟨*proof*⟩

lemma *fold-A-im-min-gallery*:
ChamberComplex.min-gallery (*fold-A* $\vdash$ *A*) (*C#Cs*) $\Longrightarrow$ *min-gallery* (*C#Cs*)
⟨*proof*⟩

lemma *fold-A-comp-fixespointwise*:
fixespointwise (*fold-A* $\circ$ *opp-fold-A*) ($\bigcup$ (*fold-A* $\vdash$ *A*))
⟨*proof*⟩

lemmas *opp-fold-A-comp-fixespointwise* =
ChamberComplexApartmentSystemTriangle.fold-A-comp-fixespointwise[*OF reflect*]

lemma *fold-A-fold*:
ChamberComplexIsomorphism (*opp-fold-A* $\vdash$ *A*) (*fold-A* $\vdash$ *A*) *fold-A*
⟨*proof*⟩

lemma *res-fold-A*: *ChamberComplexFolding A res-fold-A*
⟨*proof*⟩

lemmas *opp-res-fold-A* =
ChamberComplexApartmentSystemTriangle.res-fold-A[*OF reflect*]

end

6.2 Building locale and basic lemmas

Finally, we define a (thick) building to be a thick chamber complex with a system of apartments.

locale *Building* = *ChamberComplexWithApartmentSystem X* $\mathcal{A}$
  for *X* :: *'a set set*
  and $\mathcal{A}$ :: *'a set set set*
\+ **assumes** *thick*: *ThickChamberComplex X*
begin

abbreviation *some-third-chamber* ≡
    ThickChamberComplex.some-third-chamber X

lemmas *some-third-chamberD-facet* =
  ThickChamberComplex.some-third-chamberD-facet [*OF thick*]

lemmas *some-third-chamberD-ne* =
  ThickChamberComplex.some-third-chamberD-ne [*OF thick*]

lemmas *chamber-some-third-chamber* =
  ThickChamberComplex.chamber-some-third-chamber [*OF thick*]

end

6.3 Apartments are uniformly Coxeter

Using the assumption of thickness, we may use the special situation *ChamberComplexApartmentSystemTriangle* to verify that apartments have enough pairs of opposed foldings to ensure that they are isomorphic to a Coxeter complex. Since the apartments are all isomorphic, they are uniformly isomorphic to a single Coxeter complex.

context *Building*
begin

lemma *apartments-have-many-foldings1*:
  assumes $A \in \mathcal{A}$ *chamber C chamber D* $C \sim D$ $C \neq D$ $C \in A$ $D \in A$
  defines *E* ≡ *some-third-chamber C D* $(C \cap D)$
  defines *B* ≡ *supapartment C E*
  and *B′* ≡ *supapartment D E*
  defines *f* ≡ *restrict1* (*canonical-retraction A D* ∘ *canonical-retraction B C*) $(\bigcup A)$
  and *g* ≡ *restrict1* (*canonical-retraction A C* ∘ *canonical-retraction B′ D*) $(\bigcup A)$
  shows *f‘D* = *C ChamberComplexFolding A f*
    g‘C = *D ChamberComplexFolding A g*
⟨*proof*⟩

lemma *apartments-have-many-foldings2*:

assumes $A{\in}\mathcal{A}$ *chamber C chamber D* $C{\sim}D$ $C{\neq}D$ $C{\in}A$ $D{\in}A$
defines $E \equiv$ *some-third-chamber C D* $(C{\cap}D)$
defines $B \equiv$ *supapartment C E*
and $B' \equiv$ *supapartment D E*
defines $f \equiv$ *restrict1* (*canonical-retraction A D* $\circ$ *canonical-retraction B C*) $(\bigcup A)$
and $g \equiv$ *restrict1* (*canonical-retraction A C* $\circ$ *canonical-retraction B' D*) $(\bigcup A)$
shows *OpposedThinChamberComplexFoldings A f g C*
⟨*proof*⟩

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*⟩

lemma *apartments-have-many-foldings*:
assumes $A{\in}\mathcal{A}$ $C{\in}A$ *chamber C*
shows *ThinChamberComplexManyFoldings A C*
⟨*proof*⟩

theorem *apartments-are-coxeter*:
$A{\in}\mathcal{A} \Longrightarrow \exists S{::}'a$ *permutation set*. (
CoxeterComplex S $\wedge$
($\exists\,\psi$. *ChamberComplexIsomorphism A* (*CoxeterComplex.TheComplex S*) ψ)
)
⟨*proof*⟩

corollary *apartments-are-uniformly-coxeter*:
assumes $X{\neq}\{\}$
shows $\exists S{::}'a$ *permutation set*. *CoxeterComplex S* $\wedge$
($\forall A{\in}\mathcal{A}.\ \exists\,\psi.$
ChamberComplexIsomorphism A (*CoxeterComplex.TheComplex S*) ψ
)
⟨*proof*⟩

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.