Free Groups

Joachim Breitner

March 17, 2025

Abstract

Free Groups are, in a sense, the most generic kind of group. They are defined over a set of generators with no additional relations in between them. They play an important role in the definition of group presentations and in other fields.

This theory provides the definition of Free Group as the set of fully canceled words in the generators. The universal property is proven, as well as some isomorphisms results about Free Groups.

Contents

1 Cancelation of words of generators and their inverses

theory *Cancelation*
imports
HOL−Proofs−Lambda.Commutation
begin

This theory defines cancelation via relations. The one-step relation *cancels-to-1* a b describes that b is obtained from a by removing exactly one pair of generators, while *cancels-to* is the reflexive transitive hull of that relation. Due to confluence, this relation has a normal form, allowing for the definition of *normalize*.

1.1 Auxiliary results

Some lemmas that would be useful in a more general setting are collected beforehand.

1.1.1 Auxiliary results about relations

These were helpfully provided by Andreas Lochbihler.

theorem *lconfluent-confluent*:
$[\![$ *wfP* $(R\hat{}{-}{-}1)$; $\bigwedge a\ b\ c.\ R\ a\ b \Longrightarrow R\ a\ c \Longrightarrow \exists\, d.\ R\hat{}{*}{*}\ b\ d \wedge R\hat{}{*}{*}\ c\ d$ $]\!] \Longrightarrow$ *confluent* R
by(*auto simp add*: *diamond-def commute-def square-def intro*: *newman*)

lemma *confluentD*:
$[\![$ *confluent* R; $R\hat{}{*}{*}\ a\ b$; $R\hat{}{*}{*}\ a\ c$ $]\!] \Longrightarrow \exists\, d.\ R\hat{}{*}{*}\ b\ d \wedge R\hat{}{*}{*}\ c\ d$
by(*auto simp add*: *commute-def diamond-def square-def*)

lemma *tranclp-DomainP*: $R\hat{}{+}{+}\ a\ b \Longrightarrow$ *Domainp* R a
by(*auto elim*: *converse-tranclpE*)

lemma *confluent-unique-normal-form*:
$\llbracket$ *confluent R*; $R\hat{}{**}\ a\ b$; $R\hat{}{**}\ a\ c$; $\neg$ *Domainp R b*; $\neg$ *Domainp R c* $\rrbracket \Longrightarrow b = c$
by(*fastforce dest*!: *confluentD*[*of R a b c*] *dest*: *tranclp-DomainP rtranclpD*[**where** $a=b$] *rtranclpD*[**where** $a=c$])

1.2 Definition of the *canceling* relation

type-synonym *'a g-i* = (*bool* × *'a*)
type-synonym *'a word-g-i* = *'a g-i list*

These type aliases encode the notion of a "generator or its inverse" (*'a g-i*) and the notion of a "word in generators and their inverses" (*'a word-g-i*), which form the building blocks of Free Groups.

definition *canceling* :: *'a g-i* ⇒ *'a g-i* ⇒ *bool*
where *canceling a b* = ((*snd a* = *snd b*) ∧ (*fst a* ≠ *fst b*))

1.2.1 Simple results about canceling

A generators cancels with its inverse, either way. The relation is symmetic.

lemma *cancel-cancel*: $\llbracket$ *canceling a b*; *canceling b c* $\rrbracket \Longrightarrow a = c$
by (*auto intro*: *prod-eqI simp add*:*canceling-def*)

lemma *cancel-sym*: *canceling a b* ⟹ *canceling b a*
by (*simp add*:*canceling-def*)

lemma *cancel-sym-neg*: ¬*canceling a b* ⟹ ¬*canceling b a*
by (*rule classical*, *simp add*:*canceling-def*)

1.3 Definition of the *cancels-to* relation

First, we define the function that removes the ith and $(i+1)$st element from a word of generators, together with basic properties.

definition *cancel-at* :: *nat* ⇒ *'a word-g-i* ⇒ *'a word-g-i*
where *cancel-at i l* = *take i l* @ *drop* (*2*+*i*) *l*

lemma *cancel-at-length*[*simp*]:
1+*i* < *length l* ⟹ *length* (*cancel-at i l*) = *length l* − *2*
by(*auto simp add*: *cancel-at-def*)

lemma *cancel-at-nth1*[*simp*]:
$\llbracket$ *n* < *i*; *1*+*i* < *length l* $\rrbracket$ ⟹ (*cancel-at i l*) ! *n* = *l* ! *n*
by(*auto simp add*: *cancel-at-def nth-append*)

lemma *cancel-at-nth2*[*simp*]:
assumes *n* ≥ *i* **and** *n* < *length l* − *2*
shows (*cancel-at i l*) ! *n* = *l* ! (*n* + *2*)
proof−
from ‹*n* ≥ *i*› **and** ‹*n* < *length l* − *2*›

have $i = min\ (length\ l)\ i$
  by *auto*
with ‹$n \geq i$› **and** ‹$n < length\ l - 2$›
show $(cancel\text{-}at\ i\ l)\ !\ n = l\ !\ (n + 2)$
  by(*auto simp add*: *cancel-at-def nth-append nth-via-drop*)
qed

Then we can define the relation *cancels-to-1-at* $i\ a\ b$ which specifies that b can be obtained by a by canceling the ith and $(i+1)$st position.

Based on that, we existentially quantify over the position i to obtain the relation *cancels-to-1*, of which *cancels-to* is the reflexive and transitive closure.

A word is *canceled* if it can not be canceled any futher.

definition *cancels-to-1-at* :: $nat \Rightarrow 'a\ word\text{-}g\text{-}i \Rightarrow 'a\ word\text{-}g\text{-}i \Rightarrow bool$
where $cancels\text{-}to\text{-}1\text{-}at\ i\ l1\ l2 = (0 \leq i \wedge (1+i) < length\ l1$
$\wedge\ canceling\ (l1\ !\ i)\ (l1\ !\ (1+i))$
$\wedge\ (l2 = cancel\text{-}at\ i\ l1))$

definition *cancels-to-1* :: $'a\ word\text{-}g\text{-}i \Rightarrow 'a\ word\text{-}g\text{-}i \Rightarrow bool$
where $cancels\text{-}to\text{-}1\ l1\ l2 = (\exists\, i.\ cancels\text{-}to\text{-}1\text{-}at\ i\ l1\ l2)$

definition *cancels-to* :: $'a\ word\text{-}g\text{-}i \Rightarrow 'a\ word\text{-}g\text{-}i \Rightarrow bool$
where $cancels\text{-}to = cancels\text{-}to\text{-}1\hat{}**$

lemma *cancels-to-trans* [*trans*]:
$[\![\ cancels\text{-}to\ a\ b;\ cancels\text{-}to\ b\ c\ ]\!] \Longrightarrow cancels\text{-}to\ a\ c$
by (*auto simp add*:*cancels-to-def*)

definition *canceled* :: $'a\ word\text{-}g\text{-}i \Rightarrow bool$
where $canceled\ l = (\neg\ Domainp\ cancels\text{-}to\text{-}1\ l)$

lemma *cancels-to-1-unfold*:
assumes *cancels-to-1 x y*
obtains *xs1 x1 x2 xs2*
where $x = xs1\ @\ x1\ \#\ x2\ \#\ xs2$
  and $y = xs1\ @\ xs2$
  and *canceling x1 x2*
proof−
assume a: $(\bigwedge xs1\ x1\ x2\ xs2.\ [\![x = xs1\ @\ x1\ \#\ x2\ \#\ xs2;\ y = xs1\ @\ xs2;\ canceling\ x1\ x2]\!] \Longrightarrow thesis)$
from ‹*cancels-to-1 x y*›
obtain i **where** *cancels-to-1-at i x y*
  unfolding *cancels-to-1-def* **by** *auto*
hence $canceling\ (x\ !\ i)\ (x\ !\ Suc\ i)$
  and $y = (take\ i\ x)\ @\ (drop\ (Suc\ (Suc\ i))\ x)$
  and $x = (take\ i\ x)\ @\ x\ !\ i\ \#\ x\ !\ Suc\ i\ \#\ (drop\ (Suc\ (Suc\ i))\ x)$
unfolding *cancel-at-def* **and** *cancels-to-1-at-def* **by** (*auto simp add*: *Cons-nth-drop-Suc*)
with a **show** *thesis* **by** *blast*

qed

lemma *cancels-to-1-fold*:
canceling x1 x2 $\Longrightarrow$ *cancels-to-1* (*xs1* @ *x1* # *x2* # *xs2*) (*xs1* @ *xs2*)
unfolding *cancels-to-1-def* **and** *cancels-to-1-at-def* **and** *cancel-at-def*
by (*rule-tac x*=*length xs1* **in** *exI*, *auto simp add*:*nth-append*)

1.3.1 Existence of the normal form

One of two steps to show that we have a normal form is the following lemma, guaranteeing that by canceling, we always end up at a fully canceled word.

lemma *canceling-terminates*: *wfP* (*cancels-to-1*^−−*1*)
proof−
have *wf* (*measure length*) **by** *auto*
moreover
have {(*x*, *y*). *cancels-to-1 y x*} ⊆ *measure length*
by (*auto simp add*: *cancels-to-1-def cancel-at-def cancels-to-1-at-def*)
ultimately
have *wf* {(*x*, *y*). *cancels-to-1 y x*}
by(*rule wf-subset*)
thus *?thesis* **by** (*simp add*:*wfp-def*)
qed

The next two lemmas prepare for the proof of confluence. It does not matter in which order we cancel, we can obtain the same result.

lemma *canceling-neighbor*:
assumes *cancels-to-1-at i l a* **and** *cancels-to-1-at* (*Suc i*) *l b*
shows *a* = *b*
proof−
from ‹*cancels-to-1-at i l a*›
have *canceling* (*l* ! *i*) (*l* ! *Suc i*) **and** *i* < *length l*
by (*auto simp add*: *cancels-to-1-at-def*)

from ‹*cancels-to-1-at* (*Suc i*) *l b*›
have *canceling* (*l* ! *Suc i*) (*l* ! *Suc* (*Suc i*)) **and** *Suc* (*Suc i*) < *length l*
by (*auto simp add*: *cancels-to-1-at-def*)

from ‹*canceling* (*l* ! *i*) (*l* ! *Suc i*)› **and** ‹*canceling* (*l* ! *Suc i*) (*l* ! *Suc* (*Suc i*))›
have *l* ! *i* = *l* ! *Suc* (*Suc i*) **by** (*rule cancel-cancel*)

from ‹*cancels-to-1-at* (*Suc i*) *l b*›
have *b* = *take* (*Suc i*) *l* @ *drop* (*Suc* (*Suc* (*Suc i*))) *l*
by (*simp add*: *cancels-to-1-at-def cancel-at-def*)
also from ‹*i* < *length l*›
have ... = *take i l* @ [*l* ! *i*] @ *drop* (*Suc* (*Suc* (*Suc i*))) *l*
by(*auto simp add*: *take-Suc-conv-app-nth*)
also from ‹*l* ! *i* = *l* ! *Suc* (*Suc i*)›
have ... = *take i l* @ [*l* ! *Suc* (*Suc i*)] @ *drop* (*Suc* (*Suc* (*Suc i*))) *l*

by *simp*
also from ‹*Suc* (*Suc* *i*) < *length* *l*›
have ... = *take* *i* *l* @ *drop* (*Suc* (*Suc* *i*)) *l*
by (*simp* *add*: *Cons-nth-drop-Suc*)
also from ‹*cancels-to-1-at* *i* *l* *a*› **have** ... = *a*
by (*simp* *add*: *cancels-to-1-at-def* *cancel-at-def*)
finally show *a* = *b* **by**(*rule* *sym*)
qed

lemma *canceling-indep*:
assumes *cancels-to-1-at* *i* *l* *a* **and** *cancels-to-1-at* *j* *l* *b* **and** *j* > *Suc* *i*
obtains *c* **where** *cancels-to-1-at* (*j* − *2*) *a* *c* **and** *cancels-to-1-at* *i* *b* *c*
proof(*atomize-elim*)
from ‹*cancels-to-1-at* *i* *l* *a*›
have *Suc* *i* < *length* *l*
and *canceling* (*l* ! *i*) (*l* ! *Suc* *i*)
and *a* = *cancel-at* *i* *l*
and *length* *a* = *length* *l* − *2*
and *min* (*length* *l*) *i* = *i*
by (*auto* *simp* *add*:*cancels-to-1-at-def*)
from ‹*cancels-to-1-at* *j* *l* *b*›
have *Suc* *j* < *length* *l*
and *canceling* (*l* ! *j*) (*l* ! *Suc* *j*)
and *b* = *cancel-at* *j* *l*
and *length* *b* = *length* *l* − *2*
by (*auto* *simp* *add*:*cancels-to-1-at-def*)

let *?c* = *cancel-at* (*j* − *2*) *a*
from ‹*j* > *Suc* *i*›
have *Suc* (*Suc* (*j* − *2*)) = *j*
and *Suc* (*Suc* (*Suc* *j* − *2*)) = *Suc* *j*
by *auto*
with ‹*min* (*length* *l*) *i* = *i*› **and** ‹*j* > *Suc* *i*› **and** ‹*Suc* *j* < *length* *l*›
have (*l* ! *j*) = (*cancel-at* *i* *l* ! (*j* − *2*))
and (*l* ! (*Suc* *j*)) = (*cancel-at* *i* *l* ! *Suc* (*j* − *2*))
by(*auto* *simp* *add*:*cancel-at-def* *simp* *add*:*nth-append*)

with ‹*cancels-to-1-at* *i* *l* *a*›
and ‹*cancels-to-1-at* *j* *l* *b*›
have *canceling* (*a* ! (*j* − *2*)) (*a* ! *Suc* (*j* − *2*))
by(*auto* *simp* *add*:*cancels-to-1-at-def*)

with ‹*j* > *Suc* *i*› **and** ‹*Suc* *j* < *length* *l*› **and** ‹*length* *a* = *length* *l* − *2*›
have *cancels-to-1-at* (*j* − *2*) *a* *?c* **by** (*auto* *simp* *add*: *cancels-to-1-at-def*)

from ‹*length* *b* = *length* *l* − *2*› **and** ‹*j* > *Suc* *i*› **and** ‹*Suc* *j* < *length* *l*›
have *Suc* *i* < *length* *b* **by** *auto*

moreover from ‹*b* = *cancel-at* *j* *l*› **and** ‹*j* > *Suc* *i*› **and** ‹*Suc* *i* < *length* *l*›

have (*b* ! *i*) = (*l* ! *i*) **and** (*b* ! *Suc i*) = (*l* ! *Suc i*)
by (*auto simp add*:*cancel-at-def nth-append*)
with ‹*canceling* (*l* ! *i*) (*l* ! *Suc i*)›
have *canceling* (*b* ! *i*) (*b* ! *Suc i*) **by** *simp*

moreover from ‹*j* > *Suc i*› **and** ‹*Suc j* < *length l*›
have *min i j* = *i*
and *min* (*j* − *2*) *i* = *i*
and *min* (*length l*) *j* = *j*
and *min* (*length l*) *i* = *i*
and *Suc* (*Suc* (*j* − *2*)) = *j*
by *auto*
with ‹*a* = *cancel-at i l*› **and** ‹*b* = *cancel-at j l*› **and** ‹*Suc* (*Suc* (*j* − *2*)) = *j*›
have *cancel-at* (*j* − *2*) *a* = *cancel-at i b*
by (*auto simp add*:*cancel-at-def take-drop*)

ultimately have *cancels-to-1-at i b* (*cancel-at* (*j* − *2*) *a*)
by (*auto simp add*:*cancels-to-1-at-def*)

with ‹*cancels-to-1-at* (*j* − *2*) *a ?c*›
show ∃ *c*. *cancels-to-1-at* (*j* − *2*) *a c* ∧ *cancels-to-1-at i b c* **by** *blast*
qed

