The Banach–Tarski Paradox in Isabelle/HOL

Arthur F. Ramos David Barros Hulak
Ruy J. G. B. de Queiroz

Abstract

We formalise in Isabelle/HOL the classical Banach–Tarski paradox for the closed unit ball in three-dimensional Euclidean space. The development proves the free-group paradox, the free subgroup of $SO(3)$, the Hausdorff paradox away from a countable bad set, the absorption arguments for the sphere and the origin, and the final finite-partition theorem for the ball.

The proof follows the standard route: the paradoxical decomposition of the free group on two generators F_2, the existence of a free subgroup $F_2 \leq SO(3)$ via the AFP `Free-Groups` entry, the Hausdorff paradox for $S^2 \setminus D$ where D is a countable set of fixed points, and absorption arguments extending the partition to S^2, then radially to $B^3 \setminus \{0\}$, and finally to B^3 itself. The argument crucially relies on the axiom of choice.

Background

The Banach–Tarski paradox was proved by Banach and Tarski [1], building on Hausdorff's earlier paradoxical decomposition of the sphere [4]. The route formalised here follows the standard free-group presentation from Wagon's expositions [7, 6] and can be compared with de Rauglaudre's Coq formalisation [3]. The Isabelle development uses Breitner's AFP entry on free groups [2].

Formalisation Profile

The generated Isabelle session is `Banach_Tarski`. It is based on the HOL Analysis library, imports the AFP entry `Free-Groups`, and uses the `Primes` theory for the arithmetic freeness argument. The development introduces no local axiomatization or oracle. Its non-constructive steps use the standard Hilbert-choice operators of HOL, made explicit in the orbit-representative construction and inverse-map uses.

The checked target is Isabelle 2025-2 with AFP 2026-04-09.

Contents

theory *BT-Prelim*
imports
HOL−Analysis.Analysis
Free−Groups.FreeGroups
begin

1 The unit sphere and unit ball in $\mathbb{R}^3$

We work throughout with vectors in (*real*, *3*) *vec* (the type (*real*, *3*) *vec* from *HOL−Analysis.Cartesian-Euclidean-Space*). These abbreviations name the principal geometric objects of interest.

definition *sphere2* :: (*real^3*) *set* **where**
sphere2 = {*x*. *norm x* = *1*}

definition *ball3* :: (*real^3*) *set* **where**
ball3 = {*x*. *norm x* ≤ *1*}

lemma *sphere2-subset-ball3*: *sphere2* ⊆ *ball3*
by (*auto simp*: *sphere2-def ball3-def*)

lemma *zero-in-ball3*: *0* ∈ *ball3*
by (*simp add*: *ball3-def*)

lemma *zero-notin-sphere2*: (*0*::*real^3*) ∉ *sphere2*
by (*simp add*: *sphere2-def*)

2 Disjoint finite covers

A *partition* of a set into a finite list of pairwise disjoint pieces is the basic combinatorial object underlying "equidecomposability".

definition *pairwise-disjoint* :: *'a set list* ⇒ *bool* **where**
pairwise-disjoint Xs ⟷
(∀ *i* < *length Xs*. ∀ *j* < *length Xs*. *i* ≠ *j* ⟶ *Xs* ! *i* ∩ *Xs* ! *j* = {})

lemma *pairwise-disjoint-Nil* [*simp*]: *pairwise-disjoint* []
by (*simp add*: *pairwise-disjoint-def*)

lemma *pairwise-disjoint-singleton* [*simp*]: *pairwise-disjoint* [*X*]
by (*simp add*: *pairwise-disjoint-def*)

lemma *pairwise-disjoint-nthD*:
assumes *pairwise-disjoint Xs*
and $i < length\ Xs$ **and** $j < length\ Xs$ **and** $i \neq j$
shows $Xs\ !\ i \cap Xs\ !\ j = \{\}$
using *assms bot-set-def pairwise-disjoint-def* **by** *auto*

lemma *indexed-union-set*:
fixes $P :: 'a\ set\ list$
shows $(\bigcup i{<}length\ P.\ P\ !\ i) = \bigcup (set\ P)$
by (*auto simp add*: *set-conv-nth image-def*)

lemma *indexed-union-append*:
fixes $P\ Q :: 'a\ set\ list$
shows $(\bigcup i{<}length\ (P\ @\ Q).\ (P\ @\ Q)\ !\ i) =$
$(\bigcup i{<}length\ P.\ P\ !\ i) \cup (\bigcup i{<}length\ Q.\ Q\ !\ i)$
by (*metis Union-Un-distrib indexed-union-set set-append*)

lemma *indexed-image-union-zip*:
fixes $P :: 'a\ set\ list$ **and** $G :: ('a \Rightarrow 'b)\ list$
assumes $length\ P = length\ G$
shows $(\bigcup i{<}length\ P.\ G\ !\ i\ `\ (P\ !\ i)) =$
$(\bigcup (g,\ A){\in}set\ (zip\ G\ P).\ g\ `\ A)$
using *assms* **by** (*auto simp add*: *set-zip*)

lemma *indexed-image-union-append*:
fixes $P\ Q :: 'a\ set\ list$ **and** $G\ H :: ('a \Rightarrow 'b)\ list$
assumes *lenP*: $length\ P = length\ G$ **and** *lenQ*: $length\ Q = length\ H$
shows $(\bigcup i{<}length\ (P\ @\ Q).\ (G\ @\ H)\ !\ i\ `\ ((P\ @\ Q)\ !\ i)) =$
$(\bigcup i{<}length\ P.\ G\ !\ i\ `\ (P\ !\ i)) \cup$
$(\bigcup i{<}length\ Q.\ H\ !\ i\ `\ (Q\ !\ i))$
proof −
have *len-append*: $length\ (P\ @\ Q) = length\ (G\ @\ H)$
using *lenP lenQ* **by** *simp*
have $(\bigcup i{<}length\ (P\ @\ Q).\ (G\ @\ H)\ !\ i\ `\ ((P\ @\ Q)\ !\ i)) =$
$(\bigcup (g,\ A){\in}set\ (zip\ (G\ @\ H)\ (P\ @\ Q)).\ g\ `\ A)$
using *indexed-image-union-zip*[*OF len-append*] **.**
also have ... =
$(\bigcup (g,\ A){\in}set\ (zip\ G\ P).\ g\ `\ A) \cup$
$(\bigcup (g,\ A){\in}set\ (zip\ H\ Q).\ g\ `\ A)$
using *lenP* **by** *simp*
also have ... =
$(\bigcup i{<}length\ P.\ G\ !\ i\ `\ (P\ !\ i)) \cup$
$(\bigcup i{<}length\ Q.\ H\ !\ i\ `\ (Q\ !\ i))$
using *indexed-image-union-zip*[*OF lenP*] *indexed-image-union-zip*[*OF lenQ*] **by** *simp*
finally show *?thesis* **.**
qed

lemma *indexed-union-map-fst*:
fixes *xs* :: (*'a set* × *'b*) *list*
shows (⋃*i*<*length* (*map fst xs*). *map fst xs* ! *i*) =
(⋃*x*∈*set xs*. *fst x*)
by (*auto simp add*: *set-conv-nth*)

lemma *indexed-image-union-pairs*:
fixes *xs* :: (*'a set* × (*'a* ⇒ *'b*)) *list*
shows (⋃*i*<*length* (*map fst xs*). *map snd xs* ! *i* ‘ (*map fst xs* ! *i*)) =
(⋃*x*∈*set xs*. *snd x* ‘ *fst x*)
by (*auto simp add*: *set-conv-nth*)

lemma *pairwise-disjoint-map-fstI*:
fixes *xs* :: (*'a set* × *'b*) *list*
assumes *distinct*: *distinct xs*
and *disj*: ⋀*x y*. ⟦*x* ∈ *set xs*; *y* ∈ *set xs*; *x* ≠ *y*⟧
⟹ *fst x* ∩ *fst y* = {}
shows *pairwise-disjoint* (*map fst xs*)
unfolding *pairwise-disjoint-def*
proof (*intro allI impI*)
fix *i j*
assume *i*: *i* < *length* (*map fst xs*) **and** *j*: *j* < *length* (*map fst xs*) **and** *ij*: *i* ≠ *j*
have *xi*: *xs* ! *i* ∈ *set xs* **and** *xj*: *xs* ! *j* ∈ *set xs*
using *i j* **by** *auto*
have *neq*: *xs* ! *i* ≠ *xs* ! *j*
using *distinct i j ij* **by** (*simp add*: *nth-eq-iff-index-eq*)
show *map fst xs* ! *i* ∩ *map fst xs* ! *j* = {}
using *disj*[*OF xi xj neq*] *i j* **by** *simp*
qed

lemma *pairwise-disjoint-appendI*:
assumes *xs*: *pairwise-disjoint xs*
and *ys*: *pairwise-disjoint ys*
and *cross*: ⋀*x y*. ⟦*x* ∈ *set xs*; *y* ∈ *set ys*⟧ ⟹ *x* ∩ *y* = {}
shows *pairwise-disjoint* (*xs* @ *ys*)
unfolding *pairwise-disjoint-def*
proof (*intro allI impI*)
fix *i j*
assume *i*: *i* < *length* (*xs* @ *ys*) **and** *j*: *j* < *length* (*xs* @ *ys*) **and** *ij*: *i* ≠ *j*
show (*xs* @ *ys*) ! *i* ∩ (*xs* @ *ys*) ! *j* = {}
proof (*cases i* < *length xs*)
case *True*
show *?thesis*
proof (*cases j* < *length xs*)
case *True*
have *xs* ! *i* ∩ *xs* ! *j* = {}
by (*rule pairwise-disjoint-nthD*[*OF xs* ‹*i* < *length xs*› *True ij*])
with ‹*i* < *length xs*› *True* **show** *?thesis*
by (*simp add*: *nth-append*)

```
    next
      case False
      hence (xs @ ys) ! j ∈ set ys
        using j by (simp add: nth-append)
      moreover have (xs @ ys) ! i ∈ set xs
        by (simp add: True nth-append-left)
      ultimately show ?thesis
        using cross by simp
    qed
  next
    case False
    show ?thesis
    proof (cases j < length xs)
      case True
      hence (xs @ ys) ! j ∈ set xs
      proof −
        have (xs @ ys) ! j = xs ! j
          using True by (simp add: nth-append)
        moreover have xs ! j ∈ set xs
          by (rule nth-mem[OF True])
        ultimately show ?thesis
          by simp
      qed
      moreover have (xs @ ys) ! i ∈ set ys
        using False i by (simp add: nth-append)
      ultimately have (xs @ ys) ! j ∩ (xs @ ys) ! i = {}
        using cross by blast
      thus ?thesis
        by (simp add: Int-commute)
    next
      case False
      have ys ! (i − length xs) ∩ ys ! (j − length xs) = {}
       by (rule pairwise-disjoint-nthD[OF ys]) (use False ‹¬ i < length xs› i j ij in
auto)
      with False ‹¬ i < length xs› i j show ?thesis
        by (simp add: nth-append)
    qed
  qed
qed

lemma pairwise-disjoint-appendD1:
  assumes pairwise-disjoint (xs @ ys)
  shows pairwise-disjoint xs
  unfolding pairwise-disjoint-def
proof (intro allI impI)
  fix i j
  assume i: i < length xs and j: j < length xs and ij: i ≠ j
  have (xs @ ys) ! i ∩ (xs @ ys) ! j = {}
    by (rule pairwise-disjoint-nthD[OF assms]) (use i j ij in simp-all)
```

```
  thus xs ! i ∩ xs ! j = {}
    using i j by (simp add: nth-append)
qed

lemma pairwise-disjoint-appendD2:
  assumes pairwise-disjoint (xs @ ys)
  shows pairwise-disjoint ys
  unfolding pairwise-disjoint-def
proof (intro allI impI)
  fix i j
  assume i: i < length ys and j: j < length ys and ij: i ≠ j
  have length xs + i < length (xs @ ys) and length xs + j < length (xs @ ys)
    using i j by simp-all
  moreover have length xs + i ≠ length xs + j
    using ij by simp
  ultimately have (xs @ ys) ! (length xs + i) ∩ (xs @ ys) ! (length xs + j) = {}
    by (rule pairwise-disjoint-nthD[OF assms])
  thus ys ! i ∩ ys ! j = {}
    using i j by simp
qed

lemma pairwise-disjoint-append-crossD:
  assumes pairwise-disjoint (xs @ ys)
    and i < length xs and j < length ys
  shows xs ! i ∩ ys ! j = {}
proof −
  have i < length (xs @ ys) and length xs + j < length (xs @ ys)
    using assms by simp-all
  moreover have i ≠ length xs + j
    using assms(2) by simp
  ultimately have (xs @ ys) ! i ∩ (xs @ ys) ! (length xs + j) = {}
    by (rule pairwise-disjoint-nthD[OF assms(1)])
  moreover have (xs @ ys) ! i = xs ! i
    using assms(2) by (simp add: nth-append)
  moreover have (xs @ ys) ! (length xs + j) = ys ! j
    using assms(3) by (simp add: nth-append)
  ultimately show ?thesis
    by simp
qed

lemma pairwise-disjoint-pair-imagesI:
  fixes xs :: ('a set × ('a ⇒ 'b)) list
  assumes distinct: distinct xs
    and disj: ⋀x y. ⟦x ∈ set xs; y ∈ set xs; x ≠ y⟧
      ⟹ snd x ` fst x ∩ snd y ` fst y = {}
  shows pairwise-disjoint (map2 (λg A. g ` A) (map snd xs) (map fst xs))
proof −
  have map-eq: map2 (λg A. g ` A) (map snd xs) (map fst xs) =
      map (λx. snd x ` fst x) xs
```

by (*induction xs*) *simp-all*
show *?thesis*
unfolding *map-eq pairwise-disjoint-def*
proof (*intro allI impI*)
fix *i j*
assume *i*: $i < length\ (map\ (\lambda x.\ snd\ x\ `\ fst\ x)\ xs)$
and *j*: $j < length\ (map\ (\lambda x.\ snd\ x\ `\ fst\ x)\ xs)$ **and** *ij*: $i \neq j$
have *xi*: $xs\ !\ i \in set\ xs$ **and** *xj*: $xs\ !\ j \in set\ xs$
using *i j* **by** *auto*
have *neq*: $xs\ !\ i \neq xs\ !\ j$
using *distinct i j ij* **by** (*simp add*: *nth-eq-iff-index-eq*)
show $map\ (\lambda x.\ snd\ x\ `\ fst\ x)\ xs\ !\ i \cap$
$map\ (\lambda x.\ snd\ x\ `\ fst\ x)\ xs\ !\ j = \{\}$
using *disj*[*OF xi xj neq*] *i j* **by** *simp*
qed
qed

3 Transporting finite partitions

definition *transfer-piece* ::
$'a\ set\ list \Rightarrow ('a \Rightarrow 'a)\ list \Rightarrow 'a\ set\ list \Rightarrow$
$('a \Rightarrow 'a)\ list \Rightarrow 'a\ set\ list \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ set$
where
transfer-piece A e P g B k i l =
$\{x \in A\ !\ k.\ (e\ !\ k)\ x \in P\ !\ i \wedge (g\ !\ i)\ ((e\ !\ k)\ x) \in B\ !\ l\}$

definition *transfer-map* ::
$('a \Rightarrow 'a)\ list \Rightarrow ('a \Rightarrow 'a)\ list \Rightarrow$
$('a \Rightarrow 'a)\ list \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow ('a \Rightarrow 'a)$
where
transfer-map t g e k i l = $(t\ !\ l) \circ (g\ !\ i) \circ (e\ !\ k)$

definition *transfer-pairs* ::
$'a\ set\ list \Rightarrow ('a \Rightarrow 'a)\ list \Rightarrow 'a\ set\ list \Rightarrow$
$('a \Rightarrow 'a)\ list \Rightarrow 'a\ set\ list \Rightarrow ('a \Rightarrow 'a)\ list \Rightarrow$
$('a\ set \times ('a \Rightarrow 'a))\ set$
where
transfer-pairs A e P g B t =
$\{(transfer\text{-}piece\ A\ e\ P\ g\ B\ k\ i\ l,\ transfer\text{-}map\ t\ g\ e\ k\ i\ l)\ |$
$k\ i\ l.\ k < length\ A \wedge i < length\ P \wedge l < length\ B\}$

lemma *finite-transfer-pairs* [*simp*]: *finite* (*transfer-pairs A e P g B t*)
proof −
have *transfer-pairs A e P g B t* $\subseteq$
$(\lambda((k,\ i),\ l).\ (transfer\text{-}piece\ A\ e\ P\ g\ B\ k\ i\ l,\ transfer\text{-}map\ t\ g\ e\ k\ i\ l))\ `$
$((\{..<length\ A\} \times \{..<length\ P\}) \times \{..<length\ B\})$
proof
fix *p*
assume *p* $\in$ *transfer-pairs A e P g B t*

```
    then obtain k i l where k: k < length A and i: i < length P and l: l < length
B
      and p-eq: p = (transfer-piece A e P g B k i l, transfer-map t g e k i l)
      unfolding transfer-pairs-def by blast
    have tril-in: ((k, i), l) ∈ ({..<length A} × {..<length P}) × {..<length B}
      using k i l by simp
    show p ∈
      (λ((k, i), l). (transfer-piece A e P g B k i l, transfer-map t g e k i l)) ‘
        (({..<length A} × {..<length P}) × {..<length B})
      using p-eq tril-in by force
  qed
  thus ?thesis
    by (rule finite-subset) simp
qed

lemma transfer-source-cover:
  assumes A-cover: Y = (⋃k<length A. A ! k)
    and B-cover: X = (⋃k<length B. B ! k)
    and e-into-B: ⋀k x. ⟦k < length A; x ∈ A ! k⟧ ⟹ (e ! k) x ∈ B ! k
    and len-AB: length A = length B
    and source-cover: X = (⋃i<length P. P ! i) ∪ (⋃j<length Q. Q ! j)
    and imageP-cover: X = (⋃i<length P. gP ! i ‘ (P ! i))
    and imageQ-cover: X = (⋃j<length Q. gQ ! j ‘ (Q ! j))
  shows Y =
    (⋃p∈transfer-pairs A e P gP B t. fst p) ∪
    (⋃q∈transfer-pairs A e Q gQ B t. fst q)
proof
  show Y ⊆
      (⋃p∈transfer-pairs A e P gP B t. fst p) ∪
      (⋃q∈transfer-pairs A e Q gQ B t. fst q)
  proof
    fix x
    assume xY: x ∈ Y
    then obtain k where k: k < length A and xA: x ∈ A ! k
      using A-cover by blast
    have ex: (e ! k) x ∈ X
      using B-cover e-into-B k len-AB xA by auto
    from source-cover ex have ex-cases:
      (e ! k) x ∈ (⋃i<length P. P ! i) ∨
       (e ! k) x ∈ (⋃j<length Q. Q ! j)
      by blast
    thus x ∈
        (⋃p∈transfer-pairs A e P gP B t. fst p) ∪
        (⋃q∈transfer-pairs A e Q gQ B t. fst q)
    proof
      assume (e ! k) x ∈ (⋃i<length P. P ! i)
      then obtain i where i: i < length P and xP: (e ! k) x ∈ P ! i
        by blast
      have gxX: (gP ! i) ((e ! k) x) ∈ X
```

```
      using imageP-cover i xP by blast
    then obtain l where l: l < length B and gxB: (gP ! i) ((e ! k) x) ∈ B ! l
      using B-cover by blast
    have pair-in: (transfer-piece A e P gP B k i l, transfer-map t gP e k i l)
      ∈ transfer-pairs A e P gP B t
      using k i l unfolding transfer-pairs-def by blast
    have x-piece: x ∈ transfer-piece A e P gP B k i l
      using xA xP gxB by (simp add: transfer-piece-def)
    have set-in: fst (transfer-piece A e P gP B k i l, transfer-map t gP e k i l) ∈
      ((λp. fst p) ‘ transfer-pairs A e P gP B t)
      by (rule imageI[OF pair-in])
    have x-fst: x ∈ fst (transfer-piece A e P gP B k i l, transfer-map t gP e k i l)
      using x-piece by simp
    have x ∈ ⋃ ((λp. fst p) ‘ transfer-pairs A e P gP B t)
      using set-in x-fst by (rule UnionI)
    thus ?thesis
      by simp
  next
    assume (e ! k) x ∈ (⋃j<length Q. Q ! j)
    then obtain j where j: j < length Q and xQ: (e ! k) x ∈ Q ! j
      by blast
    have gxX: (gQ ! j) ((e ! k) x) ∈ X
      using imageQ-cover j xQ by blast
    then obtain l where l: l < length B and gxB: (gQ ! j) ((e ! k) x) ∈ B ! l
      using B-cover by blast
    have pair-in: (transfer-piece A e Q gQ B k j l, transfer-map t gQ e k j l)
      ∈ transfer-pairs A e Q gQ B t
      using k j l unfolding transfer-pairs-def by blast
    have x-piece: x ∈ transfer-piece A e Q gQ B k j l
      using xA xQ gxB by (simp add: transfer-piece-def)
    have set-in: fst (transfer-piece A e Q gQ B k j l, transfer-map t gQ e k j l) ∈
      ((λq. fst q) ‘ transfer-pairs A e Q gQ B t)
      by (rule imageI[OF pair-in])
    have x-fst: x ∈ fst (transfer-piece A e Q gQ B k j l, transfer-map t gQ e k j l)
      using x-piece by simp
    have x ∈ ⋃ ((λq. fst q) ‘ transfer-pairs A e Q gQ B t)
      using set-in x-fst by (rule UnionI)
    thus ?thesis
      by simp
  qed
 qed
 show (⋃p∈transfer-pairs A e P gP B t. fst p) ∪
   (⋃q∈transfer-pairs A e Q gQ B t. fst q) ⊆ Y
  using A-cover unfolding transfer-pairs-def transfer-piece-def by auto
qed

lemma transfer-image-cover:
 assumes A-cover: Y = (⋃k<length A. A ! k)
  and B-cover: X = (⋃k<length B. B ! k)
```

```
    and len-AB: length A = length B
    and e-left: ⋀k x. ⟦k < length A; x ∈ A ! k⟧
      ⟹ (e ! k) x ∈ B ! k ∧ (t ! k) ((e ! k) x) = x
    and t-left: ⋀k y. ⟦k < length B; y ∈ B ! k⟧
      ⟹ (t ! k) y ∈ A ! k ∧ (e ! k) ((t ! k) y) = y
    and P-sub: ⋀i. i < length P ⟹ P ! i ⊆ X
    and image-cover: X = (⋃i<length P. g ! i ` (P ! i))
  shows Y = (⋃p∈transfer-pairs A e P g B t. snd p ` fst p)
proof
  show Y ⊆ (⋃p∈transfer-pairs A e P g B t. snd p ` fst p)
  proof
    fix y
    assume yY: y ∈ Y
    then obtain l where l: l < length A and yA: y ∈ A ! l
      using A-cover by blast
    have eyB: (e ! l) y ∈ B ! l and tey: (t ! l) ((e ! l) y) = y
      using e-left[OF l yA] by auto
    have eyX: (e ! l) y ∈ X
      using B-cover eyB l len-AB by auto
    then obtain i x where i: i < length P and xP: x ∈ P ! i
      and ey-eq: (e ! l) y = (g ! i) x
      using image-cover by blast
    have lB: l < length B
      using l len-AB by simp
    have xX: x ∈ X
      using P-sub[OF i] xP by (rule subsetD)
    then obtain k where k: k < length B and xB: x ∈ B ! k
      using B-cover by blast
    have txA: (t ! k) x ∈ A ! k and etx: (e ! k) ((t ! k) x) = x
      using t-left[OF k xB] by auto
    have kA: k < length A
      using k len-AB by simp
    have piece-in: (t ! k) x ∈ transfer-piece A e P g B k i l
      using txA etx xP eyB ey-eq by (simp add: transfer-piece-def)
    have pair-in: (transfer-piece A e P g B k i l, transfer-map t g e k i l)
       ∈ transfer-pairs A e P g B t
      using kA i lB unfolding transfer-pairs-def by blast
    have transfer-map t g e k i l ((t ! k) x) = y
      using etx ey-eq tey by (simp add: transfer-map-def)
    thus y ∈ (⋃p∈transfer-pairs A e P g B t. snd p ` fst p)
      using pair-in piece-in by force
  qed
  show (⋃p∈transfer-pairs A e P g B t. snd p ` fst p) ⊆ Y
  proof
    fix y
    assume y ∈ (⋃p∈transfer-pairs A e P g B t. snd p ` fst p)
    then obtain k i l x where k: k < length A and i: i < length P
      and l: l < length B and x-piece: x ∈ transfer-piece A e P g B k i l
      and y-eq: y = transfer-map t g e k i l x
```

```
    unfolding transfer-pairs-def by auto
  have gxB: (g ! i) ((e ! k) x) ∈ B ! l
    using x-piece by (simp add: transfer-piece-def)
  have tyA: (t ! l) ((g ! i) ((e ! k) x)) ∈ A ! l
    using t-left[OF l gxB] by auto
  hence y ∈ A ! l
    using y-eq by (simp add: transfer-map-def)
  moreover have A ! l ⊆ Y
    using A-cover l len-AB by auto
  ultimately show y ∈ Y
    by blast
 qed
qed

lemma transfer-source-disjoint-same:
 assumes A-disj: pairwise-disjoint A
   and P-disj: pairwise-disjoint P
   and B-disj: pairwise-disjoint B
   and x-in: x ∈ transfer-pairs A e P g B t
   and y-in: y ∈ transfer-pairs A e P g B t
   and xy: x ≠ y
 shows fst x ∩ fst y = {}
proof (rule ccontr)
 assume nonempty: fst x ∩ fst y ≠ {}
 obtain k i l where k: k < length A and i: i < length P and l: l < length B
   and x-eq: x = (transfer-piece A e P g B k i l, transfer-map t g e k i l)
   using x-in unfolding transfer-pairs-def by blast
 obtain k' i' l' where k': k' < length A and i': i' < length P and l': l' < length
B
   and y-eq: y = (transfer-piece A e P g B k' i' l', transfer-map t g e k' i' l')
   using y-in unfolding transfer-pairs-def by blast
 from nonempty obtain z where zx: z ∈ fst x and zy: z ∈ fst y
   by blast
 have zA: z ∈ A ! k and zeP: (e ! k) z ∈ P ! i
   and zgB: (g ! i) ((e ! k) z) ∈ B ! l
   using zx x-eq by (auto simp: transfer-piece-def)
 have zA': z ∈ A ! k' and zeP': (e ! k') z ∈ P ! i'
   and zgB': (g ! i') ((e ! k') z) ∈ B ! l'
   using zy y-eq by (auto simp: transfer-piece-def)
 show False
 proof (cases k = k')
   case False
   have A ! k ∩ A ! k' = {}
     by (rule pairwise-disjoint-nthD[OF A-disj k k' False])
   with zA zA' show False
     by blast
 next
   case k-eq: True
   show False
```

```
    proof (cases i = i')
      case False
      have P ! i ∩ P ! i' = {}
        by (rule pairwise-disjoint-nthD[OF P-disj i i' False])
      moreover have (e ! k) z ∈ P ! i'
        using zeP' k-eq by simp
      ultimately show False
        using zeP by blast
    next
      case i-eq: True
      show False
      proof (cases l = l')
        case False
        have B ! l ∩ B ! l' = {}
          by (rule pairwise-disjoint-nthD[OF B-disj l l' False])
        moreover have (g ! i) ((e ! k) z) ∈ B ! l'
          using zgB' k-eq i-eq by simp
        ultimately show False
          using zgB by blast
      next
        case l-eq: True
        have x = y
          using x-eq y-eq k-eq i-eq l-eq by simp
        with xy show False
          by contradiction
      qed
    qed
  qed
qed

lemma transfer-source-disjoint-cross:
  assumes A-disj: pairwise-disjoint A
    and PQ-cross: ⋀i j. ⟦i < length P; j < length Q⟧ ⟹ P ! i ∩ Q ! j = {}
    and B-disj: pairwise-disjoint B
    and x-in: x ∈ transfer-pairs A e P gP B t
    and y-in: y ∈ transfer-pairs A e Q gQ B t
  shows fst x ∩ fst y = {}
proof (rule ccontr)
  assume nonempty: fst x ∩ fst y ≠ {}
  obtain k i l where k: k < length A and i: i < length P and l: l < length B
    and x-eq: x = (transfer-piece A e P gP B k i l, transfer-map t gP e k i l)
    using x-in unfolding transfer-pairs-def by blast
  obtain k' j l' where k': k' < length A and j: j < length Q and l': l' < length B
    and y-eq: y = (transfer-piece A e Q gQ B k' j l', transfer-map t gQ e k' j l')
    using y-in unfolding transfer-pairs-def by blast
  from nonempty obtain z where zx: z ∈ fst x and zy: z ∈ fst y
    by blast
  have zA: z ∈ A ! k and zeP: (e ! k) z ∈ P ! i
    using zx x-eq by (auto simp: transfer-piece-def)
```

```
  have zA': z ∈ A ! k' and zeQ: (e ! k') z ∈ Q ! j
    using zy y-eq by (auto simp: transfer-piece-def)
  show False
  proof (cases k = k')
    case False
    have A ! k ∩ A ! k' = {}
      by (rule pairwise-disjoint-nthD[OF A-disj k k' False])
    with zA zA' show False
      by blast
  next
    case k-eq: True
    have P ! i ∩ Q ! j = {}
      by (rule PQ-cross[OF i j])
    moreover have (e ! k) z ∈ Q ! j
      using zeQ k-eq by simp
    ultimately show False
      using zeP by blast
  qed
qed

lemma transfer-image-disjoint-same:
  assumes len-AB: length A = length B
    and A-disj: pairwise-disjoint A
    and B-disj: pairwise-disjoint B
    and image-disj: pairwise-disjoint (map2 (λh C. h ` C) g P)
    and len: length P = length g
    and g-inj: ⋀i. i < length P ⟹ inj (g ! i)
    and e-into-B: ⋀k x. ⟦k < length A; x ∈ A ! k⟧ ⟹ (e ! k) x ∈ B ! k
    and t-left: ⋀l y. ⟦l < length B; y ∈ B ! l⟧
      ⟹ (t ! l) y ∈ A ! l ∧ (e ! l) ((t ! l) y) = y
    and x-in: x ∈ transfer-pairs A e P g B t
    and y-in: y ∈ transfer-pairs A e P g B t
    and xy: x ≠ y
  shows snd x ` fst x ∩ snd y ` fst y = {}
proof (rule ccontr)
  assume nonempty: snd x ` fst x ∩ snd y ` fst y ≠ {}
  obtain k i l where k: k < length A and i: i < length P and l: l < length B
    and x-eq: x = (transfer-piece A e P g B k i l, transfer-map t g e k i l)
    using x-in unfolding transfer-pairs-def by blast
  obtain k' i' l' where k': k' < length A and i': i' < length P and l': l' < length
B
    and y-eq: y = (transfer-piece A e P g B k' i' l', transfer-map t g e k' i' l')
    using y-in unfolding transfer-pairs-def by blast
  from nonempty obtain z where zx: z ∈ snd x ` fst x and zy: z ∈ snd y ` fst y
    by blast
  then obtain a b where
      a-piece: a ∈ transfer-piece A e P g B k i l
    and z-a: z = transfer-map t g e k i l a
    and b-piece: b ∈ transfer-piece A e P g B k' i' l'
```

```
  and z-b: z = transfer-map t g e k' i' l' b
  using x-eq y-eq by auto
have aA: a ∈ A ! k and aeP: (e ! k) a ∈ P ! i
  and gaB: (g ! i) ((e ! k) a) ∈ B ! l
  using a-piece by (auto simp: transfer-piece-def)
have bA: b ∈ A ! k' and beP: (e ! k') b ∈ P ! i'
  and gbB: (g ! i') ((e ! k') b) ∈ B ! l'
  using b-piece by (auto simp: transfer-piece-def)
have lA: l < length A and lA': l' < length A
  using l l' len-AB by simp-all
have zA-l: z ∈ A ! l
  using t-left[OF l gaB] z-a by (simp add: transfer-map-def)
have zA-l': z ∈ A ! l'
  using t-left[OF l' gbB] z-b by (simp add: transfer-map-def)
show False
proof (cases l = l')
  case False
  have A ! l ∩ A ! l' = {}
    by (rule pairwise-disjoint-nthD[OF A-disj lA lA' False])
  with zA-l zA-l' show False
    by blast
next
  case l-eq: True
  have ez-a: (e ! l) z = (g ! i) ((e ! k) a)
    using t-left[OF l gaB] z-a by (simp add: transfer-map-def)
  have ez-b: (e ! l) z = (g ! i') ((e ! k') b)
    using t-left[OF l' gbB] z-b l-eq by (simp add: transfer-map-def)
  have g-eq: (g ! i) ((e ! k) a) = (g ! i') ((e ! k') b)
    using ez-a ez-b by simp
  show False
  proof (cases i = i')
    case False then
    have img-i: (g ! i) ((e ! k) a) ∈ map2 (λh C. h ‘ C) g P ! i
      using aeP i len by simp
    have img-i': (g ! i') ((e ! k') b) ∈ map2 (λh C. h ‘ C) g P ! i'
      using beP i' len by simp
    have map2 (λh C. h ‘ C) g P ! i ∩
       map2 (λh C. h ‘ C) g P ! i' = {}
    by (rule pairwise-disjoint-nthD[OF image-disj]) (use i i' False len in simp-all)
    moreover have (g ! i) ((e ! k) a) ∈ map2 (λh C. h ‘ C) g P ! i'
      using img-i' g-eq by simp
    ultimately show False
      using img-i by auto
  next
    case i-eq: True
    have e-eq: (e ! k) a = (e ! k') b
      using g-eq i-eq g-inj[OF i] by (auto dest: injD)
    show False
    proof (cases k = k')
```

```
      case False
      have kB: k < length B and kB': k' < length B
        using k k' len-AB by simp-all
      have eaB: (e ! k) a ∈ B ! k
        by (rule e-into-B[OF k aA])
      have ebB: (e ! k') b ∈ B ! k'
        by (rule e-into-B[OF k' bA])
      have B ! k ∩ B ! k' = {}
        by (rule pairwise-disjoint-nthD[OF B-disj kB kB' False])
      moreover have (e ! k) a ∈ B ! k'
        using ebB e-eq by simp
      ultimately show False
        using eaB by auto
    next
      case k-eq: True
      have x = y
        using x-eq y-eq k-eq i-eq l-eq by simp
      with xy show False
        by contradiction
    qed
  qed
 qed
qed

lemma transfer-partitioned-paradox:
  assumes len-AB: length A = length B
    and A-cover: Y = (⋃k<length A. A ! k)
    and A-disj: pairwise-disjoint A
    and B-cover: X = (⋃k<length B. B ! k)
    and B-disj: pairwise-disjoint B
    and e-left: ⋀k x. ⟦k < length A; x ∈ A ! k⟧
      ⟹ (e ! k) x ∈ B ! k ∧ (t ! k) ((e ! k) x) = x
    and t-left: ⋀k y. ⟦k < length B; y ∈ B ! k⟧
      ⟹ (t ! k) y ∈ A ! k ∧ (e ! k) ((t ! k) y) = y
    and lenP: length P = length gP
    and lenQ: length Q = length gQ
    and source-disj: pairwise-disjoint (P @ Q)
    and source-cover: X = (⋃i<length P. P ! i) ∪ (⋃j<length Q. Q ! j)
    and imageP-disj: pairwise-disjoint (map2 (λh C. h ‘ C) gP P)
    and imageQ-disj: pairwise-disjoint (map2 (λh C. h ‘ C) gQ Q)
    and imageP-cover: X = (⋃i<length P. gP ! i ‘ (P ! i))
    and imageQ-cover: X = (⋃j<length Q. gQ ! j ‘ (Q ! j))
    and gP-inj: ⋀i. i < length P ⟹ inj (gP ! i)
    and gQ-inj: ⋀j. j < length Q ⟹ inj (gQ ! j)
    and mapsP: ⋀k i l. ⟦k < length A; i < length P; l < length B⟧
      ⟹ transfer-map t gP e k i l ∈ M
    and mapsQ: ⋀k j l. ⟦k < length A; j < length Q; l < length B⟧
      ⟹ transfer-map t gQ e k j l ∈ M
  shows ∃P' Q' :: 'a set list. ∃gP' gQ' :: ('a ⇒ 'a) list.
```

```
      length P' = length gP' ∧ length Q' = length gQ' ∧
      set gP' ⊆ M ∧ set gQ' ⊆ M ∧
      pairwise-disjoint (P' @ Q') ∧
      pairwise-disjoint (map2 (λh C. h ‘ C) gP' P') ∧
      pairwise-disjoint (map2 (λh C. h ‘ C) gQ' Q') ∧
      Y = (⋃i<length P'. P' ! i) ∪ (⋃j<length Q'. Q' ! j) ∧
      Y = (⋃i<length P'. gP' ! i ‘ (P' ! i)) ∧
      Y = (⋃j<length Q'. gQ' ! j ‘ (Q' ! j))
proof −
  let ?CP = transfer-pairs A e P gP B t
  let ?CQ = transfer-pairs A e Q gQ B t
  have finite-CP: finite ?CP
    by simp
  have finite-CQ: finite ?CQ
    by simp
  obtain pairsP where setP: set pairsP = ?CP and distinctP: distinct pairsP
    using finite-distinct-list[OF finite-CP] by blast
  obtain pairsQ where setQ: set pairsQ = ?CQ and distinctQ: distinct pairsQ
    using finite-distinct-list[OF finite-CQ] by blast
  let ?P' = map fst pairsP
  let ?Q' = map fst pairsQ
  let ?gP' = map snd pairsP
  let ?gQ' = map snd pairsQ

  have P-disj: pairwise-disjoint P
    by (rule pairwise-disjoint-appendD1[OF source-disj])
  have Q-disj: pairwise-disjoint Q
    by (rule pairwise-disjoint-appendD2[OF source-disj])
  have PQ-cross: ⋀i j. ⟦i < length P; j < length Q⟧ ⟹ P ! i ∩ Q ! j = {}
    by (rule pairwise-disjoint-append-crossD[OF source-disj])

  have P'-disj: pairwise-disjoint ?P'
  proof (rule pairwise-disjoint-map-fstI[OF distinctP])
    fix x y
    assume x ∈ set pairsP y ∈ set pairsP x ≠ y
    thus fst x ∩ fst y = {}
      using setP transfer-source-disjoint-same[OF A-disj P-disj B-disj] by blast
  qed
  have Q'-disj: pairwise-disjoint ?Q'
    by (metis A-disj B-disj Q-disj distinctQ pairwise-disjoint-map-fstI setQ trans-
fer-source-disjoint-same)
  have cross-disj: ⋀x y. ⟦x ∈ set ?P'; y ∈ set ?Q'⟧ ⟹ x ∩ y = {}
  proof −
    fix x y
    assume x: x ∈ set ?P' and y: y ∈ set ?Q'
    then obtain px qy where px: px ∈ set pairsP and qy: qy ∈ set pairsQ
      and x-eq: x = fst px and y-eq: y = fst qy
      by auto
    show x ∩ y = {}
```

```
    using transfer-source-disjoint-cross[OF A-disj PQ-cross B-disj]
      setP setQ px qy x-eq y-eq by blast
  qed
  have source-disj': pairwise-disjoint (?P' @ ?Q')
    by (rule pairwise-disjoint-appendI[OF P'-disj Q'-disj cross-disj])

  have imageP-disj': pairwise-disjoint (map2 (λh C. h ‘ C) ?gP' ?P')
  proof (rule pairwise-disjoint-pair-imagesI[OF distinctP])
    fix x y
    assume x ∈ set pairsP y ∈ set pairsP x ≠ y
    thus snd x ‘ fst x ∩ snd y ‘ fst y = {}
      using setP transfer-image-disjoint-same[
        OF len-AB A-disj B-disj imageP-disj lenP gP-inj - t-left]
        e-left by blast
  qed
  have imageQ-disj': pairwise-disjoint (map2 (λh C. h ‘ C) ?gQ' ?Q')
  proof (rule pairwise-disjoint-pair-imagesI[OF distinctQ])
    fix x y
    assume x ∈ set pairsQ y ∈ set pairsQ x ≠ y
    thus snd x ‘ fst x ∩ snd y ‘ fst y = {}
      using setQ transfer-image-disjoint-same[
        OF len-AB A-disj B-disj imageQ-disj lenQ gQ-inj - t-left]
        e-left by blast
  qed

  have P-sub: ⋀i. i < length P ⟹ P ! i ⊆ X
    using source-cover by blast
  have Q-sub: ⋀j. j < length Q ⟹ Q ! j ⊆ X
    using source-cover by blast

  have source-cover': Y = (⋃i<length ?P'. ?P' ! i) ∪ (⋃j<length ?Q'. ?Q' ! j)
  proof −
    have Y = (⋃p∈?CP. fst p) ∪ (⋃q∈?CQ. fst q)
      by (rule transfer-source-cover[OF A-cover B-cover - len-AB source-cover
imageP-cover imageQ-cover])
       (use e-left in auto)
    also have ... = (⋃i<length ?P'. ?P' ! i) ∪ (⋃j<length ?Q'. ?Q' ! j)
      using setP setQ indexed-union-map-fst[of pairsP] indexed-union-map-fst[of
pairsQ]
      by simp
    finally show ?thesis .
  qed
  have imageP-cover': Y = (⋃i<length ?P'. ?gP' ! i ‘ (?P' ! i))
  proof −
    have Y = (⋃p∈?CP. snd p ‘ fst p)
      by (rule transfer-image-cover[OF A-cover B-cover len-AB e-left t-left P-sub
imageP-cover])
    also have ... = (⋃i<length ?P'. ?gP' ! i ‘ (?P' ! i))
      using setP indexed-image-union-pairs[of pairsP] by simp
```

```
    finally show ?thesis .
  qed
  have imageQ-cover': Y = (⋃j<length ?Q'. ?gQ' ! j ‘ (?Q' ! j))
  proof −
    have Y = (⋃ q∈?CQ. snd q ‘ fst q)
      by (rule transfer-image-cover[OF A-cover B-cover len-AB e-left t-left Q-sub
imageQ-cover])
    also have ... = (⋃j<length ?Q'. ?gQ' ! j ‘ (?Q' ! j))
      using setQ indexed-image-union-pairs[of pairsQ] by simp
    finally show ?thesis .
  qed

  have mapsP': set ?gP' ⊆ M
    using setP mapsP unfolding transfer-pairs-def by auto
  have mapsQ': set ?gQ' ⊆ M
    using setQ mapsQ unfolding transfer-pairs-def by auto

  show ?thesis
  proof (intro exI conjI)
    show length ?P' = length ?gP'
      by simp
    show length ?Q' = length ?gQ'
      by simp
    show set ?gP' ⊆ M
      by (rule mapsP')
    show set ?gQ' ⊆ M
      by (rule mapsQ')
    show pairwise-disjoint (?P' @ ?Q')
      by (rule source-disj')
    show pairwise-disjoint (map2 (λh C. h ‘ C) ?gP' ?P')
      by (rule imageP-disj')
    show pairwise-disjoint (map2 (λh C. h ‘ C) ?gQ' ?Q')
      by (rule imageQ-disj')
    show Y = (⋃i<length ?P'. ?P' ! i) ∪ (⋃j<length ?Q'. ?Q' ! j)
      by (rule source-cover')
    show Y = (⋃i<length ?P'. ?gP' ! i ‘ (?P' ! i))
      by (rule imageP-cover')
    show Y = (⋃j<length ?Q'. ?gQ' ! j ‘ (?Q' ! j))
      by (rule imageQ-cover')
  qed
qed

end


theory Paradoxical-Decomposition
  imports BT-Prelim
begin
```

4 Group actions on a set

An action of a (carrier) group G on a set X is a function *act* respecting the unit and multiplication on X. We do not fix the group structure here; rather, we record the algebraic laws that an action must satisfy. Concrete instances will be given in later theories (the action of F_2 on itself by left translation; the action of $SO(3)$ on S^2).

locale *group-action* =
  fixes *carrier* :: *'g set*
    and *unit* :: *'g*
    and *mult* :: *'g* ⇒ *'g* ⇒ *'g* (**infixl** ‹·› *70*)
    and *act* :: *'g* ⇒ *'a* ⇒ *'a*
    and *ground* :: *'a set*
  assumes *unit-carrier* [*simp*]: *unit* ∈ *carrier*
    and *mult-closed* [*intro*]: ⟦*g* ∈ *carrier*; *h* ∈ *carrier*⟧ ⟹ *g* · *h* ∈ *carrier*
    and *act-closed* [*intro*]: ⟦*g* ∈ *carrier*; *x* ∈ *ground*⟧ ⟹ *act g x* ∈ *ground*
    and *act-unit* [*simp*]: *x* ∈ *ground* ⟹ *act unit x* = *x*
    and *act-mult*: ⟦*g* ∈ *carrier*; *h* ∈ *carrier*; *x* ∈ *ground*⟧
      ⟹ *act* (*g* · *h*) *x* = *act g* (*act h x*)

context *group-action*
begin

definition *orbit* :: *'a* ⇒ *'a set* **where**
  orbit x = {*y*. ∃ *g* ∈ *carrier*. *y* = *act g x*}

definition *image-set* :: *'g* ⇒ *'a set* ⇒ *'a set* **where**
  image-set g A = *act g* ‘ *A*

lemma *image-set-unit*:
  assumes *A* ⊆ *ground*
  shows *image-set unit A* = *A*
    using *assms* **unfolding** *image-set-def* **by** *force*

lemma *orbit-self*:
  assumes *x* ∈ *ground*
  shows *x* ∈ *orbit x*
  using *assms* **by** (*simp add*: *orbit-def*) (*metis act-unit unit-carrier*)

end

5 Paradoxical decomposition

A subset X of the underlying set of a group action is *paradoxical* when it admits two finite collections of pairwise disjoint pieces, all contained in X, such that each collection can be translated (by group elements) so that the resulting *disjoint* union is the whole of X.