This is the confluence lemma

lemma *confluent-cancels-to-1*: *confluent cancels-to-1*
proof(*rule lconfluent-confluent*)
show *wfP cancels-to-1*$^{-1-1}$ **by** (*rule canceling-terminates*)
next
fix *a b c*
assume *cancels-to-1 a b*
then obtain *i* **where** *cancels-to-1-at i a b*
by(*simp add*: *cancels-to-1-def*)(*erule exE*)
assume *cancels-to-1 a c*
then obtain *j* **where** *cancels-to-1-at j a c*
by(*simp add*: *cancels-to-1-def*)(*erule exE*)

show ∃ *d*. *cancels-to-1*** *b d* ∧ *cancels-to-1*** *c d*
proof (*cases i*=*j*)
assume *i*=*j*
from ‹*cancels-to-1-at i a b*›
have *b* = *cancel-at i a* **by** (*simp add*:*cancels-to-1-at-def*)
moreover from ‹*i*=*j*›
have … = *cancel-at j a* **by** (*clarify*)
moreover from ‹*cancels-to-1-at j a c*›
have … = *c* **by** (*simp add*:*cancels-to-1-at-def*)
ultimately have *b* = *c* **by** (*simp*)
hence *cancels-to-1*** *b b*
and *cancels-to-1*** *c b* **by** *auto*
thus ∃ *d*. *cancels-to-1*** *b d* ∧ *cancels-to-1*** *c d* **by** *blast*

```
  next
    assume i ≠ j
    show ?thesis
    proof (cases j = Suc i)
      assume j = Suc i
        with ‹cancels-to-1-at i a b› and ‹cancels-to-1-at j a c›
        have b = c by (auto elim: canceling-neighbor)
      hence cancels-to-1** b b
        and cancels-to-1** c b by auto
      thus ∃ d. cancels-to-1** b d ∧ cancels-to-1** c d by blast
    next
      assume j ≠ Suc i
      show ?thesis
      proof (cases i = Suc j)
        assume i = Suc j
          with ‹cancels-to-1-at i a b› and ‹cancels-to-1-at j a c›
          have c = b by (auto elim: canceling-neighbor)
        hence cancels-to-1** b b
          and cancels-to-1** c b by auto
        thus ∃ d. cancels-to-1** b d ∧ cancels-to-1** c d by blast
      next
        assume i ≠ Suc j
        show ?thesis
        proof (cases i < j)
          assume i < j
            with ‹j ≠ Suc i› have Suc i < j by auto
          with ‹cancels-to-1-at i a b› and ‹cancels-to-1-at j a c›
          obtain d where cancels-to-1-at (j − 2) b d and cancels-to-1-at i c d
            by(erule canceling-indep)
          hence cancels-to-1 b d and cancels-to-1 c d
            by (auto simp add:cancels-to-1-def)
          thus ∃ d. cancels-to-1** b d ∧ cancels-to-1** c d by (auto)
        next
          assume ¬ i < j
          with ‹j ≠ Suc i› and ‹i ≠ j› and ‹i ≠ Suc j› have Suc j < i by auto
          with ‹cancels-to-1-at i a b› and ‹cancels-to-1-at j a c›
          obtain d where cancels-to-1-at (i − 2) c d and cancels-to-1-at j b d
            by −(erule canceling-indep)
          hence cancels-to-1 b d and cancels-to-1 c d
            by (auto simp add:cancels-to-1-def)
          thus ∃ d. cancels-to-1** b d ∧ cancels-to-1** c d by (auto)
        qed
      qed
    qed
  qed
qed
```

And finally, we show that there exists a unique normal form for each word.

lemma *norm-form-uniq*:
  assumes *cancels-to a b*
    and *cancels-to a c*
    and *canceled b*
    and *canceled c*
  shows $b = c$
proof−
  have *confluent cancels-to-1* **by** (*rule confluent-cancels-to-1*)
  moreover
  from ‹*cancels-to a b*› **have** *cancels-to-1*^** *a b* **by** (*simp add*: *cancels-to-def*)
  moreover
  from ‹*cancels-to a c*› **have** *cancels-to-1*^** *a c* **by** (*simp add*: *cancels-to-def*)
  moreover
  from ‹*canceled b*› **have** ¬ *Domainp cancels-to-1 b* **by** (*simp add*: *canceled-def*)
  moreover
  from ‹*canceled c*› **have** ¬ *Domainp cancels-to-1 c* **by** (*simp add*: *canceled-def*)
  ultimately
  show $b = c$
    by (*rule confluent-unique-normal-form*)
qed

1.3.2 Some properties of cancelation

Distributivity rules of cancelation and *append*.

lemma *cancel-to-1-append*:
  assumes *cancels-to-1 a b*
  shows *cancels-to-1* ($l@a@l'$) ($l@b@l'$)
proof−
  from ‹*cancels-to-1 a b*› **obtain** *i* **where** *cancels-to-1-at i a b*
    by(*simp add*: *cancels-to-1-def*)(*erule exE*)
  hence *cancels-to-1-at* (*length l* + *i*) ($l@a@l'$) ($l@b@l'$)
    by (*auto simp add*:*cancels-to-1-at-def nth-append cancel-at-def*)
  thus *cancels-to-1* ($l@a@l'$) ($l@b@l'$)
    by (*auto simp add*: *cancels-to-1-def*)
qed

lemma *cancel-to-append*:
  assumes *cancels-to a b*
  shows *cancels-to* ($l@a@l'$) ($l@b@l'$)
using *assms*
unfolding *cancels-to-def*
proof(*induct*)
  case *base* **show** *?case* **by** (*simp add*:*cancels-to-def*)
next
  case (*step b c*)
  from ‹*cancels-to-1 b c*›
  have *cancels-to-1* (l @ b @ l') (l @ c @ l') **by** (*rule cancel-to-1-append*)
  with ‹*cancels-to-1*^** (l @ a @ l') (l @ b @ l')› **show** *?case*
    by (*auto simp add*:*cancels-to-def*)

qed

lemma *cancels-to-append2*:
  assumes *cancels-to a a′*
    and *cancels-to b b′*
  shows *cancels-to* (*a*@*b*) (*a′*@*b′*)
using ‹*cancels-to a a′*›
unfolding *cancels-to-def*
proof(*induct*)
  case *base*
  from ‹*cancels-to b b′*› **have** *cancels-to* (*a*@*b*@[]) (*a*@*b′*@[])
   by (*rule cancel-to-append*)
  thus *?case* **unfolding** *cancels-to-def* **by** *simp*
next
  case (*step ba c*)
  from ‹*cancels-to-1 ba c*› **have** *cancels-to-1* ([]@*ba*@*b′*) ([]@*c*@*b′*)
   by(*rule cancel-to-1-append*)
  with ‹*cancels-to-1*^** (*a* @ *b*) (*ba* @ *b′*)›
  show *?case* **unfolding** *cancels-to-def* **by** *simp*
qed

The empty list is canceled, a one letter word is canceled and a word is trivially cancled from itself.

lemma *empty-canceled*[*simp*]: *canceled* []
by(*auto simp add*: *canceled-def cancels-to-1-def cancels-to-1-at-def*)

lemma *singleton-canceled*[*simp*]: *canceled* [*a*]
by(*auto simp add*: *canceled-def cancels-to-1-def cancels-to-1-at-def*)

lemma *cons-canceled*:
  assumes *canceled* (*a*#*x*)
  shows *canceled x*
proof(*rule ccontr*)
  assume ¬ *canceled x*
  hence *Domainp cancels-to-1 x* **by** (*simp add*:*canceled-def*)
  then obtain *x′* **where** *cancels-to-1 x x′* **by** *auto*
  then obtain *xs1 x1 x2 xs2*
   where *x*: *x* = *xs1* @ *x1* # *x2* # *xs2*
   and *canceling x1 x2* **by** (*rule cancels-to-1-unfold*)
  hence *cancels-to-1* ((*a*#*xs1*) @ *x1* # *x2* # *xs2*) ((*a*#*xs1*) @ *xs2*)
   by (*auto intro*:*cancels-to-1-fold simp del*:*append-Cons*)
  with *x*
  have *cancels-to-1* (*a*#*x*) (*a*#*xs1* @ *xs2*)
   by *simp*
  hence ¬ *canceled* (*a*#*x*) **by** (*auto simp add*:*canceled-def*)
  thus *False* **using** ‹*canceled* (*a*#*x*)› **by** *contradiction*
qed

lemma *cancels-to-self*[*simp*]: *cancels-to l l*

by (*simp add*:*cancels-to-def*)

1.4 Definition of normalization

Using the THE construct, we can define the normalization function *normalize* as the unique fully cancled word that the argument cancels to.

definition *normalize* :: $'a$ *word-g-i* $\Rightarrow$ $'a$ *word-g-i*
where *normalize* l = (*THE* l'. *cancels-to* l l' $\wedge$ *canceled* l')

Some obvious properties of the normalize function, and other useful lemmas.

lemma
  shows *normalized-canceled*[*simp*]: *canceled* (*normalize* l)
  and *normalized-cancels-to*[*simp*]: *cancels-to* l (*normalize* l)
proof−
  let *?Q* = $\{l'$. *cancels-to-1*$\hat{}$** l $l'\}$
  have $l \in$ *?Q* **by** (*auto*) **hence** $\exists x.\ x \in$ *?Q* **by** (*rule exI*)

  have *wfP cancels-to-1*$\hat{}$−−*1*
    by (*rule canceling-terminates*)
  hence $\forall Q.$ ($\exists x.\ x \in Q$) $\longrightarrow$ ($\exists z{\in}Q.\ \forall y.$ *cancels-to-1* z y $\longrightarrow$ $y \notin Q$)
    by (*simp add*:*wfp-eq-minimal*)
  hence ($\exists x.\ x \in$ *?Q*) $\longrightarrow$ ($\exists z{\in}$*?Q*. $\forall y.$ *cancels-to-1* z y $\longrightarrow$ $y \notin$ *?Q*)
    by (*erule-tac x=?Q* **in** *allE*)
  then obtain l' **where** $l' \in$ *?Q* **and** *minimal*: $\bigwedge y.$ *cancels-to-1* l' y $\Longrightarrow$ $y \notin$ *?Q*
    by *auto*

  from ‹$l' \in$ *?Q*› **have** *cancels-to* l l' **by** (*auto simp add*: *cancels-to-def*)

  have *canceled* l'
  proof(*rule ccontr*)
    assume ¬ *canceled* l' **hence** *Domainp cancels-to-1* l' **by** (*simp add*: *canceled-def*)
    then obtain y **where** *cancels-to-1* l' y **by** *auto*
    with ‹*cancels-to* l l'› **have** *cancels-to* l y **by** (*auto simp add*: *cancels-to-def*)
    from ‹*cancels-to-1* l' y› **have** $y \notin$ *?Q* **by**(*rule minimal*)
    hence ¬ *cancels-to-1*$\hat{}$** l y **by** *auto*
    hence ¬ *cancels-to* l y **by** (*simp add*: *cancels-to-def*)
    with ‹*cancels-to* l y› **show** *False* **by** *contradiction*
  qed

  from ‹*cancels-to* l l'› **and** ‹*canceled* l'›
  have *cancels-to* l l' $\wedge$ *canceled* l' **by** *simp*
  hence *cancels-to* l (*normalize* l) $\wedge$ *canceled* (*normalize* l)
    unfolding *normalize-def*
  proof (*rule theI*)
    fix $l'a$
    assume *cancels-to* l $l'a$ $\wedge$ *canceled* $l'a$
   thus $l'a = l'$ **using** ‹*cancels-to* l l' $\wedge$ *canceled* l'› **by** (*auto elim*:*norm-form-uniq*)

qed
thus *canceled* (*normalize l*) **and** *cancels-to l* (*normalize l*) **by** *auto*
qed

lemma *normalize-discover*:
assumes *canceled l′*
and *cancels-to l l′*
shows *normalize l = l′*
proof−
from ‹*canceled l′*› **and** ‹*cancels-to l l′*›
have *cancels-to l l′* ∧ *canceled l′* **by** *auto*
thus *?thesis* **unfolding** *normalize-def* **by** (*auto elim*:*norm-form-uniq*)
qed

Words, related by cancelation, have the same normal form.

lemma *normalize-canceled*[*simp*]:
assumes *cancels-to l l′*
shows *normalize l = normalize l′*
proof(*rule normalize-discover*)
show *canceled* (*normalize l′*) **by** (*rule normalized-canceled*)
next
have *cancels-to l′* (*normalize l′*) **by** (*rule normalized-cancels-to*)
with ‹*cancels-to l l′*›
show *cancels-to l* (*normalize l′*) **by** (*rule cancels-to-trans*)
qed

Normalization is idempotent.

lemma *normalize-idemp*[*simp*]:
assumes *canceled l*
shows *normalize l = l*
using *assms*
by(*rule normalize-discover*)(*rule cancels-to-self*)

This lemma lifts the distributivity results from above to the normalize function.

lemma *normalize-append-cancel-to*:
assumes *cancels-to l1 l1′*
and *cancels-to l2 l2′*
shows *normalize* (*l1* @ *l2*) = *normalize* (*l1′* @ *l2′*)
proof(*rule normalize-discover*)
show *canceled* (*normalize* (*l1′* @ *l2′*)) **by** (*rule normalized-canceled*)
next
from ‹*cancels-to l1 l1′*› **and** ‹*cancels-to l2 l2′*›
have *cancels-to* (*l1* @ *l2*) (*l1′* @ *l2′*) **by** (*rule cancels-to-append2*)
also
have *cancels-to* (*l1′* @ *l2′*) (*normalize* (*l1′* @ *l2′*)) **by** (*rule normalized-cancels-to*)
finally
show *cancels-to* (*l1* @ *l2*) (*normalize* (*l1′* @ *l2′*)).
qed

1.5 Normalization preserves generators

Somewhat obvious, but still required to formalize Free Groups, is the fact that canceling a word of generators of a specific set (and their inverses) results in a word in generators from that set.