This is the standard Banach-Tarski definition. We will instantiate it twice: once for F_2 acting on itself, and once for the rotation group acting on S^2 (and ultimately the ball).

context *group-action*
begin

definition *paradoxical* :: *'a set* ⇒ *bool* **where**
paradoxical X ⟷
(∃ *P Q* :: *'a set list*. ∃ *gP gQ* :: *'g list*.
length P = *length gP* ∧ *length Q* = *length gQ* ∧
set gP ⊆ *carrier* ∧ *set gQ* ⊆ *carrier* ∧
pairwise-disjoint (*P* @ *Q*) ∧
pairwise-disjoint (*map2 image-set gP P*) ∧
pairwise-disjoint (*map2 image-set gQ Q*) ∧
(⋃ *i*<*length P*. *P* ! *i*) ∪ (⋃ *i*<*length Q*. *Q* ! *i*) ⊆ *X* ∧
X = (⋃ *i*<*length P*. *image-set* (*gP* ! *i*) (*P* ! *i*)) ∧
X = (⋃ *i*<*length Q*. *image-set* (*gQ* ! *i*) (*Q* ! *i*)))

A specialised constructor for the most common case: each of P and Q has exactly two pieces. This matches the F_2 decomposition (split into the a / a^{-1} classes and the b / b^{-1} classes).

lemma *paradoxical-two-two*:
assumes *p1* ∈ *carrier p2* ∈ *carrier q1* ∈ *carrier q2* ∈ *carrier*
and *disj-P12*: *P1* ∩ *P2* = {}
and *disj-Q12*: *Q1* ∩ *Q2* = {}
and *disj-PQ*: (*P1* ∪ *P2*) ∩ (*Q1* ∪ *Q2*) = {}
and *sub*: *P1* ∪ *P2* ∪ *Q1* ∪ *Q2* ⊆ *X*
and *disj-imgP*: *image-set p1 P1* ∩ *image-set p2 P2* = {}
and *disj-imgQ*: *image-set q1 Q1* ∩ *image-set q2 Q2* = {}
and *cover-P*: *X* = *image-set p1 P1* ∪ *image-set p2 P2*
and *cover-Q*: *X* = *image-set q1 Q1* ∪ *image-set q2 Q2*
shows *paradoxical X*
proof −
let *?P* = [*P1*, *P2*] **and** *?Q* = [*Q1*, *Q2*]
let *?gP* = [*p1*, *p2*] **and** *?gQ* = [*q1*, *q2*]
have *lens*: *length ?P* = *length ?gP length ?Q* = *length ?gQ* **by** *simp-all*
have *closed*: *set ?gP* ⊆ *carrier set ?gQ* ⊆ *carrier*
using *assms*(*1*−*4*) **by** *auto*
have *disj-all*: *pairwise-disjoint* (*?P* @ *?Q*)
using *disj-P12 disj-Q12 disj-PQ*
by (*auto simp*: *pairwise-disjoint-def nth-Cons' nth-append Int-commute Int-Un-distrib*)
have *map2-P*: *map2 image-set ?gP ?P* = [*image-set p1 P1*, *image-set p2 P2*] **by** *simp*
have *map2-Q*: *map2 image-set ?gQ ?Q* = [*image-set q1 Q1*, *image-set q2 Q2*] **by** *simp*
have *disj-map-P*: *pairwise-disjoint* (*map2 image-set ?gP ?P*)
using *disj-imgP* **unfolding** *map2-P*
by (*auto simp*: *pairwise-disjoint-def nth-Cons' Int-commute*)

```
  have disj-map-Q: pairwise-disjoint (map2 image-set ?gQ ?Q)
    using disj-imgQ unfolding map2-Q
    by (auto simp: pairwise-disjoint-def nth-Cons' Int-commute)
  have sub-un: (⋃i<length ?P. ?P ! i) ∪ (⋃i<length ?Q. ?Q ! i) ⊆ X
    using sub by (auto simp: lessThan-Suc)
  have cov-P': X = (⋃i<length ?P. image-set (?gP ! i) (?P ! i))
    using cover-P by (auto simp: lessThan-Suc)
  have cov-Q': X = (⋃i<length ?Q. image-set (?gQ ! i) (?Q ! i))
    using cover-Q by (auto simp: lessThan-Suc)
  show ?thesis
    unfolding paradoxical-def
    using lens closed disj-all disj-map-P disj-map-Q sub-un cov-P' cov-Q'
    by blast
qed
```

With the finite-list convention used above, the empty set is paradoxical: take both families of pieces to be empty. The two translated indexed unions are then both the empty union, hence both equal to {}. This degenerate case is harmless in the later non-empty geometric applications, but recording it keeps the abstract definition honest about empty indexed unions.

```
lemma paradoxical-empty: paradoxical {}
  unfolding paradoxical-def
  by (intro exI[where x=[]])
     (auto simp: pairwise-disjoint-def image-set-def)

end
```

6 Equidecomposability

Two subsets X and Y are *equidecomposable* under a group action when they admit congruent partitions: the i-th piece of X maps onto the i-th piece of Y via some group element.

```
context group-action
begin

definition equidecomposable :: 'a set ⇒ 'a set ⇒ bool where
  equidecomposable X Y ⟷
    (∃P Q :: 'a set list. ∃gs :: 'g list.
       length P = length Q ∧ length P = length gs ∧
       set gs ⊆ carrier ∧
       pairwise-disjoint P ∧ pairwise-disjoint Q ∧
       X = (⋃i<length P. P ! i) ∧
       Y = (⋃i<length Q. Q ! i) ∧
       (∀i < length P. Q ! i = image-set (gs ! i) (P ! i)))
```

The main development uses this relation only as background notation: the sphere and ball arguments below work directly with explicit finite lists of

pieces and transformations, so the needed partition data is visible at each transfer step.

end

end

theory *Free-Action-Paradox*
  imports *Paradoxical-Decomposition*
begin

7 Free actions

An action is *free* on a set X if non-identity group elements have no fixed points in X: $g \cdot x = x$ forces $g = \mathit{unit}$.

context *group-action*
begin

definition *free-on* :: *'a set* ⇒ *bool* **where**
  free-on X ⟷
    (∀ *g* ∈ *carrier*. ∀ *x* ∈ *X*. *act g x* = *x* ⟶ *g* = *unit*)

end

8 Choice of orbit representatives

Given a free action of G on X, the orbits partition X, and for any subset of choice representatives $M \subseteq X$ (one per orbit), every $x \in X$ can be written uniquely as *act g m* for some $g \in \mathit{carrier}$ and $m \in M$.

This is where the axiom of choice enters: we pick one element from each orbit.

context *group-action*
begin

definition *orbit-eq* :: *'a* ⇒ *'a* ⇒ *bool* **where**
  orbit-eq x y ⟷ (∃ *g* ∈ *carrier*. *y* = *act g x*)

lemma *orbit-eq-refl*: *x* ∈ *ground* ⟹ *orbit-eq x x*
  unfolding *orbit-eq-def* **using** *act-unit unit-carrier* **by** *metis*

When the action is free on X, the orbit relation is symmetric on X, provided every element has a group inverse in the carrier. We do not require *group*-level inverses here, so we state symmetry conditionally.

end

9 Transport of paradoxical decompositions along free actions

This is the abstract Banach-Tarski reduction: suppose G acts freely on X, and G itself admits a paradoxical decomposition under the regular action (left multiplication on itself). Choosing one representative per G-orbit in X via the axiom of choice transports the paradox: each piece $P_i \subseteq G$ in the decomposition lifts to the union $\{act\ g\ m \mid g \in P_i,\ m \in M\}$, where M is the chosen set of representatives.

The later Hausdorff theory carries out this lift explicitly for $S^2 \setminus D$: it defines orbit representatives, proves the lifted pieces are disjoint, and proves the two translated covers directly. The predicate below only packages the freeness and closure hypotheses that such an argument needs.

context *group-action*
begin

A free action is suitable for orbit-by-orbit transport when the set is contained in the ground space, the action is free on it, and the set is closed under the action of every carrier element.

definition *transports-paradox* :: *'a set* ⇒ *bool* **where**
transports-paradox X ⟷
(*X* ⊆ *ground* ∧
free-on X ∧
(∀ *x* ∈ *X*. ∀ *g* ∈ *carrier*. *act g x* ∈ *X*))

Under the assumptions captured by *transports-paradox*, a paradoxical decomposition of the carrier transports to a paradoxical decomposition of X. The proof is the standard "orbit-by-orbit" argument and uses the axiom of choice to select one representative per orbit.

end

end

theory *Free-Group-F2*
imports
BT-Prelim
Free−Groups.FreeGroups
begin

10 Two-element generator type

We pick a concrete two-element type for the generators of F_2.

datatype *gen2* = *A* | *B*

abbreviation *Gen2* :: *gen2 set* **where**
  Gen2 ≡ {*A*, *B*}

lemma *Gen2-finite* [*simp*]: *finite Gen2*
  by *simp*

lemma *Gen2-card* [*simp*]: *card Gen2* = *2*
  by *simp*

lemma *Gen2-UNIV*: *Gen2* = (*UNIV* :: *gen2 set*)
  using *gen2.exhaust* **by** *auto*

11 The free group F_2 and its carrier

abbreviation *F2* :: (*bool* × *gen2*) *list monoid* **where**
  F2 ≡ $\mathcal{F}_{Gen2}$

The carrier of F_2 consists of cancelled words over the alphabet *UNIV* × *Gen2* = *UNIV* × *UNIV*.

lemma *F2-is-group*: *group F2*
  by (*rule free-group-is-group*)

lemma *F2-carrier-iff*:
  w ∈ *carrier F2* ⟷ *w* ∈ *lists* (*UNIV* × *Gen2*) ∧ *canceled w*
  by (*auto simp*: *free-group-def*)

lemma *F2-carrier-alt*:
  carrier F2 = {*w*. *canceled w*}
proof −
  have *lists* (*UNIV* × *Gen2*) = (*UNIV* :: (*bool* × *gen2*) *list set*)
    using *Gen2-UNIV* **by** *auto*
  thus *?thesis* **by** (*auto simp*: *free-group-def*)
qed

lemma *in-F2-canceled* [*dest*]:
  assumes *w* ∈ *carrier F2*
  shows *canceled w*
  using *assms* **by** (*simp add*: *F2-carrier-alt*)

lemma *canceled-in-F2* [*intro*]:
  assumes *canceled w*
  shows *w* ∈ *carrier F2*
  using *assms* **by** (*simp add*: *F2-carrier-alt*)

lemma *canceled-iff-no-adjacent-canceling*:
  canceled w ⟷ (∀ *i*. *Suc i* < *length w* ⟶ ¬ *canceling* (*w* ! *i*) (*w* ! *Suc i*))
proof
  assume *c*: *canceled w*

```
  show ∀ i. Suc i < length w ⟶ ¬ canceling (w ! i) (w ! Suc i)
  proof (intro allI impI notI)
    fix i
    assume len: Suc i < length w and can: canceling (w ! i) (w ! Suc i)
    have cancels-to-1-at i w (cancel-at i w)
      using len can by (auto simp: cancels-to-1-at-def)
    hence cancels-to-1 w (cancel-at i w)
      unfolding cancels-to-1-def by blast
    hence Domainp cancels-to-1 w
      by auto
    with c show False
      by (simp add: canceled-def)
  qed
next
  assume no-adj: ∀ i. Suc i < length w ⟶ ¬ canceling (w ! i) (w ! Suc i)
  show canceled w
  proof (simp add: canceled-def, intro notI)
    assume Domainp cancels-to-1 w
    then obtain w' where cancels-to-1 w w'
      by auto
    then obtain i where cancels-to-1-at i w w'
      unfolding cancels-to-1-def by auto
    hence Suc i < length w and canceling (w ! i) (w ! Suc i)
      by (auto simp: cancels-to-1-at-def)
    with no-adj show False
      by blast
  qed
qed

lemma canceled-Cons-iff:
  canceled (p # w) ⟷ canceled w ∧ (w = [] ∨ ¬ canceling p (hd w))
  unfolding canceled-iff-no-adjacent-canceling
  by (cases w)
    (auto simp: nth-Cons split: nat.splits)

lemma F2-ConsD:
  assumes p # w ∈ carrier F2
  shows w ∈ carrier F2 and w = [] ∨ ¬ canceling p (hd w)
  using assms by (simp-all add: F2-carrier-alt canceled-Cons-iff)

lemma F2-ConsI:
  assumes w ∈ carrier F2 and w = [] ∨ ¬ canceling p (hd w)
  shows p # w ∈ carrier F2
  using assms by (simp add: F2-carrier-alt canceled-Cons-iff)

lemma F2-one [simp]: 𝟭_F2 = []
  by (simp add: free-group-def)

lemma F2-mult: x ⊗_F2 y = normalize (x @ y)
```

by (*simp add*: *free-group-def*)

12 The four "starts-with" classes of F_2

A non-empty reduced word in F_2 begins with one of four letters: a, a^{-1}, b, or b^{-1}. The corresponding "starts-with" classes partition $F_2 \setminus\setminus \{1\}$.

definition *starts-with* :: *bool* ⇒ *gen2* ⇒ (*bool* × *gen2*) *list set* **where**
starts-with b x = {*w* ∈ *carrier F2*. *w* ≠ [] ∧ *hd w* = (*b*, *x*)}

abbreviation *S-a* :: (*bool* × *gen2*) *list set* **where** *S-a* ≡ *starts-with False A*
abbreviation *S-aI* :: (*bool* × *gen2*) *list set* **where** *S-aI* ≡ *starts-with True A*
abbreviation *S-b* :: (*bool* × *gen2*) *list set* **where** *S-b* ≡ *starts-with False B*
abbreviation *S-bI* :: (*bool* × *gen2*) *list set* **where** *S-bI* ≡ *starts-with True B*

lemma *starts-with-subset*: *starts-with b x* ⊆ *carrier F2*
by (*auto simp*: *starts-with-def*)

lemma *starts-with-disjoint*:
assumes (*b1*, *x1*) ≠ (*b2*, *x2*)
shows *starts-with b1 x1* ∩ *starts-with b2 x2* = {}
using *assms* **by** (*auto simp*: *starts-with-def*)

lemma *starts-with-pairwise-disjoint*:
shows *S-a* ∩ *S-aI* = {} *S-a* ∩ *S-b* = {} *S-a* ∩ *S-bI* = {}
S-aI ∩ *S-b* = {} *S-aI* ∩ *S-bI* = {} *S-b* ∩ *S-bI* = {}
[] ∉ *S-a* [] ∉ *S-aI* [] ∉ *S-b* [] ∉ *S-bI*
by (*auto simp*: *starts-with-def*)

The four classes together with the singleton $\{1\}$ partition the carrier of F_2. The proof is a case analysis on the head of a reduced word.

lemma *F2-decomposition*:
carrier F2 = {[]} ∪ *S-a* ∪ *S-aI* ∪ *S-b* ∪ *S-bI*
proof
show {[]} ∪ *S-a* ∪ *S-aI* ∪ *S-b* ∪ *S-bI* ⊆ *carrier F2*
using *starts-with-subset*
by (*auto simp*: *F2-carrier-alt empty-canceled*)
next
show *carrier F2* ⊆ {[]} ∪ *S-a* ∪ *S-aI* ∪ *S-b* ∪ *S-bI*
proof
fix *w* **assume** *w*: *w* ∈ *carrier F2*
show *w* ∈ {[]} ∪ *S-a* ∪ *S-aI* ∪ *S-b* ∪ *S-bI*
proof (*cases w*)
case *Nil* **thus** *?thesis* **by** *simp*
next
case (*Cons a u*)
obtain *b x* **where** *head*: *a* = (*b*, *x*) **by** (*cases a*) *auto*
from *w* **have** *x* ∈ *Gen2* **using** *Cons head*
by (*auto simp*: *F2-carrier-iff*)

hence $x = A \vee x = B$ **by** *auto*
with *Cons head w* **show** *?thesis*
unfolding *starts-with-def* **by** *auto*
qed
qed
qed

13 The two distinguished generators a and b

abbreviation *a-elt* :: (*bool* × *gen2*) *list* **where** *a-elt* ≡ [(*False*, *A*)]
abbreviation *aI-elt* :: (*bool* × *gen2*) *list* **where** *aI-elt* ≡ [(*True*, *A*)]
abbreviation *b-elt* :: (*bool* × *gen2*) *list* **where** *b-elt* ≡ [(*False*, *B*)]
abbreviation *bI-elt* :: (*bool* × *gen2*) *list* **where** *bI-elt* ≡ [(*True*, *B*)]

lemma *single-letter-canceled*:
canceled [(*b*, *x*)]
by *simp*

lemma *a-elt-in-F2* [*simp*]: *a-elt* ∈ *carrier F2*
by (*intro canceled-in-F2 single-letter-canceled*)

lemma *aI-elt-in-F2* [*simp*]: *aI-elt* ∈ *carrier F2*
by (*intro canceled-in-F2 single-letter-canceled*)

lemma *b-elt-in-F2* [*simp*]: *b-elt* ∈ *carrier F2*
by (*intro canceled-in-F2 single-letter-canceled*)

lemma *bI-elt-in-F2* [*simp*]: *bI-elt* ∈ *carrier F2*
by (*intro canceled-in-F2 single-letter-canceled*)

lemma *inv-a-eq-aI*: inv_{F2} *a-elt* = *aI-elt*
by (*simp add*: *inv-fg-def inv1-def*)

lemma *inv-b-eq-bI*: inv_{F2} *b-elt* = *bI-elt*
by (*simp add*: *inv-fg-def inv1-def*)

lemma *inv-aI-eq-a*: inv_{F2} *aI-elt* = *a-elt*
by (*simp add*: *inv-fg-def inv1-def*)

lemma *inv-bI-eq-b*: inv_{F2} *bI-elt* = *b-elt*
by (*simp add*: *inv-fg-def inv1-def*)

end

theory *F2-Paradox*
imports
Free-Group-F2
Paradoxical-Decomposition

begin

14 The action of F_2 on itself by left multiplication

We interpret *group-action* with carrier *carrier F2* and the action being left multiplication.

lemma *F2-one-in-carrier* [*simp*]: [] ∈ *carrier F2*
by (*metis F2-is-group F2-one group.is-monoid monoid.one-closed*)

interpretation *F2-act*:
group-action carrier F2 [] (⊗$_{F2}$) (⊗$_{F2}$) *carrier F2*
proof *unfold-locales*
show [] ∈ *carrier F2* **by** *simp*
next
fix *g h* **assume** *g* ∈ *carrier F2 h* ∈ *carrier F2*
thus *g* ⊗$_{F2}$ *h* ∈ *carrier F2*
by (*meson F2-is-group group.is-monoid monoid.m-closed*)
next
fix *x* **assume** *xin*: *x* ∈ *carrier F2*
have [] ⊗$_{F2}$ *x* = **1**$_{F2}$ ⊗$_{F2}$ *x* **by** *simp*
also have ... = *x* **using** *xin F2-is-group group.is-monoid monoid.l-one* **by** *metis*
finally show [] ⊗$_{F2}$ *x* = *x* **.**
next
fix *g h x*
assume *g* ∈ *carrier F2 h* ∈ *carrier F2 x* ∈ *carrier F2*
thus (*g* ⊗$_{F2}$ *h*) ⊗$_{F2}$ *x* = *g* ⊗$_{F2}$ (*h* ⊗$_{F2}$ *x*)
by (*meson F2-is-group group.is-monoid monoid.m-assoc*)
qed

15 Behaviour of left multiplication by a

When we left-multiply a word that begins with a^{-1} by a, the leading pair cancels, and the result is exactly the tail of the original word.

lemma *a-mult-starts-with-aI*:
assumes *w* ∈ *S-aI*
shows *a-elt* ⊗$_{F2}$ *w* = *tl w*
proof −
from *assms* **have** *w-in*: *w* ∈ *carrier F2*
and *w-ne*: *w* ≠ []
and *w-hd*: *hd w* = (*True*, *A*)
by (*auto simp*: *starts-with-def*)
from *w-ne w-hd* **obtain** *u* **where** *w-eq*: *w* = (*True*, *A*) # *u*
by (*cases w*) *auto*
from *w-in* **have** *w-canc*: *canceled w* **by** *auto*
with *w-eq* **have** *u-canc*: *canceled u* **using** *cons-canceled* **by** *metis*

```
  have can: canceling (False, A) (True, A)
    by (simp add: canceling-def)
  have prep: a-elt @ w = [(False, A), (True, A)] @ u
    using w-eq by simp
  have step: cancels-to-1-at 0 ([(False, A), (True, A)] @ u) u
    using can by (auto simp: cancels-to-1-at-def cancel-at-def)
  hence cancels-to-1 (a-elt @ w) u using prep
    by (auto simp: cancels-to-1-def)
  hence cancels-to (a-elt @ w) u
    by (auto simp: cancels-to-def)
  hence norm: normalize (a-elt @ w) = u
    using u-canc by (rule normalize-discover[rotated])
  show ?thesis using norm w-eq
    by (simp add: F2-mult)
qed

lemma b-mult-starts-with-bI:
  assumes w ∈ S-bI
  shows b-elt ⊗_F2 w = tl w
proof −
  from assms have w-in: w ∈ carrier F2
    and w-ne: w ≠ []
    and w-hd: hd w = (True, B)
    by (auto simp: starts-with-def)
  from w-ne w-hd obtain u where w-eq: w = (True, B) # u
    by (cases w) auto
  from w-in have w-canc: canceled w by auto
  with w-eq have u-canc: canceled u using cons-canceled by metis

  have can: canceling (False, B) (True, B)
    by (simp add: canceling-def)
  have prep: b-elt @ w = [(False, B), (True, B)] @ u
    using w-eq by simp
  have step: cancels-to-1-at 0 ([(False, B), (True, B)] @ u) u
    using can by (auto simp: cancels-to-1-at-def cancel-at-def)
  hence cancels-to-1 (b-elt @ w) u using prep
    by (auto simp: cancels-to-1-def)
  hence cancels-to (b-elt @ w) u
    by (auto simp: cancels-to-def)
  hence norm: normalize (b-elt @ w) = u
    using u-canc by (rule normalize-discover[rotated])
  show ?thesis using norm w-eq
    by (simp add: F2-mult)
qed
```

16 Image computations

Translating ${S_a}^{-1}$ on the left by a yields exactly the carrier of F_2 minus S_a.

lemma *image-a-S-aI*:
shows $(\otimes_{F2})$ *a-elt* ‘ *S-aI* = *carrier F2* − *S-a*
proof
show $(\otimes_{F2})$ *a-elt* ‘ *S-aI* ⊆ *carrier F2* − *S-a*
proof
fix *v* **assume** *v* ∈ $(\otimes_{F2})$ *a-elt* ‘ *S-aI*
then obtain *w* **where** *w*: *w* ∈ *S-aI* **and** *v-eq*: *v* = *a-elt* $\otimes_{F2}$ *w* **by** *auto*
from *w* **have** *w-in*: *w* ∈ *carrier F2* **and** *w-ne*: *w* ≠ []
and *w-hd*: *hd w* = (*True*, *A*)
by (*auto simp*: *starts-with-def*)
from *w-ne w-hd* **obtain** *u* **where** *w-eq*: *w* = (*True*, *A*) # *u*
by (*cases w*) *auto*
from *a-mult-starts-with-aI*[*OF w*] **have** *v-tl*: *v* = *tl w* **using** *v-eq* **by** *simp*
hence *v-u*: *v* = *u* **using** *w-eq* **by** *simp*
from *w-in* **have** *canceled w* **by** *auto*
with *w-eq* **have** *u-canc*: *canceled u* **using** *cons-canceled* **by** *metis*
have *v-canc*: *canceled v* **using** *v-u u-canc* **by** *simp*
hence *v-in*: *v* ∈ *carrier F2* **by** (*rule canceled-in-F2*)
have *not-starts-a*: *v* ∉ *S-a*
proof (*cases u*)
case *Nil* **with** *v-u* **show** *?thesis* **by** (*auto simp*: *starts-with-def*)
next
case (*Cons p ps*)
have *w-split*: *w* = [(*True*, *A*), *p*] @ *ps* **using** *w-eq Cons* **by** *simp*
have *ncan*: ¬ *canceling* (*True*, *A*) *p*
proof
assume *can-pair*: *canceling* (*True*, *A*) *p*
have *cancels-to-1-at 0 w ps*
using *w-split can-pair*
by (*auto simp*: *cancels-to-1-at-def cancel-at-def*)
hence *cancels-to-1 w ps*
by (*auto simp*: *cancels-to-1-def*)
hence ¬ *canceled w*
by (*auto simp*: *canceled-def*)
with ‹*canceled w*› **show** *False* **by** *contradiction*
qed
hence *p* ≠ (*False*, *A*)
by (*auto simp*: *canceling-def*)
hence *hd-neq*: *hd v* ≠ (*False*, *A*)
using *Cons v-u* **by** *simp*
thus *?thesis* **using** *v-in* **by** (*auto simp*: *starts-with-def*)
qed
show *v* ∈ *carrier F2* − *S-a* **using** *v-in not-starts-a* **by** *simp*
qed
next
show *carrier F2* − *S-a* ⊆ $(\otimes_{F2})$ *a-elt* ‘ *S-aI*
proof
fix *v* **assume** *v*: *v* ∈ *carrier F2* − *S-a*
hence *v-in*: *v* ∈ *carrier F2* **and** *v-not-a*: *v* ∉ *S-a* **by** *auto*

```
    let ?w = (True, A) # v
    have v-canc: canceled v using v-in by auto
    have ncan: v = [] ∨ ¬ canceling (True, A) (hd v)
    proof (cases v)
      case Nil thus ?thesis by simp
    next
      case (Cons p ps)
      from v-not-a Cons v-in have p ≠ (False, A)
        by (auto simp: starts-with-def)
      hence ¬ canceling (True, A) p
        by (cases p) (auto simp: canceling-def)
      with Cons show ?thesis by simp
    qed
    have w-canc: canceled ?w
    proof (rule ccontr)
      assume ¬ canceled ?w
      then obtain w' where cancels-to-1 ?w w'
        by (auto simp: canceled-def)
      then obtain xs1 x1 x2 xs2
        where decomp: ?w = xs1 @ x1 # x2 # xs2
        and can12: canceling x1 x2
        by (rule cancels-to-1-unfold)
      show False
      proof (cases xs1)
        case Nil
        with decomp have x1-eq: x1 = (True, A) by simp
        from decomp Nil have v-eq: v = x2 # xs2 by simp
        from v-eq x1-eq can12 ncan show False by simp
      next
        case (Cons y ys)
        with decomp have v-decomp: v = ys @ x1 # x2 # xs2 by simp
        with can12 have ¬ canceled v
          unfolding canceled-def
          by (auto intro: cancels-to-1-fold)
        with v-canc show False by contradiction
      qed
    qed
    hence w-in: ?w ∈ carrier F2 by auto
    have w-starts: ?w ∈ S-aI using w-in by (auto simp: starts-with-def)
    have v-eq: v = a-elt ⊗_F2 ?w
      using a-mult-starts-with-aI[OF w-starts] by simp
    show v ∈ (⊗_F2) a-elt ‘ S-aI using v-eq w-starts by auto
  qed
qed

lemma image-b-S-bI:
  shows (⊗_F2) b-elt ‘ S-bI = carrier F2 − S-b
proof
  show (⊗_F2) b-elt ‘ S-bI ⊆ carrier F2 − S-b
```

```
  proof
    fix v assume v ∈ (⊗_F2) b-elt ‘ S-bI
    then obtain w where w: w ∈ S-bI and v-eq: v = b-elt ⊗_F2 w by auto
    from w have w-in: w ∈ carrier F2 and w-ne: w ≠ []
      and w-hd: hd w = (True, B)
      by (auto simp: starts-with-def)
    from w-ne w-hd obtain u where w-eq: w = (True, B) # u
      by (cases w) auto
    from b-mult-starts-with-bI[OF w] have v-tl: v = tl w using v-eq by simp
    hence v-u: v = u using w-eq by simp
    from w-in have canceled w by auto
    with w-eq have u-canc: canceled u using cons-canceled by metis
    have v-canc: canceled v using v-u u-canc by simp
    hence v-in: v ∈ carrier F2 by (rule canceled-in-F2)
    have not-starts-b: v ∉ S-b
    proof (cases u)
      case Nil with v-u show ?thesis by (auto simp: starts-with-def)
    next
      case (Cons p ps)
      have w-split: w = [(True, B), p] @ ps using w-eq Cons by simp
      have ncan: ¬ canceling (True, B) p
      proof
        assume can-pair: canceling (True, B) p
        have cancels-to-1-at 0 w ps
          using w-split can-pair
          by (auto simp: cancels-to-1-at-def cancel-at-def)
        hence cancels-to-1 w ps
          by (auto simp: cancels-to-1-def)
        hence ¬ canceled w
          by (auto simp: canceled-def)
        with ‹canceled w› show False by contradiction
      qed
      hence p ≠ (False, B)
        by (auto simp: canceling-def)
      hence hd-neq: hd v ≠ (False, B)
        using Cons v-u by simp
      thus ?thesis using v-in by (auto simp: starts-with-def)
    qed
    show v ∈ carrier F2 − S-b using v-in not-starts-b by simp
  qed
next
  show carrier F2 − S-b ⊆ (⊗_F2) b-elt ‘ S-bI
  proof
    fix v assume v: v ∈ carrier F2 − S-b
    hence v-in: v ∈ carrier F2 and v-not-b: v ∉ S-b by auto
    let ?w = (True, B) # v
    have v-canc: canceled v using v-in by auto
    have ncan: v = [] ∨ ¬ canceling (True, B) (hd v)
    proof (cases v)
```

```
    case Nil thus ?thesis by simp
  next
    case (Cons p ps)
    from v-not-b Cons v-in have p ≠ (False, B)
      by (auto simp: starts-with-def)
    hence ¬ canceling (True, B) p
      by (cases p) (auto simp: canceling-def)
    with Cons show ?thesis by simp
  qed
  have w-canc: canceled ?w
  proof (rule ccontr)
    assume ¬ canceled ?w
    then obtain w' where cancels-to-1 ?w w'
      by (auto simp: canceled-def)
    then obtain xs1 x1 x2 xs2
      where decomp: ?w = xs1 @ x1 # x2 # xs2
      and can12: canceling x1 x2
      by (rule cancels-to-1-unfold)
    show False
    proof (cases xs1)
      case Nil
      with decomp have x1-eq: x1 = (True, B) by simp
      from decomp Nil have v-eq: v = x2 # xs2 by simp
      from v-eq x1-eq can12 ncan show False by simp
    next
      case (Cons y ys)
      with decomp have v-decomp: v = ys @ x1 # x2 # xs2 by simp
      with can12 have ¬ canceled v
        unfolding canceled-def
        by (auto intro: cancels-to-1-fold)
      with v-canc show False by contradiction
    qed
  qed
  hence w-in: ?w ∈ carrier F2 by auto
  have w-starts: ?w ∈ S-bI using w-in by (auto simp: starts-with-def)
  have v-eq: v = b-elt ⊗_F2 ?w
    using b-mult-starts-with-bI[OF w-starts] by simp
  show v ∈ (⊗_F2) b-elt ‘ S-bI using v-eq w-starts by auto
 qed
qed
```

17 The paradoxical decomposition of F_2

Putting everything together: S_a together with the a-translate of ${S_a}^{-1}$ covers F_2 (disjointly), and symmetrically for b. Hence F_2 is paradoxical.

```
lemma S-a-un-image-a-S-aI: S-a ∪ (⊗_F2) a-elt ‘ S-aI = carrier F2
  using image-a-S-aI starts-with-subset by blast
```

lemma *S-b-un-image-b-S-bI*: *S-b* $\cup$ $(\otimes_{F2})$ *b-elt* ‘ *S-bI* = *carrier F2*
using *image-b-S-bI starts-with-subset* **by** *blast*

lemma *S-a-disj-image-a-S-aI*: *S-a* $\cap$ $(\otimes_{F2})$ *a-elt* ‘ *S-aI* = {}
using *image-a-S-aI* **by** *blast*

lemma *S-b-disj-image-b-S-bI*: *S-b* $\cap$ $(\otimes_{F2})$ *b-elt* ‘ *S-bI* = {}
using *image-b-S-bI* **by** *blast*

definition *aI-ray* :: (*bool* × *gen2*) *list set* **where**
aI-ray = {*replicate n* (*True*, *A*) | *n*. *True*}

lemma *canceled-replicate-aI*: *canceled* (*replicate n* (*True*, *A*))
unfolding *canceled-def*
proof
assume *Domainp cancels-to-1* (*replicate n* (*True*, *A*))
then obtain *w* **where** *cancels-to-1* (*replicate n* (*True*, *A*)) *w*
by *auto*
then obtain *i* **where** *i*: *Suc i* < *length* (*replicate n* (*True*, *A*))
and *canceling* (*replicate n* (*True*, *A*) ! *i*) (*replicate n* (*True*, *A*) ! *Suc i*)
by (*auto simp*: *cancels-to-1-def cancels-to-1-at-def*)
hence *canceling* (*True*, *A*) (*True*, *A*)
by *simp*
thus *False*
by (*simp add*: *canceling-def*)
qed

lemma *aI-ray-subset-carrier*: *aI-ray* ⊆ *carrier F2*
unfolding *aI-ray-def*
by (*auto intro*: *canceled-in-F2 canceled-replicate-aI*)

lemma *aI-ray-cases*:
assumes *w* ∈ *aI-ray*
shows *w* = [] ∨ *w* ∈ *S-aI*
proof −
from *assms* **obtain** *n* **where** *w*: *w* = *replicate n* (*True*, *A*)
unfolding *aI-ray-def* **by** *blast*
show *?thesis*
proof (*cases n*)
case *0*
with *w* **show** *?thesis* **by** *simp*
next
case (*Suc m*)
hence *w* ≠ [] *hd w* = (*True*, *A*) *w* ∈ *carrier F2*
using *w aI-ray-subset-carrier assms* **by** *auto*
thus *?thesis*
by (*auto simp*: *starts-with-def*)
qed
qed

lemma *a-mult-aI-ray-pos*:
$(\otimes_{F2})$ *a-elt* ‘ (*aI-ray* − {[]}) = *aI-ray*
proof
show $(\otimes_{F2})$ *a-elt* ‘ (*aI-ray* − {[]}) ⊆ *aI-ray*
proof
fix *y*
assume *y* ∈ $(\otimes_{F2})$ *a-elt* ‘ (*aI-ray* − {[]})
then obtain *w* **where** *w-ray*: *w* ∈ *aI-ray* **and** *w-ne*: *w* ≠ []
and *y*: *y* = *a-elt* $\otimes_{F2}$ *w*
by *auto*
from *w-ray* **obtain** *n* **where** *w*: *w* = *replicate n* (*True*, *A*)
unfolding *aI-ray-def* **by** *blast*
from *w-ne w* **obtain** *m* **where** *n*: *n* = *Suc m*
by (*cases n*) *auto*
have *w-S*: *w* ∈ *S-aI*
using *aI-ray-cases*[*OF w-ray*] *w-ne* **by** *blast*
have *y* = *tl w*
using *a-mult-starts-with-aI*[*OF w-S*] *y* **by** *simp*
also have ... = *replicate m* (*True*, *A*)
using *w n* **by** *simp*
finally show *y* ∈ *aI-ray*
unfolding *aI-ray-def* **by** *blast*
qed
show *aI-ray* ⊆ $(\otimes_{F2})$ *a-elt* ‘ (*aI-ray* − {[]})
proof
fix *y*
assume *y* ∈ *aI-ray*
then obtain *n* **where** *y*: *y* = *replicate n* (*True*, *A*)
unfolding *aI-ray-def* **by** *blast*
let *?w* = *replicate* (*Suc n*) (*True*, *A*)
have *w-ray*: *?w* ∈ *aI-ray*
unfolding *aI-ray-def* **by** *blast*
have *w-ne*: *?w* ≠ []
by *simp*
have *w-S*: *?w* ∈ *S-aI*
using *aI-ray-cases*[*OF w-ray*] *w-ne* **by** *blast*
have *a-elt* $\otimes_{F2}$ *?w* = *tl ?w*
by (*rule a-mult-starts-with-aI*[*OF w-S*])
also have ... = *y*
using *y* **by** *simp*
finally show *y* ∈ $(\otimes_{F2})$ *a-elt* ‘ (*aI-ray* − {[]})
using *w-ray w-ne* **by** *blast*
qed
qed

lemma *a-mult-nonray-stays-nonray*:
assumes *w* ∈ *S-aI* **and** *w* ∉ *aI-ray*
shows *a-elt* $\otimes_{F2}$ *w* ∉ *aI-ray*

proof
  assume *ray*: *a-elt* $\otimes_{F2}$ *w* $\in$ *aI-ray*
  from *assms*(*1*) **have** *w-ne*: *w* $\neq$ [] **and** *w-hd*: *hd w* = (*True*, *A*)
    and *w-carrier*: *w* $\in$ *carrier F2*
    by (*auto simp*: *starts-with-def*)
  from *w-ne w-hd* **obtain** *u* **where** *w*: *w* = (*True*, *A*) # *u*
    by (*cases w*) *auto*
  have *tl w* $\in$ *aI-ray*
    using *ray a-mult-starts-with-aI*[*OF assms*(*1*)] **by** *simp*
  then obtain *n* **where** *u*: *u* = *replicate n* (*True*, *A*)
    unfolding *aI-ray-def* **using** *w* **by** *auto*
  hence *w* = *replicate* (*Suc n*) (*True*, *A*)
    using *w* **by** *simp*
  hence *w* $\in$ *aI-ray*
    unfolding *aI-ray-def* **by** *blast*
  with *assms*(*2*) **show** *False*
    by *contradiction*
qed

lemma *image-a-S-aI-nonray*:
  ($\otimes_{F2}$) *a-elt* ‘ (*S-aI* − *aI-ray*) = *carrier F2* − *S-a* − *aI-ray*
proof
  show ($\otimes_{F2}$) *a-elt* ‘ (*S-aI* − *aI-ray*) $\subseteq$ *carrier F2* − *S-a* − *aI-ray*
  proof
    fix *y*
    assume *y* $\in$ ($\otimes_{F2}$) *a-elt* ‘ (*S-aI* − *aI-ray*)
    then obtain *w* **where** *w*: *w* $\in$ *S-aI w* $\notin$ *aI-ray*
      and *y*: *y* = *a-elt* $\otimes_{F2}$ *w*
      by *auto*
    have *y* $\in$ *carrier F2* − *S-a*
      using *image-a-S-aI w*(*1*) *y* **by** *blast*
    moreover have *y* $\notin$ *aI-ray*
      using *a-mult-nonray-stays-nonray*[*OF w*] *y* **by** *simp*
    ultimately show *y* $\in$ *carrier F2* − *S-a* − *aI-ray*
      by *simp*
  qed
  show *carrier F2* − *S-a* − *aI-ray* $\subseteq$ ($\otimes_{F2}$) *a-elt* ‘ (*S-aI* − *aI-ray*)
  proof
    fix *y*
    assume *y*: *y* $\in$ *carrier F2* − *S-a* − *aI-ray*
    hence *y* $\in$ *carrier F2* − *S-a*
      by *simp*
    then obtain *w* **where** *w*: *w* $\in$ *S-aI* **and** *y-eq*: *y* = *a-elt* $\otimes_{F2}$ *w*
      using *image-a-S-aI* **by** *blast*
    have *w* $\notin$ *aI-ray*
    proof
      assume *w-ray*: *w* $\in$ *aI-ray*
      with *w* **have** *w* $\in$ *aI-ray* − {[]}
        by (*auto simp*: *starts-with-def*)

```
      hence y ∈ aI-ray
        using y-eq a-mult-aI-ray-pos by blast
      with y show False
        by simp
    qed
    thus y ∈ (⊗_F2) a-elt ‘ (S-aI − aI-ray)
      using w y-eq by blast
  qed
qed

theorem F2-paradoxical-partition:
  ∃ P Q :: ((bool × gen2) list) set list. ∃ gP gQ :: ((bool × gen2) list) list.
      length P = length gP ∧ length Q = length gQ ∧
      set gP ⊆ carrier F2 ∧ set gQ ⊆ carrier F2 ∧
      pairwise-disjoint (P @ Q) ∧
      pairwise-disjoint (map2 F2-act.image-set gP P) ∧
      pairwise-disjoint (map2 F2-act.image-set gQ Q) ∧
      carrier F2 = (⋃ i<length P. P ! i) ∪ (⋃ i<length Q. Q ! i) ∧
      carrier F2 = (⋃ i<length P. F2-act.image-set (gP ! i) (P ! i)) ∧
      carrier F2 = (⋃ i<length Q. F2-act.image-set (gQ ! i) (Q ! i))
proof −
  let ?A1 = S-a ∪ aI-ray
  let ?A2 = S-aI − aI-ray
  let ?B1 = S-b
  let ?B2 = S-bI
  let ?P = [?A1, ?A2]
  let ?Q = [?B1, ?B2]
  let ?gP = [[]::(bool × gen2) list, a-elt]
  let ?gQ = [[]::(bool × gen2) list, b-elt]

  have ray-disj-Sa: aI-ray ∩ S-a = {}
    using aI-ray-cases starts-with-pairwise-disjoint by blast
  have ray-disj-Sb: aI-ray ∩ S-b = {}
    using aI-ray-cases starts-with-pairwise-disjoint by blast
  have ray-disj-SbI: aI-ray ∩ S-bI = {}
    using aI-ray-cases starts-with-pairwise-disjoint by blast

  have source-cover: carrier F2 =
      (⋃ i<length ?P. ?P ! i) ∪ (⋃ i<length ?Q. ?Q ! i)
  proof
    show carrier F2 ⊆ (⋃ i<length ?P. ?P ! i) ∪ (⋃ i<length ?Q. ?Q ! i)
    proof
      fix w
      assume w: w ∈ carrier F2
      show w ∈ (⋃ i<length ?P. ?P ! i) ∪ (⋃ i<length ?Q. ?Q ! i)
      proof (cases w ∈ S-a ∨ w ∈ S-aI ∨ w ∈ S-b ∨ w ∈ S-bI)
        case True
        then consider w ∈ S-a | w ∈ S-aI | w ∈ S-b | w ∈ S-bI
          by blast
```

```
      thus ?thesis
      proof cases
        case 1
        then show ?thesis by (simp add: lessThan-Suc)
      next
        case 2
        then show ?thesis
        proof (cases w ∈ aI-ray)
          case True
          then show ?thesis by (simp add: lessThan-Suc)
        next
          case False
          with 2 show ?thesis by (simp add: lessThan-Suc)
        qed
      next
        case 3
        then show ?thesis by (simp add: lessThan-Suc)
      next
        case 4
        then show ?thesis by (simp add: lessThan-Suc)
      qed
    next
      case False
      with F2-decomposition w have w = []
        by blast
      hence w ∈ aI-ray
        unfolding aI-ray-def by (auto intro!: exI[of - 0])
      thus ?thesis
        by (simp add: lessThan-Suc)
    qed
  qed
  show (⋃ i<length ?P. ?P ! i) ∪ (⋃ i<length ?Q. ?Q ! i) ⊆ carrier F2
    using starts-with-subset[of False A] starts-with-subset[of True A]
      starts-with-subset[of False B] starts-with-subset[of True B]
      aI-ray-subset-carrier
    by (auto simp: lessThan-Suc)
qed

have source-disj: pairwise-disjoint (?P @ ?Q)
  using starts-with-pairwise-disjoint ray-disj-Sa ray-disj-Sb ray-disj-SbI
  by (auto simp: pairwise-disjoint-def nth-Cons' nth-append Int-commute)

have im-P0: F2-act.image-set ([]::(bool × gen2) list) ?A1 = ?A1
  using starts-with-subset aI-ray-subset-carrier
  by (intro F2-act.image-set-unit) auto
have im-P1: F2-act.image-set a-elt ?A2 = carrier F2 − S-a − aI-ray
  by (simp add: F2-act.image-set-def image-a-S-aI-nonray)
have im-Q0: F2-act.image-set ([]::(bool × gen2) list) ?B1 = ?B1
  using starts-with-subset
```