lemma *cancels-to-1-preserves-generators*:
  assumes *cancels-to-1 l l'*
    and $l \in$ *lists* (*UNIV* $\times$ *gens*)
  shows $l' \in$ *lists* (*UNIV* $\times$ *gens*)
proof−
  from *assms* **obtain** *i* **where** l' = *cancel-at i l*
    unfolding *cancels-to-1-def* **and** *cancels-to-1-at-def* **by** *auto*
  hence l' = *take i l* @ *drop* (*2* + *i*) *l* **unfolding** *cancel-at-def* **.**
  hence *set* l' = *set* (*take i l* @ *drop* (*2* + *i*) *l*) **by** *simp*
  moreover
  have ... = *set* (*take i l* @ *drop* (*2* + *i*) *l*) **by** *auto*
  moreover
  have ... $\subseteq$ *set* (*take i l*) $\cup$ *set* (*drop* (*2* + *i*) *l*) **by** *auto*
  moreover
  have ... $\subseteq$ *set l* **by** (*auto dest*: *in-set-takeD in-set-dropD*)
  ultimately
  have *set* $l' \subseteq$ *set l* **by** *simp*
  thus *?thesis* **using** *assms*(*2*) **by** *auto*
qed

lemma *cancels-to-preserves-generators*:
  assumes *cancels-to l l'*
    and $l \in$ *lists* (*UNIV* $\times$ *gens*)
  shows $l' \in$ *lists* (*UNIV* $\times$ *gens*)
using *assms* **unfolding** *cancels-to-def* **by** (*induct*, *auto dest*:*cancels-to-1-preserves-generators*)

lemma *normalize-preserves-generators*:
  assumes $l \in$ *lists* (*UNIV* $\times$ *gens*)
    shows *normalize l* $\in$ *lists* (*UNIV* $\times$ *gens*)
proof−
  have *cancels-to l* (*normalize l*) **by** *simp*
  thus *?thesis* **using** *assms* **by**(*rule cancels-to-preserves-generators*)
qed

Two simplification lemmas about lists.

lemma *empty-in-lists*[*simp*]:
  [] $\in$ *lists A* **by** *auto*

lemma *lists-empty*[*simp*]: *lists* {} = {[]}
  by *auto*

1.6 Normalization and renaming generators

Renaming the generators, i.e. mapping them through an injective function, commutes with normalization. Similarly, replacing generators by their inverses and vica-versa commutes with normalization. Both operations are similar enough to be handled at once here.

lemma *rename-gens-cancel-at*: *cancel-at i* (*map f l*) = *map f* (*cancel-at i l*)
unfolding *cancel-at-def* **by** (*auto simp add*:*take-map drop-map*)

lemma *rename-gens-cancels-to-1*:
assumes *inj f*
and *cancels-to-1 l l′*
shows *cancels-to-1* (*map* (*map-prod f g*) *l*) (*map* (*map-prod f g*) *l′*)
proof−
from ‹*cancels-to-1 l l′*›
obtain *ls1 l1 l2 ls2*
where *l* = *ls1* @ *l1* # *l2* # *ls2*
and *l′* = *ls1* @ *ls2*
and *canceling l1 l2*
by (*rule cancels-to-1-unfold*)

from ‹*canceling l1 l2*›
have *fst l1* ≠ *fst l2* **and** *snd l1* = *snd l2*
unfolding *canceling-def* **by** *auto*
from ‹*fst l1* ≠ *fst l2*› **and** ‹*inj f*›
have *f* (*fst l1*) ≠ *f* (*fst l2*) **by**(*auto dest*!:*inj-on-contraD*)
hence *fst* (*map-prod f g l1*) ≠ *fst* (*map-prod f g l2*) **by** *auto*
moreover
from ‹*snd l1* = *snd l2*›
have *snd* (*map-prod f g l1*) = *snd* (*map-prod f g l2*) **by** *auto*
ultimately
have *canceling* (*map-prod f g* (*l1*)) (*map-prod f g* (*l2*))
unfolding *canceling-def* **by** *auto*
hence *cancels-to-1* (*map* (*map-prod f g*) *ls1* @ *map-prod f g l1* # *map-prod f g l2* # *map* (*map-prod f g*) *ls2*) (*map* (*map-prod f g*) *ls1* @ *map* (*map-prod f g*) *ls2*)
by(*rule cancels-to-1-fold*)
with ‹*l* = *ls1* @ *l1* # *l2* # *ls2*› **and** ‹*l′* = *ls1* @ *ls2*›
show *cancels-to-1* (*map* (*map-prod f g*) *l*) (*map* (*map-prod f g*) *l′*)
by *simp*
qed

lemma *rename-gens-cancels-to*:
assumes *inj f*
and *cancels-to l l′*
shows *cancels-to* (*map* (*map-prod f g*) *l*) (*map* (*map-prod f g*) *l′*)
using ‹*cancels-to l l′*›
unfolding *cancels-to-def*
proof(*induct rule*:*rtranclp-induct*)
case (*step x z*)

from ‹*cancels-to-1 x z*› **and** ‹*inj f*›
have *cancels-to-1* (*map* (*map-prod f g*) *x*) (*map* (*map-prod f g*) *z*)
by −(*rule rename-gens-cancels-to-1*)
with ‹*cancels-to-1*$\hat{}$∗∗ (*map* (*map-prod f g*) *l*) (*map* (*map-prod f g*) *x*)›
show *cancels-to-1*$\hat{}$∗∗ (*map* (*map-prod f g*) *l*) (*map* (*map-prod f g*) *z*) **by** *auto*
qed(*auto*)

lemma *rename-gens-canceled*:
assumes *inj-on g* (*snd*‘*set l*)
and *canceled l*
shows *canceled* (*map* (*map-prod f g*) *l*)
unfolding *canceled-def*
proof

have *different-images*: $\bigwedge$ *f a b*. *f a* ≠ *f b* ⟹ *a* ≠ *b* **by** *auto*

assume *Domainp cancels-to-1* (*map* (*map-prod f g*) *l*)
then obtain *l'* **where** *cancels-to-1* (*map* (*map-prod f g*) *l*) *l'* **by** *auto*
then obtain *i* **where** *Suc i* < *length l*
and *canceling* (*map* (*map-prod f g*) *l* ! *i*) (*map* (*map-prod f g*) *l* ! *Suc i*)
by(*auto simp add*:*cancels-to-1-def cancels-to-1-at-def*)
hence *f* (*fst* (*l* ! *i*)) ≠ *f* (*fst* (*l* ! *Suc i*))
and *g* (*snd* (*l* ! *i*)) = *g* (*snd* (*l* ! *Suc i*))
by(*auto simp add*:*canceling-def*)
from ‹*f* (*fst* (*l* ! *i*)) ≠ *f* (*fst* (*l* ! *Suc i*))›
have *fst* (*l* ! *i*) ≠ *fst* (*l* ! *Suc i*) **by** −(*erule different-images*)
moreover
from ‹*Suc i* < *length l*›
have *snd* (*l* ! *i*) ∈ *snd* ‘ *set l* **and** *snd* (*l* ! *Suc i*) ∈ *snd* ‘ *set l* **by** *auto*
with ‹*g* (*snd* (*l* ! *i*)) = *g* (*snd* (*l* ! *Suc i*))›
have *snd* (*l* ! *i*) = *snd* (*l* ! *Suc i*)
using ‹*inj-on g* (*image snd* (*set l*))›
by (*auto dest*: *inj-onD*)
ultimately
have *canceling* (*l* ! *i*) (*l* ! *Suc i*) **unfolding** *canceling-def* **by** *simp*
with ‹*Suc i* < *length l*›
have *cancels-to-1-at i l* (*cancel-at i l*)
unfolding *cancels-to-1-at-def* **by** *auto*
hence *cancels-to-1 l* (*cancel-at i l*)
unfolding *cancels-to-1-def* **by** *auto*
hence ¬*canceled l*
unfolding *canceled-def* **by** *auto*
with ‹*canceled l*› **show** *False* **by** *contradiction*
qed

lemma *rename-gens-normalize*:
assumes *inj f*
and *inj-on g* (*snd* ‘ *set l*)

shows *normalize* (*map* (*map-prod f g*) *l*) = *map* (*map-prod f g*) (*normalize l*)
proof(*rule normalize-discover*)
  from ‹*inj-on g* (*image snd* (*set l*))›
  have *inj-on g* (*image snd* (*set* (*normalize l*)))
  proof (*rule subset-inj-on*)

    have *UNIV-snd*: $\bigwedge A.\ A \subseteq UNIV \times snd\ `\ A$
      proof fix *A* **and** $x::'c \times 'd$ **assume** $x \in A$
        hence $(fst\ x, snd\ x) \in (UNIV \times snd\ `\ A)$
          by −(*rule*, *auto*)
        thus $x \in (UNIV \times snd\ `\ A)$ **by** *simp*
      qed

    have $l \in lists\ (set\ l)$ **by** *auto*
    hence $l \in lists\ (UNIV \times snd\ `\ set\ l)$
      by (*rule subsetD*[*OF lists-mono*[*OF UNIV-snd*], *of l set l*])
    hence *normalize* $l \in lists\ (UNIV \times snd\ `\ set\ l)$
      by (*rule normalize-preserves-generators*[*of - snd ‘ set l*])
    thus $snd\ `\ set\ (normalize\ l) \subseteq snd\ `\ set\ l$
      by (*auto simp add*: *lists-eq-set*)
  qed
  thus *canceled* (*map* (*map-prod f g*) (*normalize l*)) **by**(*rule rename-gens-canceled*,*simp*)
next
  from ‹*inj f*›
  show *cancels-to* (*map* (*map-prod f g*) *l*) (*map* (*map-prod f g*) (*normalize l*))
    by (*rule rename-gens-cancels-to*, *simp*)
qed

end

2 Generators

theory *Generators*
imports
  HOL−Algebra.Group
  HOL−Algebra.Lattice
begin

This theory is not specific to Free Groups and could be moved to a more general place. It defines the subgroup generated by a set of generators and that homomorphisms agree on the generated subgroup if they agree on the generators.

notation *subgroup* (**infix** ‹≤› *80*)

2.1 The subgroup generated by a set

The span of a set of subgroup generators, i.e. the generated subgroup, can be defined inductively or as the intersection of all subgroups containing the generators. Here, we define it inductively and proof the equivalence

inductive-set *gen-span* :: $('a,'b)$ *monoid-scheme* $\Rightarrow$ $'a$ *set* $\Rightarrow$ $'a$ *set* (‹⟨-⟩ı›)
  for G **and** *gens*
where *gen-one* [*intro*!, *simp*]: $\mathbf{1}_G \in \langle gens\rangle_G$
  | *gen-gens*: $x \in gens \Longrightarrow x \in \langle gens\rangle_G$
  | *gen-inv*: $x \in \langle gens\rangle_G \Longrightarrow inv_G\ x \in \langle gens\rangle_G$
  | *gen-mult*: $[\![\ x \in \langle gens\rangle_G;\ y \in \langle gens\rangle_G\ ]\!] \Longrightarrow\ x \otimes_G y \in \langle gens\rangle_G$

lemma (**in** *group*) *gen-span-closed*:
  assumes $gens \subseteq carrier\ G$
  shows $\langle gens\rangle_G \subseteq carrier\ G$
proof
  fix x
  from *assms* **show** $x \in \langle gens\rangle_G \Longrightarrow x \in carrier\ G$
    by $-$(*induct rule*:*gen-span.induct*, *auto*)
qed

lemma (**in** *group*) *gen-subgroup-is-subgroup*:
    $gens \subseteq carrier\ G \Longrightarrow \langle gens\rangle_G \leq G$
by(*rule subgroupI*)(*auto intro*:*gen-span.intros simp add*:*gen-span-closed*)

lemma (**in** *group*) *gen-subgroup-is-smallest-containing*:
  assumes $gens \subseteq carrier\ G$
    shows $\bigcap\{H.\ H \leq G \wedge gens \subseteq H\} = \langle gens\rangle_G$
proof
  show $\langle gens\rangle_G \subseteq \bigcap\{H.\ H \leq G \wedge gens \subseteq H\}$
  proof(*rule Inf-greatest*)
    fix H
    assume $H \in \{H.\ H \leq G \wedge gens \subseteq H\}$
    hence $H \leq G$ **and** $gens \subseteq H$ **by** *auto*
    show $\langle gens\rangle_G \subseteq H$
    proof
      fix x
      from ‹$H \leq G$› **and** ‹$gens \subseteq H$›
      show $x \in \langle gens\rangle_G \Longrightarrow x \in H$
        unfolding *subgroup-def*
        by $-$(*induct rule*:*gen-span.induct*, *auto*)
    qed
  qed
next
  from ‹$gens \subseteq carrier\ G$›
  have $\langle gens\rangle_G \leq G$ **by** (*rule gen-subgroup-is-subgroup*)
  moreover
  have $gens \subseteq \langle gens\rangle_G$ **by** (*auto intro*:*gen-span.intros*)
  ultimately
  show $\bigcap\{H.\ H \leq G \wedge gens \subseteq H\} \subseteq \langle gens\rangle_G$
    by(*auto intro*:*Inter-lower*)
qed

2.2 Generators and homomorphisms

Two homorphisms agreeing on some elements agree on the span of those elements.

lemma *hom-unique-on-span*:
  assumes *group G*
    and *group H*
    and $gens \subseteq carrier\ G$
    and $h \in hom\ G\ H$
    and $h' \in hom\ G\ H$
    and $\forall g \in gens.\ h\ g = h'\ g$
  shows $\forall x \in \langle gens \rangle_G.\ h\ x = h'\ x$
proof
  interpret *G*: *group G* **by** *fact*
  interpret *H*: *group H* **by** *fact*
  interpret *h*: *group-hom G H h* **by** *unfold-locales fact*
  interpret *h'*: *group-hom G H h'* **by** *unfold-locales fact*

  fix *x*
  from ‹$gens \subseteq carrier\ G$› **have** $\langle gens \rangle_G \subseteq carrier\ G$ **by** (*rule G.gen-span-closed*)
  with *assms* **show** $x \in \langle gens \rangle_G \Longrightarrow h\ x = h'\ x$ **apply** −
  proof(*induct rule*:*gen-span.induct*)
    case (*gen-mult x y*)
      hence *x*: $x \in carrier\ G$ **and** *y*: $y \in carrier\ G$ **and**
        hx: $h\ x = h'\ x$ **and** *hy*: $h\ y = h'\ y$ **by** *auto*
      thus $h\ (x \otimes_G y) = h'\ (x \otimes_G y)$ **by** *simp*
  qed *auto*
qed

2.3 Sets of generators

There is no definition for "*gens* is a generating set of *G*". This is easily expressed by $\langle gens \rangle = carrier\ G$.