by (*rule F2-act.image-set-unit*)
have *im-Q1*: *F2-act.image-set b-elt ?B2* = $(\otimes_{F2})$ *b-elt ‘ S-bI*
by (*simp add*: *F2-act.image-set-def*)

have *map2P*: *map2 F2-act.image-set ?gP ?P* = [*?A1*, *carrier F2* − *S-a* − *aI-ray*]
using *im-P0 im-P1* **by** *simp*
have *map2Q*: *map2 F2-act.image-set ?gQ ?Q* = [*S-b*, $(\otimes_{F2})$ *b-elt ‘ S-bI*]
using *im-Q0 im-Q1* **by** *simp*

have *imageP-disj*: *pairwise-disjoint* (*map2 F2-act.image-set ?gP ?P*)
using *map2P* **by** (*auto simp*: *pairwise-disjoint-def nth-Cons′ Int-commute*)
have *imageQ-disj*: *pairwise-disjoint* (*map2 F2-act.image-set ?gQ ?Q*)
using *S-b-disj-image-b-S-bI map2Q*
by (*auto simp*: *pairwise-disjoint-def nth-Cons′ Int-commute*)

have *coverP*: *carrier F2* =
($\bigcup$*i*<*length ?P*. *F2-act.image-set* (*?gP* ! *i*) (*?P* ! *i*))
proof −
have ($\bigcup$*i*<*length ?P*. *F2-act.image-set* (*?gP* ! *i*) (*?P* ! *i*)) =
?A1 ∪ (*carrier F2* − *S-a* − *aI-ray*)
using *im-P0 im-P1* **by** (*auto simp*: *lessThan-Suc nth-Cons′*)
also have ... = *carrier F2*
using *starts-with-subset aI-ray-subset-carrier* **by** *auto*
finally show *?thesis*
by *simp*
qed

have *coverQ*: *carrier F2* =
($\bigcup$*i*<*length ?Q*. *F2-act.image-set* (*?gQ* ! *i*) (*?Q* ! *i*))
proof −
have ($\bigcup$*i*<*length ?Q*. *F2-act.image-set* (*?gQ* ! *i*) (*?Q* ! *i*)) =
S-b ∪ $(\otimes_{F2})$ *b-elt ‘ S-bI*
using *im-Q0 im-Q1* **by** (*auto simp*: *lessThan-Suc nth-Cons′*)
also have ... = *carrier F2*
by (*rule S-b-un-image-b-S-bI*)
finally show *?thesis*
by *simp*
qed

show *?thesis*
proof (*intro exI conjI*)
show *length ?P* = *length ?gP*
by *simp*
show *length ?Q* = *length ?gQ*
by *simp*
show *set ?gP* ⊆ *carrier F2*
by *simp*
show *set ?gQ* ⊆ *carrier F2*
by *simp*

```
    show pairwise-disjoint (?P @ ?Q)
      by (rule source-disj)
    show pairwise-disjoint (map2 F2-act.image-set ?gP ?P)
      by (rule imageP-disj)
    show pairwise-disjoint (map2 F2-act.image-set ?gQ ?Q)
      by (rule imageQ-disj)
    show carrier F2 = (⋃ i<length ?P. ?P ! i) ∪ (⋃ i<length ?Q. ?Q ! i)
      by (rule source-cover)
    show carrier F2 = (⋃ i<length ?P. F2-act.image-set (?gP ! i) (?P ! i))
      by (rule coverP)
    show carrier F2 = (⋃ i<length ?Q. F2-act.image-set (?gQ ! i) (?Q ! i))
      by (rule coverQ)
  qed
qed
```

We unfold *F2-act.paradoxical* directly to avoid locale-instantiation complications with implicit polymorphic variables.

```
theorem F2-paradoxical:
  F2-act.paradoxical (carrier F2)
proof −
  let ?P = [S-a, S-aI] and ?Q = [S-b, S-bI]
  let ?gP = [[]::(bool × gen2) list, a-elt]
  let ?gQ = [[]::(bool × gen2) list, b-elt]

  have lensP: length ?P = length ?gP by simp
  have lensQ: length ?Q = length ?gQ by simp

  have closedP: set ?gP ⊆ carrier F2 by simp
  have closedQ: set ?gQ ⊆ carrier F2 by simp

  have disj-P: S-a ∩ S-aI = {}
    by (simp add: starts-with-pairwise-disjoint)
  have disj-Q: S-b ∩ S-bI = {}
    by (simp add: starts-with-pairwise-disjoint)
  have disj-PQ: (S-a ∪ S-aI) ∩ (S-b ∪ S-bI) = {}
    using starts-with-pairwise-disjoint by blast

  have disj-all: pairwise-disjoint (?P @ ?Q)
    using disj-P disj-Q disj-PQ
    by (auto simp: pairwise-disjoint-def nth-Cons' nth-append Int-commute)

  have im-P0: F2-act.image-set ([]::(bool × gen2) list) S-a = S-a
    using starts-with-subset
    by (rule F2-act.image-set-unit)
  have im-Q0: F2-act.image-set ([]::(bool × gen2) list) S-b = S-b
    using starts-with-subset
    by (rule F2-act.image-set-unit)
  have im-P1: F2-act.image-set a-elt S-aI = (⊗F2) a-elt ‘ S-aI
    by (simp add: F2-act.image-set-def)
```

```
  have im-Q1: F2-act.image-set b-elt S-bI = (⊗F2) b-elt ‘ S-bI
    by (simp add: F2-act.image-set-def)

  have map2P: map2 F2-act.image-set ?gP ?P = [S-a, (⊗F2) a-elt ‘ S-aI]
    using im-P0 im-P1 by simp
  have map2Q: map2 F2-act.image-set ?gQ ?Q = [S-b, (⊗F2) b-elt ‘ S-bI]
    using im-Q0 im-Q1 by simp

  have disj-im-P: pairwise-disjoint (map2 F2-act.image-set ?gP ?P)
    using S-a-disj-image-a-S-aI map2P
    by (auto simp: pairwise-disjoint-def nth-Cons′ Int-commute)
  have disj-im-Q: pairwise-disjoint (map2 F2-act.image-set ?gQ ?Q)
    using S-b-disj-image-b-S-bI map2Q
    by (auto simp: pairwise-disjoint-def nth-Cons′ Int-commute)

  have un-sub: (⋃i<length ?P. ?P ! i) ∪ (⋃i<length ?Q. ?Q ! i) ⊆ carrier F2
    using starts-with-subset
    by (auto simp: lessThan-Suc nth-Cons′)

  have cov-P: carrier F2 = (⋃i<length ?P. F2-act.image-set (?gP ! i) (?P ! i))
  proof −
    have (⋃i<length ?P. F2-act.image-set (?gP ! i) (?P ! i))
       = F2-act.image-set [] S-a ∪ F2-act.image-set a-elt S-aI
      by (auto simp: lessThan-Suc nth-Cons′)
    also have ... = S-a ∪ (⊗F2) a-elt ‘ S-aI
      using im-P0 im-P1 by simp
    also have ... = carrier F2 by (rule S-a-un-image-a-S-aI)
    finally show ?thesis by simp
  qed

  have cov-Q: carrier F2 = (⋃i<length ?Q. F2-act.image-set (?gQ ! i) (?Q ! i))
  proof −
    have (⋃i<length ?Q. F2-act.image-set (?gQ ! i) (?Q ! i))
       = F2-act.image-set [] S-b ∪ F2-act.image-set b-elt S-bI
      by (auto simp: lessThan-Suc nth-Cons′)
    also have ... = S-b ∪ (⊗F2) b-elt ‘ S-bI
      using im-Q0 im-Q1 by simp
    also have ... = carrier F2 by (rule S-b-un-image-b-S-bI)
    finally show ?thesis by simp
  qed

  show ?thesis
    unfolding F2-act.paradoxical-def
    using lensP lensQ closedP closedQ disj-all disj-im-P disj-im-Q
          un-sub cov-P cov-Q
    by blast
qed

end
```

theory *Free-Rotations-SO3*
imports
Banach-Tarski.F2-Paradox
Banach-Tarski.Free-Action-Paradox
HOL−Computational-Algebra.Primes
Free−Groups.PingPongLemma
begin

18 Rotations of three-space

A rotation is an orthogonal transformation of determinant 1.

definition *rotation* :: (*real^3* ⇒ *real^3*) ⇒ *bool* **where**
rotation f ⟷ *orthogonal-transformation f* ∧ *det* (*matrix f*) = *1*

definition *SO3* :: (*real^3* ⇒ *real^3*) *set* **where**
SO3 = {*f*. *rotation f*}

SO(3) contains the identity and is closed under composition.

lemma *id-in-SO3*: *id* ∈ *SO3*
proof −
have *orthogonal-transformation* (*id*::*real^3* ⇒ *real^3*)
using *orthogonal-transformation-id* **by** (*simp add*: *id-def*)
moreover have *det* (*matrix* (*id*::*real^3* ⇒ *real^3*)) = *1*
by *simp*
ultimately show *?thesis*
unfolding *SO3-def rotation-def* **by** *simp*
qed

lemma *SO3-closed-compose*:
assumes *f* ∈ *SO3* **and** *g* ∈ *SO3*
shows *f* ∘ *g* ∈ *SO3*
proof −
from *assms* **have** *f-orth*: *orthogonal-transformation f*
and *g-orth*: *orthogonal-transformation g*
and *f-det*: *det* (*matrix f*) = *1*
and *g-det*: *det* (*matrix g*) = *1*
unfolding *SO3-def rotation-def* **by** *auto*
have *f-lin*: *linear f* **using** *f-orth* **by** (*rule orthogonal-transformation-linear*)
have *g-lin*: *linear g* **using** *g-orth* **by** (*rule orthogonal-transformation-linear*)

have *orth-comp*: *orthogonal-transformation* (*f* ∘ *g*)
using *f-orth g-orth* **by** (*rule orthogonal-transformation-compose*)

have *matrix* (*f* ∘ *g*) = *matrix f* ** *matrix g*
using *g-lin f-lin* **by** (*rule matrix-compose*)
hence *det* (*matrix* (*f* ∘ *g*)) = *det* (*matrix f*) ∗ *det* (*matrix g*)

by (*simp add*: *det-mul*)
also have ... = *1* **using** *f-det g-det* **by** *simp*
finally have *det* (*matrix* (*f* ∘ *g*)) = *1* **.**

with *orth-comp* **show** *?thesis*
unfolding *SO3-def rotation-def* **by** *simp*
qed

SO(3) elements preserve the unit sphere.

lemma *rotation-preserves-sphere2*:
assumes *f* ∈ *SO3* **and** *x* ∈ *sphere2*
shows *f x* ∈ *sphere2*
proof −
from *assms*(*1*) **have** *orthogonal-transformation f*
unfolding *SO3-def rotation-def* **by** *auto*
hence *norm* (*f x*) = *norm x*
by (*rule orthogonal-transformation-norm*)
with *assms*(*2*) **show** *?thesis*
by (*simp add*: *sphere2-def*)
qed

lemma *SO3-bij*:
assumes *f* ∈ *SO3*
shows *bij f*
using *assms orthogonal-transformation-bij*
unfolding *SO3-def rotation-def* **by** *auto*

19 Two distinguished rotations

We use the rotations R_x (about the x-axis) and R_z (about the z-axis) by the same angle ϑ with *cos* ϑ = *1*/*3*. The exact-form witnesses are unimportant for the paradoxical decomposition; what matters is the abstract conclusion that these generate a free F_2 inside SO(3).

definition *rot-c* :: *real* **where**
rot-c = *1* / *3*

definition *rot-s* :: *real* **where**
rot-s = *sqrt 8* / *3*

lemma *rot-c-sq-plus-rot-s-sq*: *rot-c*^{2} + *rot-s*^{2} = *1*
unfolding *rot-c-def rot-s-def*
by (*simp add*: *power-divide*)

lemma *rot-c-mul-plus-rot-s-mul* [*simp*]: *rot-c* ∗ *rot-c* + *rot-s* ∗ *rot-s* = *1*
using *rot-c-sq-plus-rot-s-sq* **by** (*simp add*: *power2-eq-square*)

definition *Rx-mat* :: *real^3^3* **where**

```
  Rx-mat = vector [
    vector [1, 0, 0],
    vector [0, rot-c, − rot-s],
    vector [0, rot-s, rot-c]]

definition Rz-mat :: real^3^3 where
  Rz-mat = vector [
    vector [rot-c, − rot-s, 0],
    vector [rot-s, rot-c, 0],
    vector [0, 0, 1]]

definition Rx-inv-mat :: real^3^3 where
  Rx-inv-mat = vector [
    vector [1, 0, 0],
    vector [0, rot-c, rot-s],
    vector [0, − rot-s, rot-c]]

definition Rz-inv-mat :: real^3^3 where
  Rz-inv-mat = vector [
    vector [rot-c, rot-s, 0],
    vector [− rot-s, rot-c, 0],
    vector [0, 0, 1]]

definition Rx :: real^3 ⇒ real^3 where
  Rx v = Rx-mat *v v

definition Rz :: real^3 ⇒ real^3 where
  Rz v = Rz-mat *v v

definition Rx-inv :: real^3 ⇒ real^3 where
  Rx-inv v = Rx-inv-mat *v v

definition Rz-inv :: real^3 ⇒ real^3 where
  Rz-inv v = Rz-inv-mat *v v

lemma Rx-mat-orthogonal: orthogonal-matrix Rx-mat
  unfolding orthogonal-matrix Rx-mat-def
  by (simp add: matrix-matrix-mult-def transpose-def mat-def vec-eq-iff vector-def
sum-3 forall-3
      rot-c-sq-plus-rot-s-sq power2-eq-square algebra-simps)

lemma Rz-mat-orthogonal: orthogonal-matrix Rz-mat
  unfolding orthogonal-matrix Rz-mat-def
  by (simp add: matrix-matrix-mult-def transpose-def mat-def vec-eq-iff vector-def
sum-3 forall-3
      rot-c-sq-plus-rot-s-sq power2-eq-square algebra-simps)

lemma Rx-inv-mat-orthogonal: orthogonal-matrix Rx-inv-mat
  unfolding orthogonal-matrix Rx-inv-mat-def
```

by (*simp add*: *matrix-matrix-mult-def transpose-def mat-def vec-eq-iff vector-def sum-3 forall-3*
rot-c-sq-plus-rot-s-sq power2-eq-square algebra-simps)

lemma *Rz-inv-mat-orthogonal*: *orthogonal-matrix Rz-inv-mat*
unfolding *orthogonal-matrix Rz-inv-mat-def*
by (*simp add*: *matrix-matrix-mult-def transpose-def mat-def vec-eq-iff vector-def sum-3 forall-3*
rot-c-sq-plus-rot-s-sq power2-eq-square algebra-simps)

lemma *det-Rx-mat* [*simp*]: *det Rx-mat = 1*
unfolding *Rx-mat-def*
by (*simp add*: *det-3 vector-def rot-c-sq-plus-rot-s-sq power2-eq-square algebra-simps*)

lemma *det-Rz-mat* [*simp*]: *det Rz-mat = 1*
unfolding *Rz-mat-def*
by (*simp add*: *det-3 vector-def rot-c-sq-plus-rot-s-sq power2-eq-square algebra-simps*)

lemma *det-Rx-inv-mat* [*simp*]: *det Rx-inv-mat = 1*
unfolding *Rx-inv-mat-def*
by (*simp add*: *det-3 vector-def rot-c-sq-plus-rot-s-sq power2-eq-square algebra-simps*)

lemma *det-Rz-inv-mat* [*simp*]: *det Rz-inv-mat = 1*
unfolding *Rz-inv-mat-def*
by (*simp add*: *det-3 vector-def rot-c-sq-plus-rot-s-sq power2-eq-square algebra-simps*)

lemma *Rx-mat-inverse-left*: *Rx-inv-mat ** Rx-mat = mat 1*
unfolding *Rx-inv-mat-def Rx-mat-def*
by (*simp add*: *matrix-matrix-mult-def mat-def vec-eq-iff vector-def sum-3 forall-3*
rot-c-sq-plus-rot-s-sq power2-eq-square algebra-simps)

lemma *Rx-mat-inverse-right*: *Rx-mat ** Rx-inv-mat = mat 1*
unfolding *Rx-inv-mat-def Rx-mat-def*
by (*simp add*: *matrix-matrix-mult-def mat-def vec-eq-iff vector-def sum-3 forall-3*
rot-c-sq-plus-rot-s-sq power2-eq-square algebra-simps)

lemma *Rz-mat-inverse-left*: *Rz-inv-mat ** Rz-mat = mat 1*
unfolding *Rz-inv-mat-def Rz-mat-def*
by (*simp add*: *matrix-matrix-mult-def mat-def vec-eq-iff vector-def sum-3 forall-3*
rot-c-sq-plus-rot-s-sq power2-eq-square algebra-simps)

lemma *Rz-mat-inverse-right*: *Rz-mat ** Rz-inv-mat = mat 1*
unfolding *Rz-inv-mat-def Rz-mat-def*
by (*simp add*: *matrix-matrix-mult-def mat-def vec-eq-iff vector-def sum-3 forall-3*
rot-c-sq-plus-rot-s-sq power2-eq-square algebra-simps)

The concrete generators and their inverses lie in $SO(3)$.

lemma *Rx-in-SO3*: *Rx ∈ SO3*
unfolding *SO3-def rotation-def Rx-def*

by (*simp add*: *orthogonal-transformation-matrix Rx-mat-orthogonal matrix-vector-mul-linear*)

lemma *Rz-in-SO3*: *Rz* ∈ *SO3*
unfolding *SO3-def rotation-def Rz-def*
by (*simp add*: *orthogonal-transformation-matrix Rz-mat-orthogonal matrix-vector-mul-linear*)

lemma *Rx-inv-in-SO3*: *Rx-inv* ∈ *SO3*
unfolding *SO3-def rotation-def Rx-inv-def*
by (*simp add*: *orthogonal-transformation-matrix Rx-inv-mat-orthogonal matrix-vector-mul-linear*)

lemma *Rz-inv-in-SO3*: *Rz-inv* ∈ *SO3*
unfolding *SO3-def rotation-def Rz-inv-def*
by (*simp add*: *orthogonal-transformation-matrix Rz-inv-mat-orthogonal matrix-vector-mul-linear*)

The named inverse rotations are two-sided inverses of the generators.

lemma *Rx-inv-left*: *Rx-inv* ∘ *Rx* = *id*
by (*rule ext*) (*simp add*: *Rx-inv-def Rx-def matrix-vector-mul-assoc Rx-mat-inverse-left*)

lemma *Rz-inv-left*: *Rz-inv* ∘ *Rz* = *id*
by (*rule ext*) (*simp add*: *Rz-inv-def Rz-def matrix-vector-mul-assoc Rz-mat-inverse-left*)

lemma *Rx-inv-right*: *Rx* ∘ *Rx-inv* = *id*
by (*rule ext*) (*simp add*: *Rx-inv-def Rx-def matrix-vector-mul-assoc Rx-mat-inverse-right*)

lemma *Rz-inv-right*: *Rz* ∘ *Rz-inv* = *id*
by (*rule ext*) (*simp add*: *Rz-inv-def Rz-def matrix-vector-mul-assoc Rz-mat-inverse-right*)

20 The rotations as bijections of the sphere

definition *sphere-perm* :: (*real^3* ⇒ *real^3*) ⇒ (*real^3* ⇒ *real^3*) **where**
sphere-perm f = *restrict f sphere2*

lemma *SO3-bij-betw-sphere2*:
assumes *f* ∈ *SO3*
shows *bij-betw f sphere2 sphere2*
proof −
have *f* ‘ *sphere2* ⊆ *sphere2*
using *rotation-preserves-sphere2*[*OF assms*] **by** *auto*
moreover have *sphere2* ⊆ *f* ‘ *sphere2*
proof
fix *y* **assume** *y*: *y* ∈ *sphere2*
from *SO3-bij*[*OF assms*] **obtain** *x* **where** *x*: *y* = *f x*
by (*meson bij-pointE*)
have *x* ∈ *sphere2*
proof −
from *assms* **have** *orth*: *orthogonal-transformation f*
unfolding *SO3-def rotation-def* **by** *auto*
from *x y* **have** *norm* (*f x*) = *1*
by (*simp add*: *sphere2-def*)

```
    moreover from orth have norm (f x) = norm x
      by (rule orthogonal-transformation-norm)
    ultimately show ?thesis
      by (simp add: sphere2-def)
  qed
  with x show y ∈ f ‘ sphere2
    by blast
 qed
 moreover have inj-on f sphere2
 proof −
  have inj f
    using SO3-bij[OF assms] by (rule bij-is-inj)
  thus ?thesis
    by (rule inj-on-subset) simp
 qed
 ultimately show ?thesis
  unfolding bij-betw-def by auto
qed

lemma sphere-perm-in-Bij:
 assumes f ∈ SO3
 shows sphere-perm f ∈ Bij sphere2
 using SO3-bij-betw-sphere2[OF assms]
 by (simp add: sphere-perm-def Bij-def)

lemma sphere-perm-mult:
 assumes sphere-perm f ∈ Bij sphere2 sphere-perm g ∈ Bij sphere2
 shows sphere-perm f ⊗BijGroup sphere2 sphere-perm g = sphere-perm (f ∘ g)
proof (rule ext)
 fix x
 show (sphere-perm f ⊗BijGroup sphere2 sphere-perm g) x = sphere-perm (f ∘ g)
x
 proof (cases x ∈ sphere2)
  case True
  from assms True have sphere-perm g x ∈ sphere2
    by (auto simp: Bij-def bij-betw-def)
  with assms True show ?thesis
    by (simp add: sphere-perm-def BijGroup-def compose-def restrict-def)
 next
  case False
  with assms show ?thesis
    by (simp add: sphere-perm-def BijGroup-def compose-def restrict-def)
 qed
qed

lemma sphere-perm-id: sphere-perm id = 1BijGroup sphere2
 by (simp add: sphere-perm-def BijGroup-def id-def)

lemma sphere-perm-Rx [simp]: sphere-perm Rx ∈ Bij sphere2
```

by (*rule sphere-perm-in-Bij*) (*rule Rx-in-SO3*)

lemma *sphere-perm-Rx-inv* [*simp*]: *sphere-perm Rx-inv* ∈ *Bij sphere2*
by (*rule sphere-perm-in-Bij*) (*rule Rx-inv-in-SO3*)

lemma *sphere-perm-Rz* [*simp*]: *sphere-perm Rz* ∈ *Bij sphere2*
by (*rule sphere-perm-in-Bij*) (*rule Rz-in-SO3*)

lemma *sphere-perm-Rz-inv* [*simp*]: *sphere-perm Rz-inv* ∈ *Bij sphere2*
by (*rule sphere-perm-in-Bij*) (*rule Rz-inv-in-SO3*)

lemma *sphere-perm-Rx-inv-left*:
sphere-perm Rx-inv $\otimes_{BijGroup\ sphere2}$ *sphere-perm Rx* = $\mathbf{1}_{BijGroup\ sphere2}$
by (*simp add*: *sphere-perm-mult Rx-inv-left sphere-perm-id*)

lemma *sphere-perm-Rx-inv-right*:
sphere-perm Rx $\otimes_{BijGroup\ sphere2}$ *sphere-perm Rx-inv* = $\mathbf{1}_{BijGroup\ sphere2}$
by (*simp add*: *sphere-perm-mult Rx-inv-right sphere-perm-id*)

lemma *sphere-perm-Rz-inv-left*:
sphere-perm Rz-inv $\otimes_{BijGroup\ sphere2}$ *sphere-perm Rz* = $\mathbf{1}_{BijGroup\ sphere2}$
by (*simp add*: *sphere-perm-mult Rz-inv-left sphere-perm-id*)

lemma *sphere-perm-Rz-inv-right*:
sphere-perm Rz $\otimes_{BijGroup\ sphere2}$ *sphere-perm Rz-inv* = $\mathbf{1}_{BijGroup\ sphere2}$
by (*simp add*: *sphere-perm-mult Rz-inv-right sphere-perm-id*)

lemma *inv-sphere-perm-Rx*:
$inv_{BijGroup\ sphere2}$ (*sphere-perm Rx*) = *sphere-perm Rx-inv*
proof −
interpret *B*: *group BijGroup sphere2*
by (*rule group-BijGroup*)
show *?thesis*
proof (*rule B.inv-equality*)
show *sphere-perm Rx-inv* $\otimes_{BijGroup\ sphere2}$ *sphere-perm Rx* = $\mathbf{1}_{BijGroup\ sphere2}$
by (*rule sphere-perm-Rx-inv-left*)
show *sphere-perm Rx* ∈ *carrier* (*BijGroup sphere2*)
by (*simp add*: *BijGroup-def*)
show *sphere-perm Rx-inv* ∈ *carrier* (*BijGroup sphere2*)
by (*simp add*: *BijGroup-def*)
qed
qed

lemma *inv-sphere-perm-Rz*:
$inv_{BijGroup\ sphere2}$ (*sphere-perm Rz*) = *sphere-perm Rz-inv*
proof −
interpret *B*: *group BijGroup sphere2*
by (*rule group-BijGroup*)
show *?thesis*

```
  proof (rule B.inv-equality)
   show sphere-perm Rz-inv ⊗_BijGroup sphere2 sphere-perm Rz = 1_BijGroup sphere2
     by (rule sphere-perm-Rz-inv-left)
    show sphere-perm Rz ∈ carrier (BijGroup sphere2)
     by (simp add: BijGroup-def)
    show sphere-perm Rz-inv ∈ carrier (BijGroup sphere2)
     by (simp add: BijGroup-def)
  qed
qed

interpretation sphere-Bij: group BijGroup sphere2
  by (rule group-BijGroup)
```

21 The map sigma: F_2 to SO(3)

The map *sigma* sends each reduced word in F_2 to the corresponding composition of R_x, R_z, and their inverses.

```
definition letter-to-rot :: bool × gen2 ⇒ (real^3 ⇒ real^3) where
  letter-to-rot p = (case p of
      (False, A) ⇒ Rx
    | (True,  A) ⇒ Rx-inv
    | (False, B) ⇒ Rz
    | (True,  B) ⇒ Rz-inv)

definition rho :: gen2 ⇒ (real^3 ⇒ real^3) where
  rho x = (case x of A ⇒ sphere-perm Rx | B ⇒ sphere-perm Rz)

lemma rho-in-BijGroup:
  rho ∈ Gen2 → carrier (BijGroup sphere2)
  by (auto simp: rho-def BijGroup-def)

lemma rho-image-subset-BijGroup:
  rho ‘ Gen2 ⊆ carrier (BijGroup sphere2)
  using rho-in-BijGroup by auto

definition sphere-rot-group :: (real^3 ⇒ real^3) monoid where
  sphere-rot-group =
    (BijGroup sphere2)⦇carrier := ⟨rho ‘ Gen2⟩_BijGroup sphere2⦈

lemma sphere-rot-subgroup:
  subgroup (carrier sphere-rot-group) (BijGroup sphere2)
  unfolding sphere-rot-group-def
  apply simp
  by (rule sphere-Bij.gen-subgroup-is-subgroup) (auto simp: rho-def BijGroup-def)

lemma sphere-rot-group-is-group: group sphere-rot-group
  unfolding sphere-rot-group-def
  by (rule sphere-Bij.subgroup-imp-group)
```

(*rule sphere-Bij.gen-subgroup-is-subgroup*, *rule rho-image-subset-BijGroup*)

interpretation *sphere-rot*: *group sphere-rot-group*
by (*rule sphere-rot-group-is-group*)

lemma *rho-in-sphere-rot-group*:
rho ∈ *Gen2* → *carrier sphere-rot-group*
by (*auto simp*: *sphere-rot-group-def intro*: *gen-span.gen-gens*)

lemma *sphere-rot-inclusion-hom*:
id ∈ *hom sphere-rot-group* (*BijGroup sphere2*)
using *sphere-Bij.gen-span-closed*[*OF rho-image-subset-BijGroup*]
by (*auto simp*: *hom-def sphere-rot-group-def*)

lemma *sphere-rot-generated*:
⟨*rho* ‘ *Gen2*⟩$_{\textit{sphere-rot-group}}$ = *carrier sphere-rot-group*
proof −
let *?H* = ⟨*rho* ‘ *Gen2*⟩$_{\textit{BijGroup sphere2}}$
let *?G* = (*BijGroup sphere2*)(|*carrier* := *?H*|)
have *sub*: *subgroup ?H* (*BijGroup sphere2*)
by (*rule sphere-Bij.gen-subgroup-is-subgroup*) (*rule rho-image-subset-BijGroup*)
have *left*: ⟨*rho* ‘ *Gen2*⟩$_{\textit{sphere-rot-group}}$ ⊆ *carrier sphere-rot-group*
by (*rule sphere-rot.gen-span-closed*) (*use rho-in-sphere-rot-group* **in** *auto*)
have *right*: *?H* ⊆ ⟨*rho* ‘ *Gen2*⟩$_{\textit{?G}}$
proof
fix *x*
assume *x* ∈ *?H*
thus *x* ∈ ⟨*rho* ‘ *Gen2*⟩$_{\textit{?G}}$
proof (*induct rule*: *gen-span.induct*)
case *gen-one*
have **1**$_{\textit{?G}}$ ∈ ⟨*rho* ‘ *Gen2*⟩$_{\textit{?G}}$
by (*rule gen-span.gen-one*)
thus *?case* **by** *simp*
next
case (*gen-gens x*)
thus *?case*
by (*rule gen-span.gen-gens*)
next
case (*gen-inv x*)
have *inv-eq*: *inv*$_{\textit{?G}}$ *x* = *inv*$_{\textit{BijGroup sphere2}}$ *x*
using *sphere-Bij.m-inv-consistent*[*OF sub gen-inv.hyps*(*1*)] **.**
have *inv*$_{\textit{?G}}$ *x* ∈ ⟨*rho* ‘ *Gen2*⟩$_{\textit{?G}}$
by (*rule gen-span.gen-inv*) (*rule gen-inv.hyps*(*2*))
thus *?case*
using *inv-eq* **by** *metis*
next
case (*gen-mult x y*)
have *x* ⊗$_{\textit{?G}}$ *y* ∈ ⟨*rho* ‘ *Gen2*⟩$_{\textit{?G}}$
by (*rule gen-span.gen-mult*) (*use gen-mult.hyps* **in** *auto*)

```
    thus ?case by simp
  qed
 qed
 show ?thesis
  using left right by (auto simp: sphere-rot-group-def)
qed

lemma sphere-Bij-lift-gi-rho:
 assumes snd p ∈ Gen2
 shows sphere-Bij.lift-gi rho p = sphere-perm (letter-to-rot p)
 using assms
 by (cases p)
  (auto simp: rho-def letter-to-rot-def sphere-Bij.lift-gi-def
    inv-sphere-perm-Rx inv-sphere-perm-Rz split: bool.splits gen2.splits)

lemma sphere-rot-inv-rho-eq-sphere-Bij:
 assumes i ∈ Gen2
 shows inv_{sphere-rot-group} (rho i) = inv_{BijGroup sphere2} (rho i)
proof −
 have rho-i: rho i ∈ carrier sphere-rot-group
  using rho-in-sphere-rot-group assms by auto
 have inv_{(BijGroup sphere2)(|carrier := carrier sphere-rot-group|)} (rho i) =
   inv_{BijGroup sphere2} (rho i)
  using sphere-Bij.m-inv-consistent[OF sphere-rot-subgroup rho-i] .
 thus ?thesis
  by (simp add: sphere-rot-group-def)
qed

lemma sphere-rot-lift-gi-rho-eq-sphere-Bij:
 assumes snd p ∈ Gen2
 shows sphere-rot.lift-gi rho p = sphere-Bij.lift-gi rho p
 using assms
 by (cases p)
  (auto simp: sphere-rot.lift-gi-def sphere-Bij.lift-gi-def
    sphere-rot-inv-rho-eq-sphere-Bij)

lemma sphere-rot-lift-rho-eq-sphere-Bij:
 assumes w ∈ lists (UNIV × Gen2)
 shows sphere-rot.lift rho w = sphere-Bij.lift rho w
 using assms
proof (induct w)
 case Nil
 show ?case
 proof −
  have sphere-rot.lift rho [] = 1_{sphere-rot-group}
   by simp
  also have ... = 1_{BijGroup sphere2}
   by (simp add: sphere-rot-group-def)
  also have ... = sphere-Bij.lift rho []
```

```
    by simp
  finally show ?case .
 qed
next
 case (Cons p w)
 have p-gen: snd p ∈ Gen2
  by (simp add: Gen2-UNIV)
 have w-lists: w ∈ lists (UNIV × Gen2)
  using Cons.prems by (auto simp: Gen2-UNIV)
 have IH: sphere-rot.lift rho w = sphere-Bij.lift rho w
  using Cons.hyps w-lists by simp
 have gi-rot-closed: sphere-rot.lift-gi rho p ∈ carrier sphere-rot-group
  using rho-in-sphere-rot-group p-gen by (rule sphere-rot.lift-gi-closed)
 have tail-rot-closed: sphere-rot.lift rho w ∈ carrier sphere-rot-group
  using rho-in-sphere-rot-group w-lists by (rule sphere-rot.lift-closed)
 have gi-Bij-closed: sphere-Bij.lift-gi rho p ∈ carrier (BijGroup sphere2)
  using rho-in-BijGroup p-gen by (rule sphere-Bij.lift-gi-closed)
 have tail-Bij-closed: sphere-Bij.lift rho w ∈ carrier (BijGroup sphere2)
  using rho-in-BijGroup w-lists by (rule sphere-Bij.lift-closed)
 have sphere-rot.lift rho (p # w) =
    sphere-rot.lift-gi rho p ⊗_sphere-rot-group sphere-rot.lift rho w
  using gi-rot-closed tail-rot-closed
  by (simp add: sphere-rot.lift-def)
 also have ... =
    sphere-Bij.lift-gi rho p ⊗_BijGroup sphere2 sphere-Bij.lift rho w
  using IH sphere-rot-lift-gi-rho-eq-sphere-Bij[OF p-gen]
  by (simp add: sphere-rot-group-def)
 also have ... = sphere-Bij.lift rho (p # w)
  using gi-Bij-closed tail-Bij-closed
  by (simp add: sphere-Bij.lift-def)
 finally show ?case .
qed

lemma sphere-Bij-lift-rho-injective-if-ping-pong:
 fixes Xin Xout :: gen2 ⇒ (real^3) set
 assumes sub-out: ∀ i∈Gen2. Xout i ⊆ sphere2
   and sub-in: ∀ i∈Gen2. Xin i ⊆ sphere2
   and disj-out: ∀ i∈Gen2. ∀ j∈Gen2. i ≠ j ⟶ Xout i ∩ Xout j = {}
   and disj-in: ∀ i∈Gen2. ∀ j∈Gen2. i ≠ j ⟶ Xin i ∩ Xin j = {}
   and disj-cross: ∀ i∈Gen2. ∀ j∈Gen2. Xin i ∩ Xout j = {}
   and x-sphere: x ∈ sphere2
   and ping: ∀ i∈Gen2. rho i ‘ (sphere2 − Xout i) ⊆ Xin i
  and pong: ∀ i∈Gen2. (inv_sphere-rot-group (rho i)) ‘ (sphere2 − Xin i) ⊆ Xout i
 shows inj-on (sphere-Bij.lift rho) (carrier F2)
proof −
 have iso: sphere-rot.lift rho ∈ iso F2 sphere-rot-group
 proof (rule ping-pong-lemma[
    where X = sphere2 and I = Gen2 and act = id and g = rho
     and Xout = Xout and Xin = Xin and x = x])
```

```
    show group sphere-rot-group
      by (rule sphere-rot-group-is-group)
    show id ∈ hom sphere-rot-group (BijGroup sphere2)
      by (rule sphere-rot-inclusion-hom)
    show rho ∈ Gen2 → carrier sphere-rot-group
      by (rule rho-in-sphere-rot-group)
    show ⟨rho ‘ Gen2⟩_sphere-rot-group = carrier sphere-rot-group
      by (rule sphere-rot-generated)
    show ∀ i∈Gen2. Xout i ⊆ sphere2
      by (rule sub-out)
    show ∀ i∈Gen2. Xin i ⊆ sphere2
      by (rule sub-in)
    show ∀ i∈Gen2. ∀ j∈Gen2. i ≠ j ⟶ Xout i ∩ Xout j = {}
      by (rule disj-out)
    show ∀ i∈Gen2. ∀ j∈Gen2. i ≠ j ⟶ Xin i ∩ Xin j = {}
      by (rule disj-in)
    show ∀ i∈Gen2. ∀ j∈Gen2. Xin i ∩ Xout j = {}
      by (rule disj-cross)
    show x ∈ sphere2
      by (rule x-sphere)
    show ∀ i. Gen2 = {i} ⟶ x ∉ Xout i ∧ x ∉ Xin i
      by auto
    show ∀ i∈Gen2. id (rho i) ‘ (sphere2 − Xout i) ⊆ Xin i
      using ping by simp
    show ∀ i∈Gen2. id (inv_sphere-rot-group (rho i)) ‘ (sphere2 − Xin i) ⊆ Xout i
      using pong by simp
  qed
  have inj-rot: inj-on (sphere-rot.lift rho) (carrier F2)
    using iso by (auto simp: iso-def bij-betw-def)
  show ?thesis
  proof (rule inj-onI)
    fix w v
    assume w: w ∈ carrier F2 and v: v ∈ carrier F2
      and eq: sphere-Bij.lift rho w = sphere-Bij.lift rho v
    have w-lists: w ∈ lists (UNIV × Gen2)
      using w by (simp add: F2-carrier-iff)
    have v-lists: v ∈ lists (UNIV × Gen2)
      using v by (simp add: F2-carrier-iff)
    have sphere-rot.lift rho w = sphere-Bij.lift rho w
      using w-lists by (rule sphere-rot-lift-rho-eq-sphere-Bij)
    also have ... = sphere-Bij.lift rho v
      by (rule eq)
    also have ... = sphere-rot.lift rho v
      using v-lists by (rule sphere-rot-lift-rho-eq-sphere-Bij[symmetric])
    finally have sphere-rot.lift rho w = sphere-rot.lift rho v .
    with inj-rot w v show w = v
      by (auto simp: inj-on-def)
  qed
qed
```

definition *sigma* :: (*bool* × *gen2*) *list* ⇒ (*real^3* ⇒ *real^3*) **where**
sigma w = *foldr* (λ*p f*. *letter-to-rot p* ∘ *f*) *w id*

lemma *letter-to-rot-cancel-left* [*simp*]:
letter-to-rot (*True*, *A*) ∘ *letter-to-rot* (*False*, *A*) = *id*
letter-to-rot (*False*, *A*) ∘ *letter-to-rot* (*True*, *A*) = *id*
letter-to-rot (*True*, *B*) ∘ *letter-to-rot* (*False*, *B*) = *id*
letter-to-rot (*False*, *B*) ∘ *letter-to-rot* (*True*, *B*) = *id*
by (*simp-all add*: *letter-to-rot-def Rx-inv-left Rx-inv-right Rz-inv-left Rz-inv-right*)

lemma *letter-to-rot-canceling*:
assumes *canceling p q*
shows *letter-to-rot p* ∘ *letter-to-rot q* = *id*
using *assms*
by (*cases p*; *cases q*)
(*auto split*: *bool.splits gen2.splits*
simp: *canceling-def letter-to-rot-def Rx-inv-left Rx-inv-right Rz-inv-left Rz-inv-right*)

lemma *sigma-Nil* [*simp*]: *sigma* [] = *id*
by (*simp add*: *sigma-def*)

lemma *sigma-Cons* [*simp*]: *sigma* (*p* # *w*) = *letter-to-rot p* ∘ *sigma w*
by (*simp add*: *sigma-def*)

lemma *sigma-append* [*simp*]: *sigma* (*u* @ *v*) = *sigma u* ∘ *sigma v*
by (*induct u*) (*simp-all add*: *o-assoc*)

lemma *sigma-singleton* [*simp*]: *sigma* [*p*] = *letter-to-rot p*
by (*simp add*: *fun-eq-iff*)

lemma *sigma-doubleton-canceling*:
assumes *canceling p q*
shows *sigma* [*p*, *q*] = *id*
using *letter-to-rot-canceling*[*OF assms*]
by (*simp add*: *o-def fun-eq-iff*)

lemma *sigma-cancels-to-1*:
assumes *cancels-to-1 x y*
shows *sigma x* = *sigma y*
proof −
from *assms* **obtain** *xs1 x1 x2 xs2*
where *x-eq*: *x* = *xs1* @ *x1* # *x2* # *xs2*
and *y-eq*: *y* = *xs1* @ *xs2*
and *cancel*: *canceling x1 x2*
by (*rule cancels-to-1-unfold*)
have *sigma x* = *sigma xs1* ∘ *sigma* [*x1*, *x2*] ∘ *sigma xs2*
by (*simp add*: *x-eq o-assoc*)
also have . . . = *sigma xs1* ∘ *sigma xs2*

```
    using letter-to-rot-canceling[OF cancel]
    by (auto simp: fun-eq-iff o-def)
  also have ... = sigma y
    by (simp add: y-eq)
  finally show ?thesis .
qed

lemma sigma-cancels-to:
  assumes cancels-to x y
  shows sigma x = sigma y
  using assms unfolding cancels-to-def
proof (induct rule: rtranclp-induct)
  case base
  show ?case by simp
next
  case (step y z)
  then show ?case
    using sigma-cancels-to-1 by simp
qed

lemma sigma-normalize [simp]: sigma (normalize w) = sigma w
  using sigma-cancels-to[OF normalized-cancels-to] by simp

lemma letter-to-rot-in-SO3: letter-to-rot p ∈ SO3
  by (cases p)
    (auto split: bool.splits gen2.splits
      simp: letter-to-rot-def Rx-in-SO3 Rz-in-SO3 Rx-inv-in-SO3 Rz-inv-in-SO3)

lemma sphere-perm-letter-to-rot-in-Bij [simp]:
  sphere-perm (letter-to-rot p) ∈ Bij sphere2
  by (rule sphere-perm-in-Bij) (rule letter-to-rot-in-SO3)

lemma sigma-in-SO3: sigma w ∈ SO3
proof (induct w)
  case Nil
  show ?case
    using id-in-SO3 by (simp add: id-def)
next
  case (Cons p w)
  have letter-to-rot p ∘ sigma w ∈ SO3
    using letter-to-rot-in-SO3 Cons.hyps by (rule SO3-closed-compose)
  then show ?case by (simp add: o-def)
qed

lemma sigma-preserves-sphere2:
  assumes x ∈ sphere2
  shows sigma w x ∈ sphere2
  using rotation-preserves-sphere2[OF sigma-in-SO3 assms] .
```

lemma *sigma-bij*: *bij* (*sigma w*)
using *SO3-bij sigma-in-SO3* **by** *blast*

lemma *sphere-perm-sigma-in-Bij* [*simp*]:
sphere-perm (*sigma w*) ∈ *Bij sphere2*
by (*rule sphere-perm-in-Bij*) (*rule sigma-in-SO3*)