The following is an application of *hom-unique-on-span* on a generating set of the whole group.

lemma (**in** *group*) *hom-unique-by-gens*:
  assumes *group H*
    and *gens*: $\langle gens \rangle_G = carrier\ G$
    and $h \in hom\ G\ H$
    and $h' \in hom\ G\ H$
    and $\forall g \in gens.\ h\ g = h'\ g$
  shows $\forall x \in carrier\ G.\ h\ x = h'\ x$
proof
  fix *x*

  from *gens* **have** $gens \subseteq carrier\ G$ **by** (*auto intro*:*gen-span.gen-gens*)
  with *assms* **and** *group-axioms* **have** *r*: $\forall x \in \langle gens \rangle_G.\ h\ x = h'\ x$
    by −(*erule hom-unique-on-span*, *auto*)

```
  with gens show x ∈ carrier G ⟹ h x = h' x by auto
qed

lemma (in group-hom) hom-span:
  assumes gens ⊆ carrier G
  shows h ‘ (⟨gens⟩_G) = ⟨h ‘ gens⟩_H
proof(rule Set.set-eqI, rule iffI)
  from ‹gens ⊆ carrier G›
  have ⟨gens⟩_G ⊆ carrier G by (rule G.gen-span-closed)

  fix y
  assume y ∈ h ‘ ⟨gens⟩_G
  then obtain x where x ∈ ⟨gens⟩_G and y = h x by auto
  from ‹x ∈ ⟨gens⟩_G›
  have h x ∈ ⟨h ‘ gens⟩_H
  proof(induct x)
    case (gen-inv x)
    hence x ∈ carrier G and h x ∈ ⟨h ‘ gens⟩_H
      using ‹⟨gens⟩_G ⊆ carrier G›
      by auto
    thus ?case by (auto intro:gen-span.intros)
  next
    case (gen-mult x y)
    hence x ∈ carrier G and h x ∈ ⟨h ‘ gens⟩_H
    and   y ∈ carrier G and h y ∈ ⟨h ‘ gens⟩_H
      using ‹⟨gens⟩_G ⊆ carrier G›
      by auto
    thus ?case by (auto intro:gen-span.intros)
  qed(auto intro: gen-span.intros)
  with ‹y = h x›
  show y ∈ ⟨h ‘ gens⟩_H by simp
next
  fix x
  show x ∈ ⟨h ‘ gens⟩_H ⟹ x ∈ h ‘ ⟨gens⟩
  proof(induct x rule:gen-span.induct)
    case (gen-inv y)
      then  obtain x where y = h x and x ∈ ⟨gens⟩ by auto
      moreover
      hence x ∈ carrier G  using ‹gens ⊆ carrier G›
        by (auto dest:G.gen-span-closed)
      ultimately show ?case
          by (auto intro:hom-inv[THEN sym] rev-image-eqI gen-span.gen-inv simp
del:group-hom.hom-inv hom-inv)
  next
   case (gen-mult y y')
      then  obtain x and x'
        where y = h x and x ∈ ⟨gens⟩
        and y' = h x' and x' ∈ ⟨gens⟩ by auto
      moreover
```

hence *x* ∈ *carrier G* **and** *x′* ∈ *carrier G* **using** ‹*gens* ⊆ *carrier G*›
by (*auto dest*:*G.gen-span-closed*)
ultimately show *?case*
by (*auto intro*:*hom-mult*[*THEN sym*] *rev-image-eqI gen-span.gen-mult simp del*:*group-hom.hom-mult hom-mult*)
qed(*auto intro*:*rev-image-eqI intro*:*gen-span.intros*)
qed

2.4 Product of a list of group elements

Not strictly related to generators of groups, this is still a general group concept and not related to Free Groups.

abbreviation (**in** *monoid*) *m-concat*
where *m-concat l* ≡ *foldr* (⊗) *l* **1**

lemma (**in** *monoid*) *m-concat-closed*[*simp*]:
set l ⊆ *carrier G* ⟹ *m-concat l* ∈ *carrier G*
by (*induct l*, *auto*)

lemma (**in** *monoid*) *m-concat-append*[*simp*]:
assumes *set a* ⊆ *carrier G*
and *set b* ⊆ *carrier G*
shows *m-concat* (*a*@*b*) = *m-concat a* ⊗ *m-concat b*
using *assms*
by(*induct a*)(*auto simp add*: *m-assoc*)

lemma (**in** *monoid*) *m-concat-cons*[*simp*]:
⟦ *x* ∈ *carrier G* ; *set xs* ⊆ *carrier G* ⟧ ⟹ *m-concat* (*x*#*xs*) = *x* ⊗ *m-concat xs*
by(*induct xs*)(*auto simp add*: *m-assoc*)

lemma (**in** *monoid*) *nat-pow-mult1l*:
assumes *x*: *x* ∈ *carrier G*
shows *x* ⊗ *x* [^] *n* = *x* [^] *Suc n*
proof−
have *x* ⊗ *x* [^] *n* = *x* [^] (*1*::*nat*) ⊗ *x* [^] *n* **using** *x* **by** *auto*
also have . . . = *x* [^] (*1* + *n*) **using** *x*
by (*auto dest*:*nat-pow-mult simp del*:*One-nat-def*)
also have . . . = *x* [^] *Suc n* **by** *simp*
finally show *x* ⊗ *x* [^] *n* = *x* [^] *Suc n* **.**
qed

lemma (**in** *monoid*) *m-concat-power*[*simp*]: *x* ∈ *carrier G* ⟹ *m-concat* (*replicate n x*) = *x* [^] *n*
by(*induct n*, *auto simp add*:*nat-pow-mult1l*)

2.5 Isomorphisms

A nicer way of proving that something is a group homomorphism or isomorphism.

lemma *group-homI*[*intro*]:
  assumes *range*: h ‘ $(carrier\ g1) \subseteq carrier\ g2$
    and *hom*: $\forall x\in carrier\ g1.\ \forall y\in carrier\ g1.\ h\ (x \otimes_{g1} y) = h\ x \otimes_{g2} h\ y$
  shows $h \in hom\ g1\ g2$
proof−
  have $h \in carrier\ g1 \rightarrow carrier\ g2$ **using** *range* **by** *auto*
  thus $h \in hom\ g1\ g2$ **using** *hom* **unfolding** *hom-def* **by** *auto*
qed

lemma (**in** *group-hom*) *hom-injI*:
  assumes $\forall x\in carrier\ G.\ h\ x = \mathbf{1}_H \longrightarrow x = \mathbf{1}_G$
  shows *inj-on* h $(carrier\ G)$
unfolding *inj-on-def*
proof(*rule ballI*, *rule ballI*, *rule impI*)
  fix x
  fix y
  assume x: $x\in carrier\ G$
    and y: $y\in carrier\ G$
    and $h\ x = h\ y$
  hence $h\ (x \otimes inv\ y) = \mathbf{1}_H$ **and** $x \otimes inv\ y \in carrier\ G$
    by *auto*
  with *assms*
  have $x \otimes inv\ y = \mathbf{1}$ **by** *auto*
  thus $x = y$ **using** x **and** y
    by(*auto dest*: *G.inv-equality*)
qed

lemma (**in** *group-hom*) *group-hom-isoI*:
  assumes *inj1*: $\forall x\in carrier\ G.\ h\ x = \mathbf{1}_H \longrightarrow x = \mathbf{1}_G$
    and *surj*: h ‘ $(carrier\ G) = carrier\ H$
  shows $h \in iso\ G\ H$
proof−
  from *inj1*
  have *inj-on* h $(carrier\ G)$
    by(*auto intro*: *hom-injI*)
  hence *bij*: *bij-betw* h $(carrier\ G)$ $(carrier\ H)$
    using *surj* **unfolding** *bij-betw-def* **by** *auto*
  thus *?thesis*
    unfolding *iso-def* **by** *auto*
qed

lemma *group-isoI*[*intro*]:
  assumes G: *group* G
    and H: *group* H
    and *inj1*: $\forall x\in carrier\ G.\ h\ x = \mathbf{1}_H \longrightarrow x = \mathbf{1}_G$

and *surj*: h ` (*carrier* G) = *carrier* H
and *hom*: $\forall\, x \in$ *carrier* G. $\forall\, y \in$ *carrier* G. $h\ (x \otimes_G y) = h\ x \otimes_H h\ y$
shows $h \in$ *iso* $G\ H$
proof−
from *surj*
have $h \in$ *carrier* $G \to$ *carrier* H
by *auto*
then interpret *group-hom* $G\ H\ h$ **using** G **and** H **and** *hom*
by (*auto intro*!: *group-hom.intro group-hom-axioms.intro*)
show *?thesis*
using *assms* **unfolding** *hom-def* **by** (*auto intro*: *group-hom-isoI*)
qed
end

3 The Free Group

theory *FreeGroups*
imports
HOL−Algebra.Group
Cancelation
Generators
begin

Based on the work in *Free−Groups.Cancelation*, the free group is now easily defined over the set of fully canceled words with the corresponding operations.

3.1 Inversion

To define the inverse of a word, we first create a helper function that inverts a single generator, and show that it is self-inverse.

definition *inv1* :: *'a g-i* ⇒ *'a g-i*
where *inv1* = *apfst Not*

lemma *inv1-inv1*: *inv1* ∘ *inv1* = *id*
by (*simp add*: *fun-eq-iff comp-def inv1-def*)

lemmas *inv1-inv1-simp* [*simp*] = *inv1-inv1*[*unfolded id-def*]

lemma *snd-inv1*: *snd* ∘ *inv1* = *snd*
by(*simp add*: *fun-eq-iff comp-def inv1-def*)

The inverse of a word is obtained by reversing the order of the generators and inverting each generator using *inv1*. Some properties of *inv-fg* are noted.

definition *inv-fg* :: *'a word-g-i* ⇒ *'a word-g-i*
where *inv-fg l* = *rev* (*map inv1 l*)

lemma *cancelling-inf*[*simp*]: *canceling* (*inv1 a*) (*inv1 b*) = *canceling a b*

```
  by(simp add: canceling-def inv1-def)

lemma inv-idemp: inv-fg (inv-fg l) = l
  by (auto simp add:inv-fg-def rev-map)

lemma inv-fg-cancel: normalize (l @ inv-fg l) = []
proof(induct l rule:rev-induct)
  case Nil thus ?case
    by (auto simp add: inv-fg-def)
next
  case (snoc x xs)
  have canceling x (inv1 x) by (simp add:inv1-def canceling-def)
  moreover
  let ?i = length xs
  have Suc ?i < length xs + 1 + 1 + length xs
    by auto
  moreover
  have inv-fg (xs @ [x]) = [inv1 x] @ inv-fg xs
    by (auto simp add:inv-fg-def)
  ultimately
  have cancels-to-1-at ?i (xs @ [x] @ (inv-fg (xs @ [x]))) (xs @ inv-fg xs)
    by (auto simp add:cancels-to-1-at-def cancel-at-def nth-append)
  hence cancels-to-1 (xs @ [x] @ (inv-fg (xs @ [x]))) (xs @ inv-fg xs)
    by (auto simp add: cancels-to-1-def)
  hence cancels-to (xs @ [x] @ (inv-fg (xs @ [x]))) (xs @ inv-fg xs)
    by (auto simp add:cancels-to-def)
  with ‹normalize (xs @ (inv-fg xs)) = []›
  show normalize ((xs @ [x]) @ (inv-fg (xs @ [x]))) = []
    by auto
qed

lemma inv-fg-cancel2: normalize (inv-fg l @ l) = []
proof−
  have normalize (inv-fg l @ inv-fg (inv-fg l)) = [] by (rule inv-fg-cancel)
  thus normalize (inv-fg l @ l) = [] by (simp add: inv-idemp)
qed

lemma canceled-rev:
  assumes canceled l
  shows canceled (rev l)
proof(rule ccontr)
  assume ¬canceled (rev l)
  hence Domainp cancels-to-1 (rev l) by (simp add: canceled-def)
  then obtain l′ where cancels-to-1 (rev l) l′ by auto
  then obtain i where cancels-to-1-at i (rev l) l′ by (auto simp add:cancels-to-1-def)
  hence Suc i < length (rev l)
    and canceling (rev l ! i) (rev l ! Suc i)
    by (auto simp add:cancels-to-1-at-def)
  let ?x = length l − i − 2
```

from ‹*Suc i* < *length* (*rev l*)›
have *Suc ?x* < *length l* **by** *auto*
moreover
from ‹*Suc i* < *length* (*rev l*)›
have *i* < *length l* **and** *length l* − *Suc i* = *Suc*(*length l* − *Suc* (*Suc i*)) **by** *auto*
hence *rev l* ! *i* = *l* ! *Suc ?x* **and** *rev l* ! *Suc i* = *l* ! *?x*
by (*auto simp add*: *rev-nth map-nth*)
with ‹*canceling* (*rev l* ! *i*) (*rev l* ! *Suc i*)›
have *canceling* (*l* ! *Suc ?x*) (*l* ! *?x*) **by** *auto*
hence *canceling* (*l* ! *?x*) (*l* ! *Suc ?x*) **by** (*rule cancel-sym*)
hence *canceling* (*l* ! *?x*) (*l* ! *Suc ?x*) **by** *simp*
ultimately
have *cancels-to-1-at ?x l* (*cancel-at ?x l*)
by (*auto simp add*:*cancels-to-1-at-def*)
hence *cancels-to-1 l* (*cancel-at ?x l*)
by (*auto simp add*:*cancels-to-1-def*)
hence ¬*canceled l*
by (*auto simp add*:*canceled-def*)
with ‹*canceled l*› **show** *False* **by** *contradiction*
qed

lemma *inv-fg-closure1*:
assumes *canceled l*
shows *canceled* (*inv-fg l*)
unfolding *inv-fg-def* **and** *inv1-def* **and** *apfst-def*
proof−
have *inj Not* **by** (*auto intro*:*injI*)
moreover
have *inj-on id* (*snd* ‘ *set l*) **by** *auto*
ultimately
have *canceled* (*map* (*map-prod Not id*) *l*)
using ‹*canceled l*›
by −(*rule rename-gens-canceled*)
thus *canceled* (*rev* (*map* (*map-prod Not id*) *l*)) **by** (*rule canceled-rev*)
qed

lemma *inv-fg-closure2*:
l ∈ *lists* (*UNIV* × *gens*) ⟹ *inv-fg l* ∈ *lists* (*UNIV* × *gens*)
by (*auto iff*:*lists-eq-set simp add*:*inv1-def inv-fg-def*)

3.2 The definition

Finally, we can define the Free Group over a set of generators, and show that it is indeed a group.

definition *free-group* :: *'a set* => ((*bool* ∗ *'a*) *list*) *monoid* (‹$\mathcal{F}$ı›)
where
$\mathcal{F}_{gens}$ ≡ ⦇
carrier = {*l*∈*lists* (*UNIV* × *gens*). *canceled l* },
mult = λ *x y*. *normalize* (*x* @ *y*),

$one = []$
$\rrparenthesis$

lemma *occuring-gens-in-element*:
$x \in carrier\ \mathcal{F}_{gens} \Longrightarrow x \in lists\ (UNIV \times gens)$
by(*auto simp add*:*free-group-def*)

theorem *free-group-is-group*: *group* $\mathcal{F}_{gens}$
proof
fix $x\ y$
assume $x \in carrier\ \mathcal{F}_{gens}$ **hence** x: $x \in lists\ (UNIV \times gens)$ **by**
(*rule occuring-gens-in-element*)
assume $y \in carrier\ \mathcal{F}_{gens}$ **hence** y: $y \in lists\ (UNIV \times gens)$ **by**
(*rule occuring-gens-in-element*)
from x **and** y
have $x \otimes_{\mathcal{F}_{gens}} y \in lists\ (UNIV \times gens)$
by (*auto intro*!: *normalize-preserves-generators simp add*:*free-group-def append-in-lists-conv*)
thus $x \otimes_{\mathcal{F}_{gens}} y \in carrier\ \mathcal{F}_{gens}$
by (*auto simp add*:*free-group-def*)
next
fix $x\ y\ z$
have *cancels-to* (x @ y) (*normalize* (x @ (y::'*a word-g-i*)))
and *cancels-to* z (z::'*a word-g-i*)
by *auto*
hence *normalize* (*normalize* (x @ y) @ z) = *normalize* ((x @ y) @ z)
by (*rule normalize-append-cancel-to*[*THEN sym*])
also
have ... = *normalize* (x @ (y @ z)) **by** *auto*
also
have *cancels-to* (y @ z) (*normalize* (y @ (z::'*a word-g-i*)))
and *cancels-to* x (x::'*a word-g-i*)
by *auto*
hence *normalize* (x @ (y @ z)) = *normalize* (x @ *normalize* (y @ z))
by −(*rule normalize-append-cancel-to*)
finally
show $x \otimes_{\mathcal{F}_{gens}} y \otimes_{\mathcal{F}_{gens}} z =$
$x \otimes_{\mathcal{F}_{gens}} (y \otimes_{\mathcal{F}_{gens}} z)$
by (*auto simp add*:*free-group-def*)
next
show $\mathbf{1}_{\mathcal{F}_{gens}} \in carrier\ \mathcal{F}_{gens}$
by (*auto simp add*:*free-group-def*)
next
fix x
assume $x \in carrier\ \mathcal{F}_{gens}$
thus $\mathbf{1}_{\mathcal{F}_{gens}} \otimes_{\mathcal{F}_{gens}} x = x$
by (*auto simp add*:*free-group-def*)

next
  fix x
  assume $x \in carrier\ \mathcal{F}_{gens}$
  thus $x \otimes_{\mathcal{F}_{gens}} \mathbf{1}_{\mathcal{F}_{gens}} = x$
    by (*auto simp add:free-group-def*)
next
  show $carrier\ \mathcal{F}_{gens} \subseteq Units\ \mathcal{F}_{gens}$
  proof (*simp add:free-group-def Units-def*, *rule subsetI*)
    fix x :: *'a word-g-i*
    let $?x' = inv\text{-}fg\ x$
    assume $x \in \{y{\in}lists(UNIV{\times}gens).\ canceled\ y\}$
    hence $?x' \in lists(UNIV{\times}gens) \wedge canceled\ ?x'$
      by (*auto elim:inv-fg-closure1 simp add:inv-fg-closure2*)
    moreover
    have $normalize\ (?x'\ @\ x) = []$
     and $normalize\ (x\ @\ ?x') = []$
      by (*auto simp add:inv-fg-cancel inv-fg-cancel2*)
    ultimately
    have $\exists\, y.\ y \in lists\ (UNIV\ \times\ gens) \wedge$
              $canceled\ y \wedge$
              $normalize\ (y\ @\ x) = [] \wedge normalize\ (x\ @\ y) = []$
      by *auto*
    with ‹$x \in \{y{\in}lists(UNIV{\times}gens).\ canceled\ y\}$›
    show $x \in \{y \in lists\ (UNIV\ \times\ gens).\ \ canceled\ y\ \ \wedge$
         $(\exists\, x.\ x \in lists\ (UNIV\ \times\ gens) \wedge$
              $canceled\ x \wedge$
              $normalize\ (x\ @\ y) = [] \wedge normalize\ (y\ @\ x) = [])\}$
      by *auto*
  qed
qed

lemma *inv-is-inv-fg*[*simp*]:
  $x \in carrier\ \mathcal{F}_{gens} \Longrightarrow inv_{\mathcal{F}_{gens}}\ x = inv\text{-}fg\ x$
by (*rule group.inv-equality*,*auto simp add:free-group-is-group*,*auto simp add*: *free-group-def inv-fg-cancel inv-fg-cancel2 inv-fg-closure1 inv-fg-closure2*)

3.3 The universal property

Free Groups are important due to their universal property: Every map of the set of generators to another group can be extended uniquely to an homomorphism from the Free Group.

definition *insert* (‹ι›)
  where $\iota\ g = [(False,\ g)]$

lemma *insert-closed*:
  $g \in gens \Longrightarrow \iota\ g \in carrier\ \mathcal{F}_{gens}$
  by (*auto simp add:insert-def free-group-def*)

```
definition (in group) lift-gi
  where lift-gi f gi = (if fst gi then inv (f (snd gi)) else f (snd gi))

lemma (in group) lift-gi-closed:
  assumes cl: f ∈ gens → carrier G
      and snd gi ∈ gens
  shows lift-gi f gi ∈ carrier G
using assms by (auto simp add:lift-gi-def)

definition (in group) lift
  where lift f w = m-concat (map (lift-gi f) w)

lemma (in group) lift-nil[simp]: lift f [] = 1
 by (auto simp add:lift-def)

lemma (in group) lift-closed[simp]:
  assumes cl: f ∈ gens → carrier G
      and x ∈ lists (UNIV × gens)
  shows lift f x ∈ carrier G
proof−
  have set (map (lift-gi f) x) ⊆ carrier G
    using ‹x ∈ lists (UNIV × gens)›
    by (auto simp add:lift-gi-closed[OF cl])
  thus lift f x ∈ carrier G
    by (auto simp add:lift-def)
qed

lemma (in group) lift-append[simp]:
  assumes cl: f ∈ gens → carrier G
      and x ∈ lists (UNIV × gens)
      and y ∈ lists (UNIV × gens)
  shows lift f (x @ y) = lift f x ⊗ lift f y
proof−
  from ‹x ∈ lists (UNIV × gens)›
  have set (map snd x) ⊆ gens by auto
  hence set (map (lift-gi f) x) ⊆ carrier G
    by (induct x)(auto simp add:lift-gi-closed[OF cl])
  moreover
  from ‹y ∈ lists (UNIV × gens)›
  have set (map snd y) ⊆ gens by auto
  hence set (map (lift-gi f) y) ⊆ carrier G
    by (induct y)(auto simp add:lift-gi-closed[OF cl])
  ultimately
  show lift f (x @ y) = lift f x ⊗ lift f y
    by (auto simp add:lift-def m-assoc simp del:set-map foldr-append)
qed

lemma (in group) lift-cancels-to:
  assumes cancels-to x y
```

```
    and x ∈ lists (UNIV × gens)
    and cl: f ∈ gens → carrier G
  shows lift f x = lift f y
using assms
unfolding cancels-to-def
proof(induct rule:rtranclp-induct)
  case (step y z)
    from ‹cancels-to-1** x y›
    and ‹x ∈ lists (UNIV × gens)›
    have y ∈ lists (UNIV × gens)
      by −(rule cancels-to-preserves-generators, simp add:cancels-to-def)
    hence lift f x = lift f y
      using step by auto
    also
    from ‹cancels-to-1 y z›
    obtain ys1 y1 y2 ys2
      where y: y = ys1 @ y1 # y2 # ys2
      and z = ys1 @ ys2
      and canceling y1 y2
    by (rule cancels-to-1-unfold)
    have lift f y  = lift f (ys1 @ [y1] @ [y2] @ ys2)
      using y by simp
    also
    from y and cl and ‹y ∈ lists (UNIV × gens)›
    have lift f (ys1 @ [y1] @ [y2] @ ys2)
        = lift f ys1 ⊗ (lift f [y1] ⊗ lift f [y2]) ⊗ lift f ys2
       by (auto intro:lift-append[OF cl] simp del: append-Cons simp add:m-assoc
iff:lists-eq-set)
    also
    from cl[THEN funcset-image]
     and y and ‹y ∈ lists (UNIV × gens)›
     and ‹canceling y1 y2›
    have (lift f [y1] ⊗ lift f [y2]) = 1
      by (auto simp add:lift-def lift-gi-def canceling-def iff:lists-eq-set)
    hence lift f ys1 ⊗ (lift f [y1] ⊗ lift f [y2]) ⊗ lift f ys2
          = lift f ys1 ⊗ 1 ⊗ lift f ys2
      by simp
    also
    from y and ‹y ∈ lists (UNIV × gens)›
     and cl
     have lift f ys1 ⊗ 1 ⊗ lift f ys2 = lift f (ys1 @ ys2)
      by (auto intro:lift-append iff:lists-eq-set)
    also
    from ‹z = ys1 @ ys2›
    have lift f (ys1 @ ys2) = lift f z by simp
    finally show lift f x = lift f z .
qed auto

lemma (in group) lift-is-hom:
```

```
  assumes cl: f ∈ gens → carrier G
  shows lift f ∈ hom F_gens G
proof−
  {
    fix x
    assume x ∈ carrier F_gens
    hence x ∈ lists (UNIV × gens)
      unfolding free-group-def by simp
    hence lift f x ∈ carrier G
     by (induct x, auto simp add:lift-def lift-gi-closed[OF cl])
  }
  moreover
  { fix x
    assume x ∈ carrier F_gens
    fix y
    assume y ∈ carrier F_gens

    from ‹x ∈ carrier F_gens› and ‹y ∈ carrier F_gens›
    have x ∈ lists (UNIV × gens) and y ∈ lists (UNIV × gens)
      by (auto simp add:free-group-def)

    have cancels-to (x @ y) (normalize (x @ y)) by simp
    from ‹x ∈ lists (UNIV × gens)› and ‹y ∈ lists (UNIV × gens)›
     and lift-cancels-to[THEN sym, OF ‹cancels-to (x @ y) (normalize (x @ y))›]
and cl
    have lift f (x ⊗_F_gens y) = lift f (x @ y)
      by (auto simp add:free-group-def iff:lists-eq-set)
    also
    from ‹x ∈ lists (UNIV × gens)› and ‹y ∈ lists (UNIV × gens)› and cl
    have lift f (x @ y) = lift f x ⊗ lift f y
      by simp
    finally
    have lift f (x ⊗_F_gens y) = lift f x ⊗ lift f y .
  }
  ultimately
  show lift f ∈ hom F_gens G
    by auto
qed

lemma gens-span-free-group:
shows ⟨ι ‘ gens⟩_F_gens = carrier F_gens
proof
  interpret group F_gens by (rule free-group-is-group)
  show ⟨ι ‘ gens⟩_F_gens ⊆ carrier F_gens
  by(rule gen-span-closed, auto simp add:insert-def free-group-def)

  show carrier F_gens ⊆ ⟨ι ‘ gens⟩_F_gens
  proof
    fix x
```

```
show x ∈ carrier ℱ_gens ⟹ x ∈ ⟨ι ` gens⟩_ℱ_gens
proof(induct x)
case Nil
  have one ℱ_gens ∈ ⟨ι ` gens⟩_ℱ_gens
    by simp
  thus [] ∈ ⟨ι ` gens⟩_ℱ_gens
    by (simp add:free-group-def)
next
case (Cons a x)
  from ‹a # x ∈ carrier ℱ_gens›
  have x ∈ carrier ℱ_gens
    by (auto intro:cons-canceled simp add:free-group-def)
  hence x ∈ ⟨ι ` gens⟩_ℱ_gens
    using Cons by simp
  moreover

  from ‹a # x ∈ carrier ℱ_gens›
  have snd a ∈ gens
    by (auto simp add:free-group-def)
  hence isa: ι (snd a) ∈ ⟨ι ` gens⟩_ℱ_gens
    by (auto simp add:insert-def intro:gen-gens)
  have [a] ∈ ⟨ι ` gens⟩_ℱ_gens
  proof(cases fst a)
    case False
      hence [a] = ι (snd a) by (cases a, auto simp add:insert-def)
       with isa show [a] ∈ ⟨ι ` gens⟩_ℱ_gens by simp
   next
    case True
      from ‹snd a ∈ gens›
      have ι (snd a) ∈ carrier ℱ_gens
        by (auto simp add:free-group-def insert-def)
      with True
      have [a] = inv_ℱ_gens (ι (snd a))
        by (cases a, auto simp add:insert-def inv-fg-def inv1-def)
      moreover
      from isa
      have inv_ℱ_gens (ι (snd a)) ∈ ⟨ι ` gens⟩_ℱ_gens
        by (auto intro:gen-inv)
      ultimately
      show [a] ∈ ⟨ι ` gens⟩_ℱ_gens
        by simp
  qed
  ultimately
  have mult ℱ_gens [a] x ∈ ⟨ι ` gens⟩_ℱ_gens
    by (auto intro:gen-mult)
  with
  ‹a # x ∈ carrier ℱ_gens›
  show a # x ∈ ⟨ι ` gens⟩_ℱ_gens by (simp add:free-group-def)
```

```
    qed
  qed
qed

lemma (in group) lift-is-unique:
  assumes group G
  and cl: f ∈ gens → carrier G
  and h ∈ hom F_gens G
  and ∀ g ∈ gens. h (ι g) = f g
  shows ∀ x ∈ carrier F_gens. h x = lift f x
unfolding gens-span-free-group[THEN sym]
proof(rule hom-unique-on-span[of F_gens G])
  show group F_gens by (rule free-group-is-group)
next
  show group G by fact
next
  show ι ‘ gens ⊆ carrier F_gens
    by(auto intro:insert-closed)
next
  show h ∈ hom F_gens G by fact
next
  show lift f ∈ hom F_gens G by (rule lift-is-hom[OF cl])
next
  from ‹∀ g∈ gens. h (ι g) = f g› and cl[THEN funcset-image]
  show ∀ g∈ ι ‘ gens. h g = lift f g
    by(auto simp add:insert-def lift-def lift-gi-def)
qed

end
```

4 The Unit Group

```
theory UnitGroup
imports
   HOL−Algebra.Group
   Generators
begin
```

There is, up to isomorphisms, only one group with one element.

```
definition unit-group :: unit monoid
where
  unit-group ≡ (|
     carrier = UNIV,
     mult = λ x y. (),
     one = ()
  |)

theorem unit-group-is-group: group unit-group
  by (rule groupI, auto simp add:unit-group-def)
```

theorem (**in** *group*) *unit-group-unique*:
  assumes *card* (*carrier G*) = *1*
  shows $\exists$ *h*. *h* $\in$ *iso G unit-group*
proof−
  from *assms* **obtain** *x* **where** *carrier G* = {*x*} **by** (*auto dest*: *card-eq-SucD*)
  hence (λ *x*. ()) $\in$ *iso G unit-group*
   by −(*rule group-isoI*, *auto simp add*:*unit-group-is-group is-group*, *simp add*:*unit-group-def*)
  thus *?thesis* **by** *auto*
qed

end
theory *C2*
imports *HOL*−*Algebra.Group*
begin

5 The group C2

The two-element group is defined over the set of boolean values. This allows to use the equality of boolean values as the group operation.

definition *C2*
  where *C2* = (| *carrier* = *UNIV*, *mult* = (=), *one* = *True* |)

lemma [*simp*]: $(\otimes_{C2}) = (=)$
  unfolding *C2-def* **by** *simp*

lemma [*simp*]: $\mathbf{1}_{C2}$ = *True*
  unfolding *C2-def* **by** *simp*

lemma [*simp*]: *carrier C2* = *UNIV*
  unfolding *C2-def* **by** *simp*

lemma *C2-is-group*: *group C2*
  unfolding *C2-def*
  by (*rule groupI*, *auto simp add*:*Units-def*)

end

6 Isomorphisms of Free Groups

theory *Isomorphisms*
imports
   UnitGroup
   HOL−*Algebra.IntRing*
   FreeGroups
   C2
   HOL−*Cardinals.Cardinal-Order-Relation*
begin

6.1 The Free Group over the empty set

The Free Group over an empty set of generators is isomorphic to the trivial group.

lemma *free-group-over-empty-set*: $\exists h.\ h \in iso\ \mathcal{F}_{\{\}}\ unit\text{-}group$
proof(*rule group.unit-group-unique*)
  show *group* $\mathcal{F}_{\{\}}$ **by** (*rule free-group-is-group*)
next
  have *carrier* $\mathcal{F}_{\{\}::'a\ set} = \{[]\}$
    by (*auto simp add:free-group-def*)
  thus *card* (*carrier* $\mathcal{F}_{\{\}::'a\ set}) = 1$
    by *simp*
qed

6.2 The Free Group over one generator

The Free Group over one generator is isomorphic to the free abelian group over one element, also known as the integers.