lemma *sphere-Bij-lift-rho-eq-sigma*:
assumes *w* ∈ *lists* (*UNIV* × *Gen2*)
shows *sphere-Bij.lift rho w* = *sphere-perm* (*sigma w*)
using *assms*
proof (*induct w*)
case *Nil*
show *?case*
by (*simp add*: *sphere-perm-id*[*symmetric*] *id-def*)
next
case (*Cons p w*)
have *p-gen*: *snd p* ∈ *Gen2*
by (*simp add*: *Gen2-UNIV*)
have *w-lists*: *w* ∈ *lists* (*UNIV* × *Gen2*)
using *Cons.prems* **by** (*auto simp*: *Gen2-UNIV*)
have *IH*: *sphere-Bij.lift rho w* = *sphere-perm* (*sigma w*)
using *Cons.hyps w-lists* **by** *simp*
have *singleton-lists*: [*p*] ∈ *lists* (*UNIV* × *Gen2*)
by (*simp add*: *Gen2-UNIV*)
have *sphere-Bij.lift rho* (*p* # *w*) = *sphere-Bij.lift rho* ([*p*] @ *w*)
by *simp*
also have ... = *sphere-Bij.lift rho* [*p*] ⊗$_{BijGroup\ sphere2}$ *sphere-Bij.lift rho w*
using *sphere-Bij.lift-append*[*OF rho-in-BijGroup singleton-lists w-lists*] **.**
also have *sphere-Bij.lift rho* [*p*] = *sphere-perm* (*letter-to-rot p*)
proof −
have *gi-closed*: *sphere-Bij.lift-gi rho p* ∈ *carrier* (*BijGroup sphere2*)
using *rho-in-BijGroup p-gen* **by** (*rule sphere-Bij.lift-gi-closed*)
have *sphere-Bij.lift rho* [*p*] = *sphere-Bij.lift-gi rho p*
using *rho-in-BijGroup p-gen*
by (*simp add*: *sphere-Bij.lift-def gi-closed*)
also have ... = *sphere-perm* (*letter-to-rot p*)
using *p-gen* **by** (*rule sphere-Bij-lift-gi-rho*)
finally show *?thesis* **.**
qed
also have *sphere-perm* (*letter-to-rot p*) ⊗$_{BijGroup\ sphere2}$ *sphere-Bij.lift rho w* =
sphere-perm (*letter-to-rot p*) ⊗$_{BijGroup\ sphere2}$ *sphere-perm* (*sigma w*)
using *IH* **by** *simp*
also have ... = *sphere-perm* (*letter-to-rot p* ∘ *sigma w*)
by (*rule sphere-perm-mult*) *simp-all*
also have ... = *sphere-perm* (*sigma* (*p* # *w*))
by *simp*
finally show *?case* **.**
qed

lemma *sigma-injective-if-sphere-lift-injective*:
  assumes *inj-on* (*sphere-Bij.lift rho*) (*carrier F2*)
  shows *inj-on sigma* (*carrier F2*)
proof (*rule inj-onI*)
  fix *w v*
  assume *w*: *w* ∈ *carrier F2* **and** *v*: *v* ∈ *carrier F2* **and** *eq*: *sigma w* = *sigma v*
  have *w-lists*: *w* ∈ *lists* (*UNIV* × *Gen2*)
    using *w* **by** (*simp add*: *F2-carrier-iff*)
  have *v-lists*: *v* ∈ *lists* (*UNIV* × *Gen2*)
    using *v* **by** (*simp add*: *F2-carrier-iff*)
  have *sphere-Bij.lift rho w* = *sphere-perm* (*sigma w*)
    using *w-lists* **by** (*rule sphere-Bij-lift-rho-eq-sigma*)
  also have ... = *sphere-perm* (*sigma v*)
    using *eq* **by** *simp*
  also have ... = *sphere-Bij.lift rho v*
    using *v-lists* **by** (*rule sphere-Bij-lift-rho-eq-sigma*[*symmetric*])
  finally have *lift-eq*: *sphere-Bij.lift rho w* = *sphere-Bij.lift rho v* **.**
  show *w* = *v*
    using *assms w v lift-eq* **by** (*auto simp*: *inj-on-def*)
qed

lemma *sigma-injective-if-ping-pong*:
  fixes *Xin Xout* :: *gen2* ⇒ (*real^3*) *set*
  assumes *sub-out*: ∀ *i*∈*Gen2*. *Xout i* ⊆ *sphere2*
    and *sub-in*: ∀ *i*∈*Gen2*. *Xin i* ⊆ *sphere2*
    and *disj-out*: ∀ *i*∈*Gen2*. ∀ *j*∈*Gen2*. *i* ≠ *j* ⟶ *Xout i* ∩ *Xout j* = {}
    and *disj-in*: ∀ *i*∈*Gen2*. ∀ *j*∈*Gen2*. *i* ≠ *j* ⟶ *Xin i* ∩ *Xin j* = {}
    and *disj-cross*: ∀ *i*∈*Gen2*. ∀ *j*∈*Gen2*. *Xin i* ∩ *Xout j* = {}
    and *x-sphere*: *x* ∈ *sphere2*
    and *ping*: ∀ *i*∈*Gen2*. *rho i* ‘ (*sphere2* − *Xout i*) ⊆ *Xin i*
    and *pong*: ∀ *i*∈*Gen2*. (*inv*$_{\textit{sphere-rot-group}}$ (*rho i*)) ‘ (*sphere2* − *Xin i*) ⊆ *Xout i*
  shows *inj-on sigma* (*carrier F2*)
  by (*rule sigma-injective-if-sphere-lift-injective*)
    (*rule sphere-Bij-lift-rho-injective-if-ping-pong*[
      OF sub-out sub-in disj-out disj-in disj-cross x-sphere ping pong])

The ping-pong criterion gives a reusable injectivity criterion for *sigma*.

22 Arithmetic freeness invariant

The classical proof that the above rotations generate a free subgroup is arithmetic rather than geometric: scale each letter by *3*, so every word has entries in the integer quadratic ring generated by *sqrt 2*. A nonempty reduced word leaves a coefficient nonzero modulo *3* after applying it to a suitable basis vector, whereas the identity matrix would have all scaled off-axis coefficients divisible by *3*.

The following definitions make that invariant explicit. They are deliberately separated from real-vector semantics so that the finite modulo-*3* induction can be discharged by simplification and case analysis before being connected back to *sigma*.

type-synonym *zsqrt2* = *int* × *int*
type-synonym *zvec3* = *zsqrt2* × *zsqrt2* × *zsqrt2*

definition *zsqrt2-val* :: *zsqrt2* ⇒ *real* **where**
zsqrt2-val q = *of-int* (*fst q*) + *of-int* (*snd q*) ∗ *sqrt 2*

definition *zsqrt2-add* :: *zsqrt2* ⇒ *zsqrt2* ⇒ *zsqrt2* **where**
zsqrt2-add p q = (*fst p* + *fst q*, *snd p* + *snd q*)

definition *zsqrt2-neg* :: *zsqrt2* ⇒ *zsqrt2* **where**
zsqrt2-neg p = (− *fst p*, − *snd p*)

definition *zsqrt2-sub* :: *zsqrt2* ⇒ *zsqrt2* ⇒ *zsqrt2* **where**
zsqrt2-sub p q = *zsqrt2-add p* (*zsqrt2-neg q*)

definition *zsqrt2-scale* :: *int* ⇒ *zsqrt2* ⇒ *zsqrt2* **where**
zsqrt2-scale n p = (*n* ∗ *fst p*, *n* ∗ *snd p*)

definition *zsqrt2-mul-sqrt2* :: *zsqrt2* ⇒ *zsqrt2* **where**
zsqrt2-mul-sqrt2 p = (*2* ∗ *snd p*, *fst p*)

definition *zsqrt2-div3* :: *zsqrt2* ⇒ *bool* **where**
zsqrt2-div3 p ⟷ *3 dvd fst p* ∧ *3 dvd snd p*

definition *zvec3-div3* :: *zvec3* ⇒ *bool* **where**
zvec3-div3 v ⟷
(*case v of* (*x*, *y*, *z*) ⇒ *zsqrt2-div3 x* ∧ *zsqrt2-div3 y* ∧ *zsqrt2-div3 z*)

definition *zvec3-scale* :: *int* ⇒ *zvec3* ⇒ *zvec3* **where**
zvec3-scale n v = (*case v of* (*x*, *y*, *z*) ⇒
(*zsqrt2-scale n x*, *zsqrt2-scale n y*, *zsqrt2-scale n z*))

definition *zvec3-val* :: *zvec3* ⇒ *real^3* **where**
zvec3-val v = (*case v of* (*x*, *y*, *z*) ⇒
vector [*zsqrt2-val x*, *zsqrt2-val y*, *zsqrt2-val z*])

definition *ze-x* :: *zvec3* **where**
ze-x = ((*1*, *0*), (*0*, *0*), (*0*, *0*))

definition *ze-y* :: *zvec3* **where**
ze-y = ((*0*, *0*), (*1*, *0*), (*0*, *0*))

definition *ze-z* :: *zvec3* **where**
ze-z = ((*0*, *0*), (*0*, *0*), (*1*, *0*))

```
definition zRx-step :: zvec3 ⇒ zvec3 where
  zRx-step v = (case v of (x, y, z) ⇒
    (zsqrt2-scale 3 x,
     zsqrt2-sub y (zsqrt2-scale 2 (zsqrt2-mul-sqrt2 z)),
     zsqrt2-add (zsqrt2-scale 2 (zsqrt2-mul-sqrt2 y)) z))

definition zRx-inv-step :: zvec3 ⇒ zvec3 where
  zRx-inv-step v = (case v of (x, y, z) ⇒
    (zsqrt2-scale 3 x,
     zsqrt2-add y (zsqrt2-scale 2 (zsqrt2-mul-sqrt2 z)),
     zsqrt2-sub z (zsqrt2-scale 2 (zsqrt2-mul-sqrt2 y))))

definition zRz-step :: zvec3 ⇒ zvec3 where
  zRz-step v = (case v of (x, y, z) ⇒
    (zsqrt2-sub x (zsqrt2-scale 2 (zsqrt2-mul-sqrt2 y)),
     zsqrt2-add (zsqrt2-scale 2 (zsqrt2-mul-sqrt2 x)) y,
     zsqrt2-scale 3 z))

definition zRz-inv-step :: zvec3 ⇒ zvec3 where
  zRz-inv-step v = (case v of (x, y, z) ⇒
    (zsqrt2-add x (zsqrt2-scale 2 (zsqrt2-mul-sqrt2 y)),
     zsqrt2-sub y (zsqrt2-scale 2 (zsqrt2-mul-sqrt2 x)),
     zsqrt2-scale 3 z))

definition zletter-step :: bool × gen2 ⇒ zvec3 ⇒ zvec3 where
  zletter-step p = (case p of
      (False, A) ⇒ zRx-step
    | (True,  A) ⇒ zRx-inv-step
    | (False, B) ⇒ zRz-step
    | (True,  B) ⇒ zRz-inv-step)

definition zword-step :: (bool × gen2) list ⇒ zvec3 ⇒ zvec3 where
  zword-step w v = foldr (λp f. zletter-step p ∘ f) w id v

definition zword-witness :: (bool × gen2) list ⇒ zvec3 where
  zword-witness w = (case snd (last w) of A ⇒ ze-y | B ⇒ ze-x)

datatype r3 = R0 | R1 | R2

type-synonym rcoef = r3 × r3
type-synonym rvec3 = rcoef × rcoef × rcoef

fun r3-add :: r3 ⇒ r3 ⇒ r3 where
  r3-add R0 x = x
| r3-add x R0 = x
| r3-add R1 R1 = R2
| r3-add R1 R2 = R0
| r3-add R2 R1 = R0
| r3-add R2 R2 = R1
```

fun *r3-neg* :: *r3* ⇒ *r3* **where**
  r3-neg R0 = *R0*
| *r3-neg R1* = *R2*
| *r3-neg R2* = *R1*

definition *r3-sub* :: *r3* ⇒ *r3* ⇒ *r3* **where**
  r3-sub x y = *r3-add x* (*r3-neg y*)

fun *r3-double* :: *r3* ⇒ *r3* **where**
  r3-double R0 = *R0*
| *r3-double R1* = *R2*
| *r3-double R2* = *R1*

definition *rcoef-add* :: *rcoef* ⇒ *rcoef* ⇒ *rcoef* **where**
  rcoef-add p q = (*r3-add* (*fst p*) (*fst q*), *r3-add* (*snd p*) (*snd q*))

definition *rcoef-neg* :: *rcoef* ⇒ *rcoef* **where**
  rcoef-neg p = (*r3-neg* (*fst p*), *r3-neg* (*snd p*))

definition *rcoef-sub* :: *rcoef* ⇒ *rcoef* ⇒ *rcoef* **where**
  rcoef-sub p q = *rcoef-add p* (*rcoef-neg q*)

definition *rcoef-double* :: *rcoef* ⇒ *rcoef* **where**
  rcoef-double p = (*r3-double* (*fst p*), *r3-double* (*snd p*))

definition *rcoef-mul-sqrt2* :: *rcoef* ⇒ *rcoef* **where**
  rcoef-mul-sqrt2 p = (*r3-double* (*snd p*), *fst p*)

definition *rRx-step* :: *rvec3* ⇒ *rvec3* **where**
  rRx-step v = (*case v of* (*x*, *y*, *z*) ⇒
    ((*R0*, *R0*),
     rcoef-sub y (*rcoef-double* (*rcoef-mul-sqrt2 z*)),
     rcoef-add (*rcoef-double* (*rcoef-mul-sqrt2 y*)) *z*))

definition *rRx-inv-step* :: *rvec3* ⇒ *rvec3* **where**
  rRx-inv-step v = (*case v of* (*x*, *y*, *z*) ⇒
    ((*R0*, *R0*),
     rcoef-add y (*rcoef-double* (*rcoef-mul-sqrt2 z*)),
     rcoef-sub z (*rcoef-double* (*rcoef-mul-sqrt2 y*))))

definition *rRz-step* :: *rvec3* ⇒ *rvec3* **where**
  rRz-step v = (*case v of* (*x*, *y*, *z*) ⇒
    (*rcoef-sub x* (*rcoef-double* (*rcoef-mul-sqrt2 y*)),
     rcoef-add (*rcoef-double* (*rcoef-mul-sqrt2 x*)) *y*,
     (*R0*, *R0*)))

definition *rRz-inv-step* :: *rvec3* ⇒ *rvec3* **where**
  rRz-inv-step v = (*case v of* (*x*, *y*, *z*) ⇒

```
    (rcoef-add x (rcoef-double (rcoef-mul-sqrt2 y)),
     rcoef-sub y (rcoef-double (rcoef-mul-sqrt2 x)),
     (R0, R0)))

definition rletter-step :: bool × gen2 ⇒ rvec3 ⇒ rvec3 where
  rletter-step p = (case p of
      (False, A) ⇒ rRx-step
    | (True,  A) ⇒ rRx-inv-step
    | (False, B) ⇒ rRz-step
    | (True,  B) ⇒ rRz-inv-step)

definition rword-step :: (bool × gen2) list ⇒ rvec3 ⇒ rvec3 where
  rword-step w v = foldr (λp f. rletter-step p ∘ f) w id v

definition re-x :: rvec3 where
  re-x = ((R1, R0), (R0, R0), (R0, R0))

definition re-y :: rvec3 where
  re-y = ((R0, R0), (R1, R0), (R0, R0))

definition rword-witness :: (bool × gen2) list ⇒ rvec3 where
  rword-witness w = (case snd (last w) of A ⇒ re-y | B ⇒ re-x)

definition rzero-vec :: rvec3 where
  rzero-vec = ((R0, R0), (R0, R0), (R0, R0))

definition rstate :: bool × gen2 ⇒ rvec3 ⇒ bool where
  rstate p v ⟷
    v ∈ (case p of
      (False, A) ⇒ {
          ((R0, R0), (R0, R1), (R1, R0)),
          ((R0, R0), (R0, R2), (R2, R0)),
          ((R0, R0), (R1, R0), (R0, R2)),
          ((R0, R0), (R2, R0), (R0, R1))}
    | (True, A) ⇒ {
          ((R0, R0), (R0, R1), (R2, R0)),
          ((R0, R0), (R0, R2), (R1, R0)),
          ((R0, R0), (R1, R0), (R0, R1)),
          ((R0, R0), (R2, R0), (R0, R2))}
    | (False, B) ⇒ {
          ((R0, R1), (R1, R0), (R0, R0)),
          ((R0, R2), (R2, R0), (R0, R0)),
          ((R1, R0), (R0, R2), (R0, R0)),
          ((R2, R0), (R0, R1), (R0, R0))}
    | (True, B) ⇒ {
          ((R0, R1), (R2, R0), (R0, R0)),
          ((R0, R2), (R1, R0), (R0, R0)),
          ((R1, R0), (R0, R1), (R0, R0)),
          ((R2, R0), (R0, R2), (R0, R0))})
```

lemma *rstate-single*:
rstate p (*rletter-step p* (*case snd p of A* ⇒ *re-y* | *B* ⇒ *re-x*))
by (*cases p*)
(*auto split*: *bool.splits gen2.splits*
simp: *rstate-def rletter-step-def rRx-step-def rRx-inv-step-def*
rRz-step-def rRz-inv-step-def re-x-def re-y-def rcoef-add-def
rcoef-sub-def rcoef-neg-def rcoef-double-def rcoef-mul-sqrt2-def r3-sub-def)

lemma *rstate-step-FA-FA*:
rstate (*False*, *A*) *v* ⟹ *rstate* (*False*, *A*) (*rletter-step* (*False*, *A*) *v*)
by (*simp add*: *rstate-def rletter-step-def rRx-step-def rcoef-add-def rcoef-sub-def*
rcoef-neg-def rcoef-double-def rcoef-mul-sqrt2-def r3-sub-def; *elim disjE*; *simp*)

lemma *rstate-step-FA-FB*:
rstate (*False*, *B*) *v* ⟹ *rstate* (*False*, *A*) (*rletter-step* (*False*, *A*) *v*)
by (*simp add*: *rstate-def rletter-step-def rRx-step-def rcoef-add-def rcoef-sub-def*
rcoef-neg-def rcoef-double-def rcoef-mul-sqrt2-def r3-sub-def; *elim disjE*; *simp*)

lemma *rstate-step-FA-TB*:
rstate (*True*, *B*) *v* ⟹ *rstate* (*False*, *A*) (*rletter-step* (*False*, *A*) *v*)
by (*simp add*: *rstate-def rletter-step-def rRx-step-def rcoef-add-def rcoef-sub-def*
rcoef-neg-def rcoef-double-def rcoef-mul-sqrt2-def r3-sub-def; *elim disjE*; *simp*)

lemma *rstate-step-TA-TA*:
rstate (*True*, *A*) *v* ⟹ *rstate* (*True*, *A*) (*rletter-step* (*True*, *A*) *v*)
by (*simp add*: *rstate-def rletter-step-def rRx-inv-step-def rcoef-add-def rcoef-sub-def*
rcoef-neg-def rcoef-double-def rcoef-mul-sqrt2-def r3-sub-def; *elim disjE*; *simp*)

lemma *rstate-step-TA-FB*:
rstate (*False*, *B*) *v* ⟹ *rstate* (*True*, *A*) (*rletter-step* (*True*, *A*) *v*)
by (*simp add*: *rstate-def rletter-step-def rRx-inv-step-def rcoef-add-def rcoef-sub-def*
rcoef-neg-def rcoef-double-def rcoef-mul-sqrt2-def r3-sub-def; *elim disjE*; *simp*)

lemma *rstate-step-TA-TB*:
rstate (*True*, *B*) *v* ⟹ *rstate* (*True*, *A*) (*rletter-step* (*True*, *A*) *v*)
by (*simp add*: *rstate-def rletter-step-def rRx-inv-step-def rcoef-add-def rcoef-sub-def*
rcoef-neg-def rcoef-double-def rcoef-mul-sqrt2-def r3-sub-def; *elim disjE*; *simp*)

lemma *rstate-step-FB-FB*:
rstate (*False*, *B*) *v* ⟹ *rstate* (*False*, *B*) (*rletter-step* (*False*, *B*) *v*)
by (*simp add*: *rstate-def rletter-step-def rRz-step-def rcoef-add-def rcoef-sub-def*
rcoef-neg-def rcoef-double-def rcoef-mul-sqrt2-def r3-sub-def; *elim disjE*; *simp*)

lemma *rstate-step-FB-FA*:
rstate (*False*, *A*) *v* ⟹ *rstate* (*False*, *B*) (*rletter-step* (*False*, *B*) *v*)
by (*simp add*: *rstate-def rletter-step-def rRz-step-def rcoef-add-def rcoef-sub-def*
rcoef-neg-def rcoef-double-def rcoef-mul-sqrt2-def r3-sub-def; *elim disjE*; *simp*)

lemma *rstate-step-FB-TA*:
rstate (*True*, *A*) *v* ⟹ *rstate* (*False*, *B*) (*rletter-step* (*False*, *B*) *v*)
by (*simp add*: *rstate-def rletter-step-def rRz-step-def rcoef-add-def rcoef-sub-def rcoef-neg-def rcoef-double-def rcoef-mul-sqrt2-def r3-sub-def*; *elim disjE*; *simp*)

lemma *rstate-step-TB-TB*:
rstate (*True*, *B*) *v* ⟹ *rstate* (*True*, *B*) (*rletter-step* (*True*, *B*) *v*)
by (*simp add*: *rstate-def rletter-step-def rRz-inv-step-def rcoef-add-def rcoef-sub-def rcoef-neg-def rcoef-double-def rcoef-mul-sqrt2-def r3-sub-def*; *elim disjE*; *simp*)

lemma *rstate-step-TB-FA*:
rstate (*False*, *A*) *v* ⟹ *rstate* (*True*, *B*) (*rletter-step* (*True*, *B*) *v*)
by (*simp add*: *rstate-def rletter-step-def rRz-inv-step-def rcoef-add-def rcoef-sub-def rcoef-neg-def rcoef-double-def rcoef-mul-sqrt2-def r3-sub-def*; *elim disjE*; *simp*)

lemma *rstate-step-TB-TA*:
rstate (*True*, *A*) *v* ⟹ *rstate* (*True*, *B*) (*rletter-step* (*True*, *B*) *v*)
by (*simp add*: *rstate-def rletter-step-def rRz-inv-step-def rcoef-add-def rcoef-sub-def rcoef-neg-def rcoef-double-def rcoef-mul-sqrt2-def r3-sub-def*; *elim disjE*; *simp*)