abbreviation *int-group*
  where $int\text{-}group \equiv (\!|\ carrier = carrier\ \mathcal{Z},\ monoid.mult = (+),\ one = 0::int\ |\!)$

lemma *replicate-set-eq*[*simp*]: $\forall x \in set\ xs.\ x = y \Longrightarrow xs = replicate\ (length\ xs)\ y$
  by(*induct xs*)*auto*

lemma *int-group-gen-by-one*: $\langle\{1\}\rangle_{int\text{-}group} = carrier\ int\text{-}group$
proof
  show $\langle\{1\}\rangle_{int\text{-}group} \subseteq carrier\ int\text{-}group$
    by *auto*
  show $carrier\ int\text{-}group \subseteq \langle\{1\}\rangle_{int\text{-}group}$
  proof
    interpret *int*: *group int-group*
      using *int.a-group* **by** *auto*
    fix *x*
    have *plus1*: $1 \in \langle\{1\}\rangle_{int\text{-}group}$
      by (*auto intro*:*gen-span.gen-gens*)
    hence $inv_{int\text{-}group}\ 1 \in \langle\{1\}\rangle_{int\text{-}group}$
      by (*auto intro*:*gen-span.gen-inv*)
    moreover
    have $-1 = inv_{int\text{-}group}\ 1$
      by (*rule sym*, *rule int.inv-equality*) *simp-all*
    ultimately
    have *minus1*: $-1 \in \langle\{1\}\rangle_{int\text{-}group}$
      by (*simp*)

    show $x \in \langle\{1::int\}\rangle_{int\text{-}group}$
    proof(*induct x rule*:*int-induct*[*of - 0::int*])
    case *base*
      have $\mathbf{1}_{int\text{-}group} \in \langle\{1::int\}\rangle_{int\text{-}group}$

```
      by (rule gen-span.gen-one)
    thus 0 ∈ ⟨{1}⟩int-group
      by simp
  next
  case (step1 i)
    from ‹i ∈ ⟨{1}⟩int-group› and plus1
    have i ⊗int-group 1 ∈ ⟨{1}⟩int-group
      by (rule gen-span.gen-mult)
    thus i + 1 ∈ ⟨{1}⟩int-group by simp
  next
  case (step2 i)
    from ‹i ∈ ⟨{1}⟩int-group› and minus1
    have i ⊗int-group −1 ∈ ⟨{1}⟩int-group
      by (rule gen-span.gen-mult)
    thus i − 1 ∈ ⟨{1}⟩int-group
      by simp
  qed
 qed
qed

lemma free-group-over-one-gen: ∃ h. h ∈ iso F{()} int-group
proof−
 interpret int: group int-group
   using int.a-group by auto
 define f :: unit ⇒ int where f x = 1 for x
 have f ∈ {()} → carrier int-group
   by auto
 hence int.lift f ∈ hom F{()} int-group
   by (rule int.lift-is-hom)
 then
 interpret hom: group-hom F{()} int-group int.lift f
   unfolding group-hom-def group-hom-axioms-def
   using int.a-group by(auto intro: free-group-is-group)

 {
   fix x
   assume x ∈ carrier F{()}
   hence canceled x by (auto simp add:free-group-def)
   assume int.lift f x = (0::int)
   have x = []
   proof(rule ccontr)
     assume x ≠ []
     then obtain a and xs where x = a # xs by (cases x, auto)
     hence length (takeWhile (λy. y = a) x) > 0 by auto
     then obtain i where i: length (takeWhile (λy. y = a) x) = Suc i
       by (cases length (takeWhile (λy. y = a) x), auto)
     have Suc i ≥ length x
     proof(rule ccontr)
       assume ¬ length x ≤ Suc i
```

hence *length* (*takeWhile* (λ*y*. *y* = *a*) *x*) < *length x* **using** *i* **by** *simp*
hence ¬ (λ*y*. *y* = *a*) (*x* ! *length* (*takeWhile* (λ*y*. *y* = *a*) *x*))
by (*rule nth-length-takeWhile*)
hence ¬ (λ*y*. *y* = *a*) (*x* ! *Suc i*) **using** *i* **by** *simp*
hence *fst* (*x* ! *Suc i*) ≠ *fst a* **by** (*cases x* ! *Suc i*, *cases a*, *auto*)
moreover
{
have *takeWhile* (λ*y*. *y* = *a*) *x* ! *i* = *x* ! *i*
using *i* **by** (*auto intro*: *takeWhile-nth*)
moreover
have (*takeWhile* (λ*y*. *y* = *a*) *x*) ! *i* ∈ *set* (*takeWhile* (λ*y*. *y* = *a*) *x*)
using *i* **by** *auto*
ultimately
have (λ*y*. *y* = *a*) (*x* ! *i*)
by (*auto dest*:*set-takeWhileD*)
}
hence *fst* (*x* ! *i*) = *fst a* **by** *auto*
moreover
have *snd* (*x* ! *i*) = *snd* (*x* ! *Suc i*) **by** *simp*
ultimately
have *canceling* (*x* ! *i*) (*x* ! *Suc i*) **unfolding** *canceling-def* **by** *auto*
hence *cancels-to-1-at i x* (*cancel-at i x*)
using ‹¬ *length x* ≤ *Suc i*› **unfolding** *cancels-to-1-at-def*
by (*auto simp add*:*length-takeWhile-le*)
hence *cancels-to-1 x* (*cancel-at i x*) **unfolding** *cancels-to-1-def* **by** *auto*
hence ¬ *canceled x* **unfolding** *canceled-def* **by** *auto*
thus *False* **using** ‹*canceled x*› **by** *contradiction*
qed
hence *length* (*takeWhile* (λ*y*. *y* = *a*) *x*) = *length x*
using *i*[*THEN sym*] **by** (*auto dest*:*le-antisym simp add*:*length-takeWhile-le*)
hence *takeWhile* (λ*y*. *y* = *a*) *x* = *x*
by (*subst takeWhile-eq-take*, *simp*)
moreover
have ∀ *y* ∈ *set* (*takeWhile* (λ*y*. *y* = *a*) *x*). *y* = *a*
by (*auto dest*: *set-takeWhileD*)
ultimately
have ∀ *y* ∈ *set x*. *y* = *a* **by** *auto*
hence *x* = *replicate* (*length x*) *a* **by** *simp*
hence *int.lift f x* = *int.lift f* (*replicate* (*length x*) *a*) **by** *simp*
also have ... = *pow int-group* (*int.lift-gi f a*) (*length x*)
apply (*induct x*)
using *local.int.nat-pow-Suc local.int.nat-pow-0*
apply (*auto simp*: *int.lift-def* [*simplified*])
done
also have ... = (*int.lift-gi f a*) ∗ *int* (*length x*)
apply (*induct x*)
using *local.int.nat-pow-Suc local.int.nat-pow-0*
by (*auto simp*: *int-distrib*)
finally have ... = *0* **using** ‹*int.lift f x* = *0*› **by** *simp*

hence *nat* (*abs* (*group.lift-gi int-group f a* $*$ *int* (*length x*))) = *0* **by** *simp*
hence *nat* (*abs* (*group.lift-gi int-group f a*)) $*$ *length x* = *0* **by** *simp*
hence *nat* (*abs* (*group.lift-gi int-group f a*)) = *0*
using ‹$x \neq []$› **by** *auto*
moreover
have $inv_{int\text{-}group}$ *1* = -1
using *int.inv-equality* **by** *auto*
hence *abs* (*group.lift-gi int-group f a*) = *1*
using *int.is-group*
by(*auto simp add*: *group.lift-gi-def f-def*)
ultimately
show *False* **by** *simp*
qed
}
hence $\forall x \in carrier\ \mathcal{F}_{\{()\}}.\ int.lift\ f\ x = \mathbf{1}_{int\text{-}group} \longrightarrow x = \mathbf{1}_{\mathcal{F}_{\{()\}}}$
by (*auto simp add*:*free-group-def*)
moreover
{
have *carrier* $\mathcal{F}_{\{()\}} = \langle insert`\{()\}\rangle_{\mathcal{F}_{\{()\}}}$
by (*rule gens-span-free-group*[*THEN sym*])
moreover
have *carrier int-group* = $\langle\{1\}\rangle_{int\text{-}group}$
by (*rule int-group-gen-by-one*[*THEN sym*])
moreover
have *int.lift f* ‘ *insert* ‘ {()} = {*1*}
by (*auto simp add*: *int.lift-def* [*simplified*] *insert-def f-def int.lift-gi-def* [*simplified*])
moreover
have *int.lift f* ‘ $\langle insert`\{()\}\rangle_{\mathcal{F}_{\{()\}}} = \langle int.lift\ f\ `\ (insert\ `\{()\})\rangle_{int\text{-}group}$
by (*rule hom.hom-span*, *auto intro*:*insert-closed*)
ultimately
have *int.lift f* ‘ *carrier* $\mathcal{F}_{\{()\}}$ = *carrier int-group*
by *simp*
}
ultimately
have *int.lift f* $\in$ *iso* $\mathcal{F}_{\{()\}}$ *int-group*
using ‹*int.lift f* $\in$ *hom* $\mathcal{F}_{\{()\}}$ *int-group*›
using *hom.hom-mult int.is-group*
by (*auto intro*:*group-isoI simp add*: *free-group-is-group*)
thus *?thesis* **by** *auto*
qed

6.3 Free Groups over isomorphic sets of generators

Free Groups are isomorphic if their set of generators are isomorphic.

definition *lift-generator-function* :: $('a \Rightarrow 'b) \Rightarrow (bool \times 'a)\ list \Rightarrow (bool \times 'b)\ list$
where *lift-generator-function f* = *map* (*map-prod id f*)

```
theorem isomorphic-free-groups:
  assumes bij-betw f gens1 gens2
  shows lift-generator-function f ∈ iso ℱ_gens1 ℱ_gens2
unfolding lift-generator-function-def
proof(rule group-isoI)
  show ∀ x∈carrier ℱ_gens1.
       map (map-prod id f) x = 1_ℱ_gens2 ⟶ x = 1_ℱ_gens1
    by(auto simp add:free-group-def)
next
  from ‹bij-betw f gens1 gens2› have inj-on f gens1 by (auto simp:bij-betw-def)
  show map (map-prod id f) ‘ carrier ℱ_gens1 = carrier ℱ_gens2
  proof(rule Set.set-eqI,rule iffI)
   from ‹bij-betw f gens1 gens2› have f ‘ gens1 = gens2 by (auto simp:bij-betw-def)
    fix x :: (bool × 'b) list
    assume x ∈ image (map (map-prod id f)) (carrier ℱ_gens1)
    then obtain y :: (bool × 'a) list where x = map (map-prod id f) y
                and y ∈ carrier ℱ_gens1 by auto
    from ‹y ∈ carrier ℱ_gens1›
   have canceled y and y ∈ lists(UNIV×gens1) by (auto simp add:free-group-def)

    from ‹y ∈ lists (UNIV×gens1)›
      and ‹x = map (map-prod id f) y›
      and ‹image f gens1 = gens2›
    have x ∈ lists (UNIV×gens2)
      by (auto iff:lists-eq-set)
    moreover

    from ‹x = map (map-prod id f) y›
     and ‹y ∈ lists (UNIV×gens1)›
     and ‹canceled y›
     and ‹inj-on f gens1›
    have canceled x
         by (auto intro!:rename-gens-canceled subset-inj-on[OF ‹inj-on f gens1›]
iff:lists-eq-set)
    ultimately
    show x ∈ carrier ℱ_gens2 by (simp add:free-group-def)
  next
    fix x
    assume x ∈ carrier ℱ_gens2
    hence canceled x and x ∈ lists (UNIV×gens2)
      unfolding free-group-def by auto
    define y where y = map (map-prod id (the-inv-into gens1 f)) x
    have map (map-prod id f) y =
          map (map-prod id f) (map (map-prod id (the-inv-into gens1 f)) x)
      by (simp add:y-def)
    also have ... = map (map-prod id f ∘ map-prod id (the-inv-into gens1 f)) x
      by simp
    also have ... = map (map-prod id (f ∘ the-inv-into gens1 f)) x
```

```
    by auto
  also have ... = map id x
  proof(rule map-ext, rule impI)
    fix xa :: bool × 'b
    assume xa ∈ set x
    from ‹x ∈ lists (UNIV×gens2)›
    have set (map snd x) ⊆ gens2  by auto
    hence snd ` set x ⊆ gens2 by (simp add: set-map)
    with ‹xa ∈ set x› have snd xa ∈ gens2 by auto
    with ‹bij-betw f gens1 gens2› have snd xa ∈ f`gens1
      by (auto simp add: bij-betw-def)

    have map-prod id (f ∘ the-inv-into gens1 f) xa
         = map-prod id (f ∘ the-inv-into gens1 f) (fst xa, snd xa) by simp
    also have ... = (fst xa, f (the-inv-into gens1 f (snd xa)))
      by (auto simp del:prod.collapse)
    also
    from ‹snd xa ∈ image f gens1› and ‹inj-on f gens1›
    have ... = (fst xa, snd xa)
      by (auto elim:f-the-inv-into-f simp del:prod.collapse)
    also have ... = id xa by simp
    finally show map-prod id (f ∘ the-inv-into gens1 f) xa = id xa.
  qed
  also have ... = x unfolding id-def by auto
  finally have map (map-prod id f) y = x.
  moreover
  {
    from ‹bij-betw f gens1 gens2›
   have bij-betw (the-inv-into gens1 f) gens2 gens1 by (rule bij-betw-the-inv-into)
    hence inj-on (the-inv-into gens1 f) gens2 by (rule bij-betw-imp-inj-on)

    with ‹canceled x›
     and ‹x ∈ lists (UNIV×gens2)›
    have canceled y
      by (auto intro!:rename-gens-canceled[OF subset-inj-on] simp add:y-def)
    moreover
    {
      from ‹bij-betw (the-inv-into gens1 f) gens2 gens1›
       and ‹x∈lists(UNIV×gens2)›
      have y ∈ lists(UNIV×gens1)
        unfolding y-def and bij-betw-def
        by (auto iff:lists-eq-set dest!:subsetD)
    }
    ultimately
    have y ∈ carrier F_gens1 by (simp add:free-group-def)
  }
  ultimately
  show x ∈ map (map-prod id f) ` carrier F_gens1 by auto
qed
```

```
next
  from ‹bij-betw f gens1 gens2› have inj-on f gens1 by (auto simp:bij-betw-def)
  {
  fix x
  assume x ∈ carrier F_gens1
  fix y
  assume y ∈ carrier F_gens1

  from ‹x ∈ carrier F_gens1› and ‹y ∈ carrier F_gens1›
  have x ∈ lists(UNIV×gens1) and y ∈ lists(UNIV×gens1)
    by (auto simp add:occuring-gens-in-element)


  have map (map-prod id f) (x ⊗F_gens1 y)
       = map (map-prod id f) (normalize (x@y)) by (simp add:free-group-def)
  also
       from ‹x ∈ lists(UNIV×gens1)› and ‹y ∈ lists(UNIV×gens1)›
        and ‹inj-on f gens1›
       have ... = normalize (map (map-prod id f) (x@y))
         by −(rule rename-gens-normalize[THEN sym],
              auto intro!: subset-inj-on[OF ‹inj-on f gens1›] iff:lists-eq-set)
  also have ... = normalize (map (map-prod id f) x @ map (map-prod id f) y)
       by (auto)
  also have ... = map (map-prod id f) x ⊗F_gens2 map (map-prod id f) y
       by (simp add:free-group-def)
  finally have map (map-prod id f) (x ⊗F_gens1 y) =
               map (map-prod id f) x ⊗F_gens2 map (map-prod id f) y.
  }
  thus ∀x∈carrier F_gens1.
       ∀y∈carrier F_gens1.
          map (map-prod id f) (x ⊗F_gens1 y) =
          map (map-prod id f) x ⊗F_gens2 map (map-prod id f) y
   by auto
qed (auto intro: free-group-is-group)
```

6.4 Bases of isomorphic free groups

Isomorphic free groups have bases of same cardinality. The proof is very different for infinite bases and for finite bases.

The proof for the finite case uses the set of of homomorphisms from the free group to the group with two elements, as suggested by Christian Sievers. The definition of *hom* is not suitable for proofs about the cardinality of that set, as its definition does not require extensionality. This is amended by the following definition:

```
definition homr
  where homr G H = {h. h ∈ hom G H ∧ h ∈ extensional (carrier G)}
```

```
lemma (in group-hom) restrict-hom[intro!]:
  shows restrict h (carrier G) ∈ homr G H
  unfolding homr-def and hom-def
  by (auto)

lemma hom-F-C2-Powerset:
  ∃ f. bij-betw f (Pow X) (homr (F_X) C2)
proof
  interpret F: group F_X by (rule free-group-is-group)
  interpret C2: group C2 by (rule C2-is-group)
  let ?f = λS . restrict (C2.lift (λx. x ∈ S)) (carrier F_X)
  let ?f' = λh . X ∩ Collect(h ∘ insert)
  show bij-betw ?f (Pow X) (homr (F_X) C2)
  proof(induct rule: bij-betwI[of ?f - - ?f'])
  case 1 show ?case
    proof
      fix S assume S ∈ Pow X
      interpret h: group-hom F_X C2 C2.lift (λx. x ∈ S)
        by unfold-locales (auto intro: C2.lift-is-hom)
      show ?f S ∈ homr F_X C2
        by (rule h.restrict-hom)
     qed
  next
  case 2 show ?case by auto next
  case (3 S) show ?case
    proof (induct rule: Set.set-eqI)
      case (1 x) show ?case
      proof(cases x ∈ X)
      case True thus ?thesis using insert-closed[of x X]
         by (auto simp add:insert-def C2.lift-def C2.lift-gi-def)
      next case False thus ?thesis using 3 by auto
    qed
  qed
  next
  case (4 h)
    hence hom: h ∈ hom F_X C2
      and extn: h ∈ extensional (carrier F_X)
      unfolding homr-def by auto
   have ∀ x ∈ carrier F_X . h x = group.lift C2 (λz. z ∈ X & (h ∘ FreeGroups.insert)
z) x
     by (rule C2.lift-is-unique[OF C2-is-group - hom, of (λz. z ∈ X & (h ∘ Free-
Groups.insert) z)],
            auto)
    thus ?case
    by −(rule extensionalityI[OF restrict-extensional extn], auto)
  qed
qed
```

```
lemma group-iso-betw-hom:
  assumes group G1 and group G2
      and iso: i ∈ iso G1 G2
  shows  ∃ f . bij-betw f (homr G2 H) (homr G1 H)
proof−
  interpret G2: group G2 by (rule ‹group G2›)
  let ?i' = restrict (inv-into (carrier G1) i) (carrier G2)
  have inv-into (carrier G1) i ∈ iso G2 G1
    by (simp add: ‹group G1› group.iso-set-sym iso)
  hence iso': ?i' ∈ iso G2 G1
    by (auto simp add:Group.iso-def hom-def G2.m-closed)
  show ?thesis
   proof(rule, induct rule: bij-betwI[of (λh. compose (carrier G1) h i) - - (λh.
compose (carrier G2) h ?i')])
  case 1
    show ?case
    proof
      fix h assume h ∈ homr G2 H
      hence compose (carrier G1) h i ∈ hom G1 H
        using iso
      by (auto intro: group.hom-compose[OF ‹group G1›, of - G2] simp add:Group.iso-def
homr-def)
      thus compose (carrier G1) h i ∈ homr G1 H
        unfolding homr-def by simp
     qed
  next
  case 2
    show ?case
    proof
      fix h assume h ∈ homr G1 H
      hence compose (carrier G2) h ?i' ∈ hom G2 H
        using iso'
      by (auto intro: group.hom-compose[OF ‹group G2›, of - G1] simp add:Group.iso-def
homr-def)
      thus compose (carrier G2) h ?i' ∈ homr G2 H
        unfolding homr-def by simp
     qed
  next
  case (3 x)
    hence compose (carrier G2) (compose (carrier G1) x i) ?i'
          = compose (carrier G2) x (compose (carrier G2) i ?i')
      using iso iso'
      by (auto intro: compose-assoc[THEN sym]   simp add:Group.iso-def hom-def
homr-def)
    also have ... = compose (carrier G2) x (λy∈carrier G2. y)
      using iso
    by (subst compose-id-inv-into, auto simp add:Group.iso-def hom-def bij-betw-def)
    also have ... = x
      using 3
```

by (*auto intro*:*compose-Id simp add*:*homr-def*)
finally
show *?case* **.**
next
case (*4 y*)
hence *compose* (*carrier G1*) (*compose* (*carrier G2*) *y ?i'*) *i*
= *compose* (*carrier G1*) *y* (*compose* (*carrier G1*) *?i' i*)
using *iso iso'*
by (*auto intro*: *compose-assoc*[*THEN sym*] *simp add*:*Group.iso-def hom-def homr-def*)
also have ... = *compose* (*carrier G1*) *y* ($\lambda x \in$*carrier G1*. *x*)
using *iso*
by (*subst compose-inv-into-id*, *auto simp add*:*Group.iso-def hom-def bij-betw-def*)
also have ... = *y*
using *4*
by (*auto intro*:*compose-Id simp add*:*homr-def*)
finally
show *?case* **.**
qed
qed

lemma *isomorphic-free-groups-bases-finite*:
assumes *iso*: $i \in$ *iso* $\mathcal{F}_X$ $\mathcal{F}_Y$
and *finite*: *finite X*
shows $\exists f$. *bij-betw f X Y*
proof−
obtain *f*
where *bij-betw f* (*homr* $\mathcal{F}_Y$ *C2*) (*homr* $\mathcal{F}_X$ *C2*)
using *group-iso-betw-hom*[*OF free-group-is-group free-group-is-group iso*]
by *auto*
moreover
obtain *g'*
where *bij-betw g'* (*Pow X*) (*homr* ($\mathcal{F}_X$) *C2*)
using *hom-F-C2-Powerset* **by** *auto*
then obtain *g*
where *bij-betw g* (*homr* ($\mathcal{F}_X$) *C2*) (*Pow X*)
by (*auto intro*: *bij-betw-inv-into*)
moreover
obtain *h*
where *bij-betw h* (*Pow Y*) (*homr* ($\mathcal{F}_Y$) *C2*)
using *hom-F-C2-Powerset* **by** *auto*
ultimately
have *bij-betw* ($g \circ f \circ h$) (*Pow Y*) (*Pow X*)
by (*auto intro*: *bij-betw-trans*)
hence *eq-card*: *card* (*Pow Y*) = *card* (*Pow X*)
by (*rule bij-betw-same-card*)
with *finite*
have *finite* (*Pow Y*)
by −(*rule card-ge-0-finite*, *auto simp add*:*card-Pow*)

hence *finite′*: *finite Y* **by** *simp*

with *eq-card finite*
have *card X = card Y*
by (*auto simp add*:*card-Pow*)
with *finite finite′*
show *?thesis*
by (*rule finite-same-card-bij*)
qed

The proof for the infinite case is trivial once the fact that the free group over an infinite set has the same cardinality is established.

lemma *free-group-card-infinite*:
assumes ¬ *finite X*
shows $|X| =o\ |carrier\ \mathcal{F}_X|$
proof−
have *inj-on insert X*
by (*rule inj-onI*) (*auto simp add*: *insert-def*)
moreover have *insert* ‘ $X \subseteq$ *carrier* $\mathcal{F}_X$
by (*auto intro*: *insert-closed*)
ultimately have $\exists f.$ *inj-on f X* $\wedge$ *f* ‘ $X \subseteq$ *carrier* $\mathcal{F}_X$
by *auto*
then have $|X| \leq o\ |carrier\ \mathcal{F}_X|$
by (*simp add*: *card-of-ordLeq*)
moreover
have $|carrier\ \mathcal{F}_X| \leq o\ |lists\ ((UNIV::bool\ set)\times X)|$
by (*auto intro*!:*card-of-mono1 simp add*:*free-group-def*)
moreover
have $|lists\ ((UNIV::bool\ set)\times X)| =o\ |(UNIV::bool\ set)\times X|$
using ‹¬ *finite X*›
by (*auto intro*:*card-of-lists-infinite dest*!:*finite-cartesian-productD2*)
moreover
have $|(UNIV::bool\ set)\times X| =o\ |X|$
using ‹¬ *finite X*›
by (*auto intro*: *card-of-Times-infinite*[*OF - - ordLess-imp-ordLeq*[*OF finite-ordLess-infinite2*], *THEN conjunct2*])
ultimately
show $|X| =o\ |carrier\ \mathcal{F}_X|$
by (*subst ordIso-iff-ordLeq*, *auto intro*: *ord-trans*)
qed

theorem *isomorphic-free-groups-bases*:
assumes *iso*: $i \in iso\ \mathcal{F}_X\ \mathcal{F}_Y$
shows $\exists f.$ *bij-betw f X Y*
proof(*cases finite X*)
case *True*
thus *?thesis* **using** *iso* **by** −(*rule isomorphic-free-groups-bases-finite*)
next
case *False* **show** *?thesis*

proof(*cases finite Y*)
case *True*
from *iso* **obtain** i' **where** $i' \in iso\ \mathcal{F}_Y\ \mathcal{F}_X$
using *free-group-is-group group.iso-set-sym* **by** *blast*
with ‹*finite Y*›
have $\exists f.$ *bij-betw f Y X* **by** −(*rule isomorphic-free-groups-bases-finite*)
thus $\exists f.$ *bij-betw f X Y* **by** (*auto intro*: *bij-betw-the-inv-into*) **next**
case *False*
from ‹¬ *finite X*› **have** $|X| =o\ |carrier\ \mathcal{F}_X|$
by (*rule free-group-card-infinite*)
moreover
from ‹¬ *finite Y*› **have** $|Y| =o\ |carrier\ \mathcal{F}_Y|$
by (*rule free-group-card-infinite*)
moreover
from *iso* **have** $|carrier\ \mathcal{F}_X| =o\ |carrier\ \mathcal{F}_Y|$
by (*auto simp add*:*Group.iso-def iff*:*card-of-ordIso*[*THEN sym*])
ultimately
have $|X| =o\ |Y|$ **by** (*auto intro*: *ordIso-equivalence*)
thus *?thesis* **by** (*subst card-of-ordIso*)
qed
qed

end

7 The Ping Pong lemma

theory *PingPongLemma*
imports
HOL−Algebra.Bij
FreeGroups
begin

The Ping Pong Lemma is a way to recognice a Free Group by its action on a set (often a topological space or a graph). The name stems from the way that elements of the set are passed forth and back between the subsets given there.

We start with two auxiliary lemmas, one about the identity of the group of bijections, and one about sets of cardinality larger than one.

lemma *Bij-one*[*simp*]:
assumes $x \in X$
shows $\mathbf{1}_{BijGroup\ X}\ x = x$
using *assms* **by** (*auto simp add*: *BijGroup-def*)

lemma *other-member*:
assumes $I \neq \{\}$ **and** $i \in I$ **and** *card* $I \neq 1$
obtains j **where** $j{\in}I$ **and** $j{\neq}i$
proof(*cases finite I*)
case *True*

hence $I - \{i\} \neq \{\}$ **using** ‹*card I* $\neq$ *1*› **and** ‹$i{\in}I$› **by** (*metis Suc-eq-plus1-left card-Diff-subset-Int card-Suc-Diff1 diff-add-inverse2 diff-self-eq-0 empty-Diff finite.emptyI inf-bot-left minus-nat.diff-0*)
thus *?thesis* **using** *that* **by** *auto*
next
case *False*
hence $I - \{i\} \neq \{\}$ **by** (*metis Diff-empty finite.emptyI finite-Diff-insert*)
thus *?thesis* **using** *that* **by** *auto*
qed

And now we can attempt the lemma. The gencount condition is a weaker variant of "x has to lie outside all subsets" that is only required if the set of generators is one. Otherwise, we will be able to find a suitable x to start with in the proof.

lemma *ping-pong-lemma*:
assumes *group G*
and $act \in hom\ G\ (BijGroup\ X)$
and $g \in (I \rightarrow carrier\ G)$
and $\langle g\ `\ I\rangle_G = carrier\ G$
and *sub1*: $\forall i{\in}I.\ Xout\ i \subseteq X$
and *sub2*: $\forall i{\in}I.\ Xin\ i \subseteq X$
and *disj1*: $\forall i{\in}I.\ \forall j{\in}I.\ i \neq j \longrightarrow Xout\ i \cap Xout\ j = \{\}$
and *disj2*: $\forall i{\in}I.\ \forall j{\in}I.\ i \neq j \longrightarrow Xin\ i \cap Xin\ j = \{\}$
and *disj3*: $\forall i{\in}I.\ \forall j{\in}I.\ Xin\ i \cap Xout\ j = \{\}$
and $x \in X$
and *gencount*: $\forall\ i\ .\ I = \{i\} \longrightarrow (x \notin Xout\ i \wedge x \notin Xin\ i)$
and *ping*: $\forall i{\in}I.\ act\ (g\ i)\ `\ (X - Xout\ i) \subseteq Xin\ i$
and *pong*: $\forall i{\in}I.\ act\ (inv_G\ (g\ i))\ `\ (X - Xin\ i) \subseteq Xout\ i$
shows $group.lift\ G\ g \in iso\ (\mathcal{F}_I)\ G$
proof−
interpret *F*: *group* $\mathcal{F}_I$
using *assms* **by** (*auto simp add*: *free-group-is-group*)
interpret *G*: *group G* **by** *fact*
interpret *B*: *group BijGroup X* **using** *group-BijGroup* **by** *auto*
interpret *act*: *group-hom G BijGroup X act* **by** (*unfold-locales*) *fact*
interpret *h*: *group-hom* $\mathcal{F}_I$ *G G.lift g*
using *F.is-group G.is-group G.lift-is-hom assms*
by (*auto intro*!: *group-hom.intro group-hom-axioms.intro*)

show *?thesis*
proof(*rule h.group-hom-isoI*)

Injectivity is the hard part of the proof.

show $\forall x{\in}carrier\ \mathcal{F}_I.\ G.lift\ g\ x = \mathbf{1}_G \longrightarrow x = \mathbf{1}_{\mathcal{F}_I}$
proof(*rule*+)

We lift the Xout and Xin sets to generators and their inveres, and create variants of the disj-conditions:

define *Xout'* **where** $Xout' = (\lambda(b,i{::}'d).\ if\ b\ then\ Xin\ i\ else\ Xout\ i)$

```
define Xin' where Xin' = (λ(b,i::'d). if b then Xout i else Xin i)

have disj1': ∀i∈(UNIV × I). ∀j∈(UNIV × I). i ≠ j ⟶ Xout' i ∩ Xout' j = {}
  using disj1[rule-format] disj2[rule-format] disj3[rule-format]
  by (auto simp add:Xout'-def Xin'-def split:if-splits, blast+)
have disj2': ∀i∈(UNIV × I). ∀j∈(UNIV × I). i ≠ j ⟶ Xin' i ∩ Xin' j = {}
  using disj1[rule-format] disj2[rule-format] disj3[rule-format]
  by (auto simp add:Xout'-def Xin'-def split:if-splits, blast+)
have disj3': ∀i∈(UNIV × I). ∀j∈(UNIV × I). ¬ canceling i j ⟶ Xin' i ∩ Xout' j = {}
  using disj1[rule-format] disj2[rule-format] disj3[rule-format]
  by (auto simp add:canceling-def Xout'-def Xin'-def split:if-splits, blast)
```

We need to pick a suitable element of the set to play ping pong with. In particular, it needs to be outside of the Xout-set of the last generator in the list, and outside the in-set of the first element. This part of the proof is surprisingly tedious, because there are several cases, some similar but not the same.

```
fix w
assume w: w ∈ carrier ℱ_I

obtain x where x ∈ X
  and x1: w = [] ∨ x ∉ Xout' (last w)
  and x2: w = [] ∨ x ∉ Xin' (hd w)
proof−
  { assume I = {}
    hence w = [] using w by (auto simp add:free-group-def)
    hence ?thesis using that ‹x∈X› by auto
  }
  moreover
  { assume card I = 1
    then obtain i where I={i} by (auto dest: card-eq-SucD)
    assume w≠[]
    hence snd (hd w) = i and snd (last w) = i
      using w ‹I={i}›
      apply (cases w, auto simp add:free-group-def)
      apply (cases w rule:rev-exhaust, auto simp add:free-group-def)
      done
    hence ?thesis using gencount[rule-format, OF ‹I={i}›] that[OF ‹x∈X›] ‹w≠[]›
    by (cases last w, cases hd w, auto simp add:Xout'-def Xin'-def split:if-splits)
  }
  moreover
  { assume I ≠ {} and card I ≠ 1 and w ≠ []

    from ‹w ≠ []› and w
    obtain b i where hd: hd w = (b,i) and i∈I
      by (cases w, auto simp add:free-group-def)
    from ‹w ≠ []› and w
```

```
obtain b′ i′ where last: last w = (b′,i′) and i′∈I
  by (cases w rule: rev-exhaust, auto simp add:free-group-def)
```

What follows are two very similar cases, but the correct choice of variables depends on where we find x.

```
{
obtain b′′ i′′ where
  (b′′,i′′) ≠ (b,i) and
  (b′′,i′′) ≠ (b′,i′) and
  ¬ canceling (b′′, i′′) (b′,i′) and
  i′′∈I
proof(cases i=i′)
  case True
  obtain j where j∈I and j≠i using ‹card I ≠ 1› and ‹i∈I›
    by −(rule other-member, auto)
 with True show ?thesis using that by (auto simp add:canceling-def)
next
  case False thus ?thesis using that ‹i∈I› ‹i′ ∈ I›
  by (simp add:canceling-def, metis)
qed
let ?g = (b′′,i′′)

assume x ∈ Xout′ (last w)
hence x ∉ Xout′ ?g
  using disj1′[rule-format, OF - - ‹?g ≠ (b′,i′)›]
      ‹i ∈ I› ‹i′∈I› ‹i′′∈I› hd last
  by auto
hence act (G.lift-gi g ?g) x ∈ Xin′ ?g (is ?x ∈ -) using ‹i′′ ∈ I› ‹x ∈
X›
  ping[rule-format, OF ‹i′′ ∈ I›, THEN subsetD]
  pong[rule-format, OF ‹i′′ ∈ I›, THEN subsetD]
  by (auto simp add:G.lift-def G.lift-gi-def Xout′-def Xin′-def)
hence ?x ∉ Xout′ (last w) ∧ ?x ∉ Xin′ (hd w)
  using
    disj3′[rule-format, OF - - ‹¬ canceling (b′′, i′′) (b′,i′)›]
    disj2′[rule-format, OF - -  ‹?g ≠ (b,i)›]
    ‹i ∈ I› ‹i′∈I› ‹i′′∈I› hd last
  by (auto simp add: canceling-def)
moreover
note ‹i′′ ∈ I›
hence g i′′ ∈ carrier G using ‹g ∈ (I → carrier G)› by auto
hence G.lift-gi g ?g ∈ carrier G
  by (auto simp add:G.lift-gi-def inv1-def)
hence act (G.lift-gi g ?g) ∈ carrier (BijGroup X)
  using ‹act ∈ hom G (BijGroup X)› by auto
hence ?x ∈ X using ‹x∈X›
  by (auto simp add:BijGroup-def Bij-def bij-betw-def)
ultimately have ?thesis using that[of ?x] by auto
}
moreover
```

```
  {
  obtain b'' i'' where
    ¬ canceling (b'',i'') (b,i) and
    ¬ canceling (b'',i'') (b',i') and
    (b,i) ≠ (b'',i'') and
    i''∈I
  proof(cases i=i')
    case True
    obtain j where j∈I and j≠i using  ‹card I ≠ 1› and ‹i∈I›
      by -(rule other-member, auto)
   with True show ?thesis using that by (auto simp add:canceling-def)
  next
    case False thus ?thesis using that ‹i∈I› ‹i' ∈ I›
    by (simp add:canceling-def, metis)
  qed
  let ?g = (b'',i'')
  note cancel-sym-neg[OF ‹¬ canceling (b'',i'') (b,i)›]
  note cancel-sym-neg[OF ‹¬ canceling (b'',i'') (b',i')›]

  assume x ∈ Xin' (hd w)
  hence x ∉ Xout' ?g
    using disj3'[rule-format, OF - - ‹¬ canceling (b,i) ?g›]
       ‹i ∈ I› ‹i'∈I› ‹i''∈I› hd last
    by auto
  hence act (G.lift-gi g ?g) x ∈ Xin' ?g (is ?x ∈ -) using ‹i'' ∈ I› ‹x ∈
X›
    ping[rule-format, OF ‹i'' ∈ I›, THEN subsetD]
    pong[rule-format, OF ‹i'' ∈ I›, THEN subsetD]
    by (auto simp add:G.lift-def G.lift-gi-def Xout'-def Xin'-def)
  hence ?x ∉ Xout' (last w) ∧ ?x ∉ Xin' (hd w)
    using
      disj3'[rule-format, OF - - ‹¬ canceling ?g (b',i')›]
      disj2'[rule-format, OF - -  ‹(b,i) ≠ ?g›]
      ‹i ∈ I› ‹i'∈I› ‹i''∈I› hd last
    by (auto simp add: canceling-def)
  moreover
  note ‹i'' ∈ I›
  hence g i'' ∈ carrier G using ‹g ∈ (I → carrier G)› by auto
  hence G.lift-gi g ?g ∈ carrier G
    by (auto simp add:G.lift-gi-def)
  hence act (G.lift-gi g ?g) ∈ carrier (BijGroup X)
    using ‹act ∈ hom G (BijGroup X)› by auto
  hence ?x ∈ X using ‹x∈X›
    by (auto simp add:BijGroup-def Bij-def bij-betw-def)
  ultimately have ?thesis using that[of ?x] by auto
  }
  moreover note calculation
}
ultimately show ?thesis using ‹x∈ X› that by auto
```

```
  qed
```

The proof works by induction over the length of the word. Each inductive step is one ping as in ping pong. At the end, we land in one of the subsets of X, so the word cannot be the identity.

```
  from x1 and w
  have w = [] ∨ act (G.lift g w) x ∈ Xin' (hd w)
  proof(induct w)
    case Nil show ?case by simp
  next case (Cons w ws)
    note C = Cons
```

The following lemmas establish all "obvious" element relations that will be required during the proof.

```
    note calculation = Cons(3)
    moreover have x∈X by fact
  moreover have snd w ∈ I using calculation by (auto simp add:free-group-def)

    moreover have g ∈ (I → carrier G) by fact
    moreover have g (snd w) ∈ carrier G using calculation by auto
    moreover have ws ∈ carrier ℱ_I
      using calculation by (auto intro:cons-canceled simp add:free-group-def)
    moreover have G.lift g ws ∈ carrier G and G.lift g [w] ∈ carrier G
       using calculation by (auto simp add: free-group-def)
    moreover have act (G.lift g ws) ∈ carrier (BijGroup X)
            and act (G.lift g [w]) ∈ carrier (BijGroup X)
            and act (G.lift g (w#ws)) ∈ carrier (BijGroup X)
            and act (g (snd w)) ∈ carrier (BijGroup X)
       using calculation by auto
    moreover have act (g (snd w)) ∈ Bij X
       using calculation by (auto simp add:BijGroup-def)
    moreover have act (G.lift g ws) x ∈ X (is ?x2 ∈ X)
      using calculation by (auto simp add:BijGroup-def Bij-def bij-betw-def)
    moreover have act (G.lift g [w]) ?x2 ∈ X
      using calculation by (auto simp add:BijGroup-def Bij-def bij-betw-def)
    moreover have act (G.lift g (w#ws)) x ∈ X
      using calculation by (auto simp add:BijGroup-def Bij-def bij-betw-def)
    moreover note mems = calculation

    have act (G.lift g ws) x ∉ Xout' w
    proof(cases ws)
      case Nil
        moreover have x ∉ Xout' w using Cons(2) Nil
          unfolding Xout'-def using mems
          by (auto split:if-splits)
        ultimately show act (G.lift g ws) x ∉ Xout' w
          using mems by auto
    next case (Cons ww wws)
      hence act (G.lift g ws) x ∈ Xin' (hd ws)
        using C mems by simp
```

moreover have *Xin′* (*hd ws*) $\cap$ *Xout′ w* = {}
proof−
have $\neg$ *canceling* (*hd ws*) *w*
proof
assume *canceling* (*hd ws*) *w*
hence *cancels-to-1* (*w#ws*) *wws* **using** *Cons*
by(*auto simp add*:*cancel-sym cancels-to-1-def cancels-to-1-at-def cancel-at-def*)
thus *False* **using** ‹*w#ws* $\in$ *carrier* $\mathcal{F}_I$›
by(*auto simp add*:*free-group-def canceled-def*)
qed

have *w* $\in$ *UNIV* $\times$ *I hd ws* $\in$ *UNIV* $\times$ *I*
using ‹*snd w* $\in$ *I*› *mems Cons*
by (*cases w*, *auto*, *cases hd ws*, *auto simp add*:*free-group-def*)
thus *?thesis*
by− (*rule disj3′*[*rule-format*, *OF* - - ‹$\neg$ *canceling* (*hd ws*) *w*›], *auto*)
qed
ultimately show *act* (*G.lift g ws*) *x* $\notin$ *Xout′ w* **using** *Cons* **by** *auto*
qed
show *?case*
proof−
have *act* (*G.lift g* (*w # ws*)) *x* = *act* (*G.lift g* ([*w*] @ *ws*)) *x* **by** *simp*
also have ... = *act* (*G.lift g* [*w*] $\otimes_G$ *G.lift g ws*) *x*
using *mems* **by** (*subst G.lift-append*, *auto simp add*:*free-group-def*)
also have ... = (*act* (*G.lift g* [*w*]) $\otimes_{BijGroup\ X}$ *act* (*G.lift g ws*)) *x*
using *mems* **by** (*auto simp add*:*act.hom-mult free-group-def intro*!:*G.lift-closed*)
also have ... = *act* (*G.lift g* [*w*]) (*act* (*G.lift g ws*) *x*)
using *mems* **by** (*auto simp add*:*BijGroup-def compose-def*)
also have ... $\notin$ *act* (*G.lift g* [*w*]) ‘ *Xout′ w*
apply(*rule ccontr*)
apply *simp*
apply (*erule imageE*)
apply (*subst* (*asm*) *inj-on-eq-iff*[*of act* (*G.lift g* [*w*]) *X*])
using *mems* ‹*act* (*G.lift g ws*) *x* $\notin$ *Xout′ w*› ‹$\forall$ *i*$\in$*I*. *Xout i* $\subseteq$ *X*› ‹$\forall$ *i*$\in$*I*. *Xin i* $\subseteq$ *X*›
apply (*auto simp add*:*BijGroup-def Bij-def bij-betw-def free-group-def Xout′-def split*:*if-splits*)
apply *blast*+
done
finally
have *act* (*G.lift g* (*w # ws*)) *x* $\in$ *Xin′ w*
proof−
assume *act* (*G.lift g* (*w # ws*)) *x* $\notin$ *act* (*G.lift g* [*w*]) ‘ *Xout′ w*
hence *act* (*G.lift g* (*w # ws*)) *x* $\in$ (*X* − *act* (*G.lift g* [*w*]) ‘ *Xout′ w*)
using *mems* **by** *auto*
also have ... $\subseteq$ *act* (*G.lift g* [*w*]) ‘ *X* − *act* (*G.lift g* [*w*]) ‘ *Xout′ w*
using ‹*act* (*G.lift g* [*w*]) $\in$ *carrier* (*BijGroup X*)›

```
                by (auto simp add:BijGroup-def Bij-def bij-betw-def)
            also have ... ⊆ act (G.lift g [w]) ‘ (X − Xout′ w)
                by (rule image-diff-subset)
            also have ... ⊆ Xin′ w
            proof(cases fst w)
              assume ¬ fst w
              thus ?thesis
                using mems
                  by (auto intro!: ping[rule-format, THEN subsetD] simp add:
Xout′-def Xin′-def G.lift-def G.lift-gi-def free-group-def)
            next assume fst w
              thus ?thesis
                using mems
                  by (auto intro!: pong[rule-format, THEN subsetD] simp add:
restrict-def inv-BijGroup Xout′-def Xin′-def G.lift-def G.lift-gi-def free-group-def)
            qed
            finally show ?thesis .
          qed
          thus ?thesis by simp
        qed
      qed
        moreover assume G.lift g w = 1_G
      ultimately show w = 1_F_I
        using ‹x∈X› Cons(1) x2 ‹w ∈ carrier F_I›
      by (cases w, auto simp add:free-group-def Xin′-def split:if-splits)
    qed
  next
```

Surjectivity is relatively simple, and often not even mentioned in human proofs.

```
  have G.lift g ‘ carrier F_I =
        G.lift g ‘ ⟨ι ‘ I⟩_F_I
    by (metis gens-span-free-group)
  also have ... = ⟨G.lift g ‘ (ι ‘ I) ⟩_G
    by (auto intro!:h.hom-span simp add: insert-closed)
  also have ... = ⟨g ‘ I ⟩_G
    proof−
      have ∀ i ∈ I. G.lift g (ι i) = g i
        using ‹g ∈ (I → carrier G)›
        by (auto simp add:insert-def G.lift-def G.lift-gi-def intro:G.r-one)
      then have G.lift g ‘ (ι ‘ I) = g ‘ I
          by (auto intro!: image-cong simp add: image-comp [symmetric, THEN
sym])
      thus ?thesis by simp
    qed
   also have ... = carrier G using assms by simp
   finally show G.lift g ‘ carrier F_I = carrier G.
 qed
qed
```

end