lemmas *rstate-step-cases* =
rstate-step-FA-FA rstate-step-FA-FB rstate-step-FA-TB
rstate-step-TA-TA rstate-step-TA-FB rstate-step-TA-TB
rstate-step-FB-FB rstate-step-FB-FA rstate-step-FB-TA
rstate-step-TB-TB rstate-step-TB-FA rstate-step-TB-TA

lemma *rstate-step*:
assumes *rstate q v* **and** ¬ *canceling p q*
shows *rstate p* (*rletter-step p v*)
using *assms*
by (*cases p*; *cases q*; *cases fst p*; *cases snd p*; *cases fst q*; *cases snd q*)
(*auto simp*: *canceling-def rstate-step-cases*)

lemma *rstate-nonzero*:
assumes *rstate p v*
shows *v* ≠ *rzero-vec*
using *assms*
by (*cases p*)
(*auto split*: *bool.splits gen2.splits simp*: *rstate-def rzero-vec-def*)

lemma *rword-step-Cons* [*simp*]:
rword-step (*p* # *w*) *v* = *rletter-step p* (*rword-step w v*)
by (*simp add*: *rword-step-def*)

lemma *zword-step-Cons* [*simp*]:
zword-step (*p* # *w*) *v* = *zletter-step p* (*zword-step w v*)
by (*simp add*: *zword-step-def*)

lemma *rword-witness-Cons-nonempty* [*simp*]:

```
  assumes w ≠ []
  shows rword-witness (p # w) = rword-witness w
  using assms by (cases w) (simp-all add: rword-witness-def)

lemma rword-step-witness-state:
  assumes w ∈ carrier F2 and w ≠ []
  shows rstate (hd w) (rword-step w (rword-witness w))
  using assms
proof (induct w)
  case Nil
  then show ?case
    by simp
next
  case (Cons p w)
  have tail: w ∈ carrier F2 and ncan-tail: w = [] ∨ ¬ canceling p (hd w)
    using Cons.prems(1) by (auto dest: F2-ConsD)
  show ?case
  proof (cases w)
    case Nil
    then show ?thesis
      using rstate-single[of p] by (simp add: rword-step-def rword-witness-def)
  next
    case (Cons q ws)
    have rstate q (rword-step (q # ws) (rword-witness (q # ws)))
      using Cons.hyps[OF tail] Cons by simp
    moreover have ¬ canceling p q
      using ncan-tail Cons by simp
    ultimately have rstate p (rletter-step p (rword-step (q # ws) (rword-witness
(q # ws))))
      by (rule rstate-step)
    then show ?thesis
      using Cons by simp
  qed
qed

definition r3-of-int :: int ⇒ r3 where
  r3-of-int n = (if n mod 3 = 0 then R0 else if n mod 3 = 1 then R1 else R2)

definition rcoef-of-z :: zsqrt2 ⇒ rcoef where
  rcoef-of-z p = (r3-of-int (fst p), r3-of-int (snd p))

definition rvec3-of-z :: zvec3 ⇒ rvec3 where
  rvec3-of-z v = (case v of (x, y, z) ⇒ (rcoef-of-z x, rcoef-of-z y, rcoef-of-z z))

lemma r3-of-int-add:
  r3-of-int (a + b) = r3-add (r3-of-int a) (r3-of-int b)
  by (auto simp: r3-of-int-def split: if-splits; presburger)

lemma r3-of-int-neg:
```

r3-of-int (− *a*) = *r3-neg* (*r3-of-int a*)
by (*auto simp*: *r3-of-int-def split*: *if-splits*; *presburger*)

lemma *r3-of-int-sub*:
r3-of-int (*a* − *b*) = *r3-add* (*r3-of-int a*) (*r3-neg* (*r3-of-int b*))
by (*auto simp*: *r3-of-int-def split*: *if-splits*; *presburger*)

lemma *r3-of-int-double*:
r3-of-int (*2* ∗ *a*) = *r3-double* (*r3-of-int a*)
by (*auto simp*: *r3-of-int-def split*: *if-splits*; *presburger*)

lemma *r3-of-int-triple*:
r3-of-int (*3* ∗ *a*) = *R0*
by (*simp add*: *r3-of-int-def*)

lemma *rcoef-of-z-simps* [*simp*]:
rcoef-of-z (*zsqrt2-add p q*) = *rcoef-add* (*rcoef-of-z p*) (*rcoef-of-z q*)
rcoef-of-z (*zsqrt2-neg p*) = *rcoef-neg* (*rcoef-of-z p*)
rcoef-of-z (*zsqrt2-sub p q*) = *rcoef-sub* (*rcoef-of-z p*) (*rcoef-of-z q*)
rcoef-of-z (*zsqrt2-scale 2 p*) = *rcoef-double* (*rcoef-of-z p*)
rcoef-of-z (*zsqrt2-scale 3 p*) = (*R0*, *R0*)
rcoef-of-z (*zsqrt2-mul-sqrt2 p*) = *rcoef-mul-sqrt2* (*rcoef-of-z p*)
by (*simp-all add*: *rcoef-of-z-def zsqrt2-add-def zsqrt2-neg-def zsqrt2-sub-def*
zsqrt2-scale-def zsqrt2-mul-sqrt2-def rcoef-add-def rcoef-neg-def rcoef-sub-def
rcoef-double-def rcoef-mul-sqrt2-def r3-of-int-add r3-of-int-neg
r3-of-int-sub r3-of-int-double r3-of-int-triple)

lemma *rvec3-of-z-basis* [*simp*]:
rvec3-of-z ze-x = *re-x*
rvec3-of-z ze-y = *re-y*
by (*simp-all add*: *rvec3-of-z-def rcoef-of-z-def ze-x-def ze-y-def re-x-def re-y-def*
r3-of-int-def)

lemma *rstep-of-z-simps* [*simp*]:
rvec3-of-z (*zRx-step v*) = *rRx-step* (*rvec3-of-z v*)
rvec3-of-z (*zRx-inv-step v*) = *rRx-inv-step* (*rvec3-of-z v*)
rvec3-of-z (*zRz-step v*) = *rRz-step* (*rvec3-of-z v*)
rvec3-of-z (*zRz-inv-step v*) = *rRz-inv-step* (*rvec3-of-z v*)
by (*cases v*; *simp add*: *rvec3-of-z-def zRx-step-def zRx-inv-step-def zRz-step-def*
zRz-inv-step-def rRx-step-def rRx-inv-step-def rRz-step-def rRz-inv-step-def)+

lemma *rletter-step-of-z* [*simp*]:
rvec3-of-z (*zletter-step p v*) = *rletter-step p* (*rvec3-of-z v*)
by (*cases p*)
(*auto split*: *bool.splits gen2.splits simp*: *zletter-step-def rletter-step-def*)

lemma *rword-step-of-z* [*simp*]:
rvec3-of-z (*zword-step w v*) = *rword-step w* (*rvec3-of-z v*)
proof (*induct w arbitrary*: *v*)

```
  case Nil
  then show ?case
    by (simp add: zword-step-def rword-step-def)
next
  case (Cons p w)
  then show ?case
    by simp
qed

lemma rword-witness-of-z [simp]:
  rvec3-of-z (zword-witness w) = rword-witness w
  by (cases snd (last w))
    (simp-all add: zword-witness-def rword-witness-def)

lemma r3-of-int-eq-R0-iff:
  r3-of-int n = R0 ⟷ 3 dvd n
  by (simp add: r3-of-int-def dvd-eq-mod-eq-0)

lemma rcoef-of-z-zero-iff:
  rcoef-of-z p = (R0, R0) ⟷ zsqrt2-div3 p
  by (cases p) (simp add: rcoef-of-z-def zsqrt2-div3-def r3-of-int-eq-R0-iff)

lemma rvec3-of-z-zero-iff:
  rvec3-of-z v = rzero-vec ⟷ zvec3-div3 v
 by (cases v) (simp add: rvec3-of-z-def rzero-vec-def zvec3-div3-def rcoef-of-z-zero-iff)

lemma zsqrt2-val-simps [simp]:
  zsqrt2-val (zsqrt2-add p q) = zsqrt2-val p + zsqrt2-val q
  zsqrt2-val (zsqrt2-neg p) = − zsqrt2-val p
  zsqrt2-val (zsqrt2-sub p q) = zsqrt2-val p − zsqrt2-val q
  zsqrt2-val (zsqrt2-scale n p) = of-int n ∗ zsqrt2-val p
 unfolding zsqrt2-val-def zsqrt2-add-def zsqrt2-neg-def zsqrt2-sub-def zsqrt2-scale-def
  by (simp-all add: algebra-simps)

lemma zsqrt2-mul-sqrt2-val [simp]:
  zsqrt2-val (zsqrt2-mul-sqrt2 p) = sqrt 2 ∗ zsqrt2-val p
  unfolding zsqrt2-val-def zsqrt2-mul-sqrt2-def
  by (simp add: algebra-simps)

lemma sqrt-prime-irrational-dev:
  fixes p :: nat
  assumes prime p
  shows sqrt p ∉ ℚ
proof
  from assms have p: p > 1
    by (rule prime-gt-1-nat)
  assume sqrt p ∈ ℚ
  then obtain m n :: nat
    where n: n ≠ 0
```

```
    and sqrt-rat: |sqrt p| = m / n
    and coprime m n
  by (rule Rats-abs-nat-div-natE)
have eq: m^2 = p * n^2
proof -
  from n sqrt-rat have m = |sqrt p| * n
    by simp
  then have m^2 = (sqrt p)^2 * n^2
    by (simp add: power-mult-distrib)
  also have (sqrt p)^2 = p
    by simp
  also have ... * n^2 = p * n^2
    by simp
  finally show ?thesis
    by linarith
qed
have p dvd m ∧ p dvd n
proof
  from eq have p dvd m^2 ..
  with assms show p dvd m
    by (rule prime-dvd-power)
  then obtain k where m = p * k ..
  with eq have p * n^2 = p^2 * k^2
    by algebra
  with p have n^2 = p * k^2
    by (simp add: power2-eq-square)
  then have p dvd n^2 ..
  with assms show p dvd n
    by (rule prime-dvd-power)
qed
then have p dvd gcd m n
  by simp
with ‹coprime m n› have p = 1
  by simp
with p show False
  by simp
qed

lemma sqrt-2-not-rat-dev: sqrt 2 ∉ ℚ
  using sqrt-prime-irrational-dev[of 2] by simp

lemma zsqrt2-val-eq-iff:
  zsqrt2-val p = zsqrt2-val q ⟷ p = q
proof
  assume eq: zsqrt2-val p = zsqrt2-val q
  obtain a b c d where defs: p = (a, b) q = (c, d)
    by (cases p; cases q) auto
  let ?A = a - c
  let ?B = b - d
```

```
  from eq defs have eq0: of-int ?A + of-int ?B * sqrt 2 = (0::real)
    by (simp add: zsqrt2-val-def algebra-simps)
  have ?B = 0
  proof (rule ccontr)
    assume ?B ≠ 0
    hence sqrt 2 = - of-int ?A / of-int ?B
      using eq0 by (simp add: field-simps)
    moreover have - of-int ?A / of-int ?B ∈ ℚ
      by simp
    ultimately have sqrt 2 ∈ ℚ
      by simp
    with sqrt-2-not-rat-dev show False
      by simp
  qed
  moreover with eq0 have ?A = 0
    by simp
  ultimately show p = q
    using defs by simp
qed simp

lemma zvec3-val-eq-iff:
  zvec3-val u = zvec3-val v ⟷ u = v
  by (cases u; cases v)
    (auto simp: zvec3-val-def vector-def vec-eq-iff forall-3 zsqrt2-val-eq-iff)

lemma zvec3-scale-val [simp]:
  zvec3-val (zvec3-scale n v) = scaleR (of-int n) (zvec3-val v)
  by (cases v)
    (simp add: zvec3-scale-def zvec3-val-def vector-def vec-eq-iff forall-3)

lemma sqrt8-eq-2-sqrt2: sqrt 8 = 2 * sqrt 2
  proof -
  have sqrt (8::real) = sqrt ((4::real) * 2)
    by simp
  also have ... = sqrt (4::real) * sqrt 2
    by (subst real-sqrt-mult) simp
  also have ... = 2 * sqrt 2
    by simp
  finally show ?thesis .
qed

lemma zletter-step-single-mod3:
  ¬ zvec3-div3 (zletter-step (False, A) ze-y)
  ¬ zvec3-div3 (zletter-step (True, A) ze-y)
  ¬ zvec3-div3 (zletter-step (False, B) ze-x)
  ¬ zvec3-div3 (zletter-step (True, B) ze-x)
 by (simp-all add: zletter-step-def zRx-step-def zRx-inv-step-def zRz-step-def zRz-inv-step-def
      ze-x-def ze-y-def zvec3-div3-def zsqrt2-div3-def zsqrt2-add-def zsqrt2-sub-def
      zsqrt2-neg-def zsqrt2-scale-def zsqrt2-mul-sqrt2-def)
```

lemma *zRx-step-val*:
zvec3-val (*zRx-step v*) = *scaleR 3* (*Rx* (*zvec3-val v*))
by (*cases v*)
(*simp add*: *zvec3-val-def zRx-step-def Rx-def Rx-mat-def rot-c-def rot-s-def*
sqrt8-eq-2-sqrt2 matrix-vector-mult-def vector-def vec-eq-iff sum-3 forall-3 algebra-simps)

lemma *zRx-inv-step-val*:
zvec3-val (*zRx-inv-step v*) = *scaleR 3* (*Rx-inv* (*zvec3-val v*))
by (*cases v*)
(*simp add*: *zvec3-val-def zRx-inv-step-def Rx-inv-def Rx-inv-mat-def rot-c-def rot-s-def*
sqrt8-eq-2-sqrt2 matrix-vector-mult-def vector-def vec-eq-iff sum-3 forall-3 algebra-simps)

lemma *zRz-step-val*:
zvec3-val (*zRz-step v*) = *scaleR 3* (*Rz* (*zvec3-val v*))
by (*cases v*)
(*simp add*: *zvec3-val-def zRz-step-def Rz-def Rz-mat-def rot-c-def rot-s-def*
sqrt8-eq-2-sqrt2 matrix-vector-mult-def vector-def vec-eq-iff sum-3 forall-3 algebra-simps)

lemma *zRz-inv-step-val*:
zvec3-val (*zRz-inv-step v*) = *scaleR 3* (*Rz-inv* (*zvec3-val v*))
by (*cases v*)
(*simp add*: *zvec3-val-def zRz-inv-step-def Rz-inv-def Rz-inv-mat-def rot-c-def rot-s-def*
sqrt8-eq-2-sqrt2 matrix-vector-mult-def vector-def vec-eq-iff sum-3 forall-3 algebra-simps)

lemma *zletter-step-val*:
zvec3-val (*zletter-step p v*) = *scaleR 3* (*letter-to-rot p* (*zvec3-val v*))
by (*cases p*)
(*auto split*: *bool.splits gen2.splits*
simp: *zletter-step-def letter-to-rot-def zRx-step-val zRx-inv-step-val*
zRz-step-val zRz-inv-step-val)

lemma *letter-to-rot-linear*: *linear* (*letter-to-rot p*)
unfolding *letter-to-rot-def Rx-def Rx-inv-def Rz-def Rz-inv-def*
by (*cases p*)
(*simp split*: *bool.splits gen2.splits add*: *matrix-vector-mul-linear*)

lemma *zword-step-val*:
zvec3-val (*zword-step w v*) =
scaleR ((*3*::*real*) ^ *length w*) (*sigma w* (*zvec3-val v*))
proof (*induct w arbitrary*: *v*)
case *Nil*
show *?case*

```
    by (simp add: zword-step-def)
next
  case (Cons p w)
  have lin: linear (letter-to-rot p)
    by (rule letter-to-rot-linear)
  have zvec3-val (zword-step (p # w) v) =
      zvec3-val (zletter-step p (zword-step w v))
    by (simp add: zword-step-def)
  also have ... = scaleR 3 (letter-to-rot p (zvec3-val (zword-step w v)))
    by (rule zletter-step-val)
  also have ... =
      scaleR 3 (letter-to-rot p
        (scaleR ((3::real) ^ length w) (sigma w (zvec3-val v))))
    using Cons.hyps by simp
  also have ... =
      scaleR ((3::real) ^ Suc (length w)) ((letter-to-rot p ∘ sigma w) (zvec3-val v))
    using linear.scaleR[OF lin, of ((3::real) ^ length w) sigma w (zvec3-val v)]
    by (simp add: scaleR-scaleR mult-ac)
  also have ... = scaleR ((3::real) ^ length (p # w)) (sigma (p # w) (zvec3-val
v))
    by simp
  finally show ?case .
qed

lemma zvec3-scale-power3-div3:
  assumes n > 0
  shows zvec3-div3 (zvec3-scale ((3::int) ^ n) v)
  using assms
  by (cases v)
    (auto simp: zvec3-div3-def zsqrt2-div3-def zvec3-scale-def zsqrt2-scale-def
      intro!: dvd-mult2)

lemma zword-step-identity-div3:
  assumes w ≠ [] and sigma w = id
  shows zvec3-div3 (zword-step w (zword-witness w))
proof −
  have len-pos: length w > 0
    using assms(1) by (cases w) auto
  have zvec3-val (zword-step w (zword-witness w)) =
      scaleR ((3::real) ^ length w) (zvec3-val (zword-witness w))
    using zword-step-val[of w zword-witness w] assms(2) by simp
  also have ... = zvec3-val (zvec3-scale ((3::int) ^ length w) (zword-witness w))
    by simp
  finally have zword-step w (zword-witness w) =
      zvec3-scale ((3::int) ^ length w) (zword-witness w)
    using zvec3-val-eq-iff by blast
  thus ?thesis
    using zvec3-scale-power3-div3[OF len-pos, of zword-witness w] by simp
qed
```

lemma *zword-step-witness-not-div3-if-nonempty-F2*:
assumes $w \in$ *carrier F2* **and** $w \neq []$
shows $\neg$ *zvec3-div3* (*zword-step w* (*zword-witness w*))
proof
assume *div3*: *zvec3-div3* (*zword-step w* (*zword-witness w*))
have *rstate* (*hd w*) (*rword-step w* (*rword-witness w*))
using *assms* **by** (*rule rword-step-witness-state*)
hence *rword-step w* (*rword-witness w*) $\neq$ *rzero-vec*
by (*rule rstate-nonzero*)
moreover from *div3* **have** *rword-step w* (*rword-witness w*) = *rzero-vec*
by (*simp add*: *rvec3-of-z-zero-iff*[*symmetric*])
ultimately show *False*
by *contradiction*
qed

The word interpretation is an injective homomorphism into $SO(3)$.

theorem *sigma-homomorphism*:
assumes *w1* $\in$ *carrier F2 w2* $\in$ *carrier F2*
shows *sigma* (*w1* $\otimes_{F2}$ *w2*) = *sigma w1* $\circ$ *sigma w2*
by (*simp add*: *F2-mult*)

theorem *sigma-nontrivial-word-not-id*:
assumes $w \in$ *carrier F2* **and** $w \neq []$
shows *sigma w* $\neq$ *id*
proof
assume *sigma w* = *id*
hence *zvec3-div3* (*zword-step w* (*zword-witness w*))
using *assms*(*2*) **by** (*intro zword-step-identity-div3*)
moreover have $\neg$ *zvec3-div3* (*zword-step w* (*zword-witness w*))
using *assms* **by** (*rule zword-step-witness-not-div3-if-nonempty-F2*)
ultimately show *False*
by *simp*
qed

lemma *sigma-injective-if-trivial-kernel*:
assumes *kernel*: $\bigwedge w.\ w \in$ *carrier F2* $\Longrightarrow$ $w \neq []$ $\Longrightarrow$ *sigma w* $\neq$ *id*
shows *inj-on sigma* (*carrier F2*)
proof (*rule inj-onI*)
interpret *F*: *group F2*
by (*rule F2-is-group*)
fix *w v*
assume *w*: $w \in$ *carrier F2* **and** *v*: $v \in$ *carrier F2* **and** *eq*: *sigma w* = *sigma v*
let *?iv* = inv_{F2} *v*
have *iv*: *?iv* $\in$ *carrier F2*
using *v* **by** (*rule F.inv-closed*)
have *prod*: *w* $\otimes_{F2}$ *?iv* $\in$ *carrier F2*
using *w iv* **by** (*rule F.m-closed*)
have *sigma* (*w* $\otimes_{F2}$ *?iv*) = *sigma w* $\circ$ *sigma ?iv*

```
    using w iv by (rule sigma-homomorphism)
  also have ... = sigma v ∘ sigma ?iv
    using eq by simp
  also have ... = sigma (v ⊗_F2 ?iv)
    using v iv by (rule sigma-homomorphism[symmetric])
  also have ... = sigma []
    using F.r-inv[OF v] by simp
  also have ... = id
    by simp
  finally have prod-id: sigma (w ⊗_F2 ?iv) = id .
  have prod-one: w ⊗_F2 ?iv = []
  proof (rule ccontr)
    assume w ⊗_F2 ?iv ≠ []
    with kernel[OF prod] prod-id show False
      by simp
  qed
  have w = w ⊗_F2 1_F2
    using w by simp
  moreover have ?iv ⊗_F2 v = 1_F2
    using v by (rule F.l-inv)
  ultimately have w = w ⊗_F2 (?iv ⊗_F2 v)
    by simp
  also have ... = (w ⊗_F2 ?iv) ⊗_F2 v
    using w iv v by (simp add: F.m-assoc)
  also have ... = v
    using prod-one v by simp
  finally show w = v .
qed

theorem sigma-injective-on-F2:
  inj-on sigma (carrier F2)
  by (rule sigma-injective-if-trivial-kernel) (rule sigma-nontrivial-word-not-id)

theorem sigma-image-in-SO3:
  assumes w ∈ carrier F2
  shows sigma w ∈ SO3
  by (rule sigma-in-SO3)
```

These three theorems together establish that *sigma* is a group monomorphism from F_2 to SO(3) – i.e., the image is a copy of F_2 inside SO(3). This supplies the free subgroup used in the geometric part of the Banach-Tarski theorem.

```
end


theory Hausdorff-Paradox
  imports Free-Rotations-SO3
begin
```

23 The bad set of fixed points

definition *fixed-point-set* :: (*real^3* ⇒ *real^3*) ⇒ (*real^3*) *set* **where**
fixed-point-set f = {*x* ∈ *sphere2*. *f x* = *x*}

definition *bad-set-D* :: (*real^3*) *set* **where**
bad-set-D = (⋃ *w* ∈ *carrier F2* − {[]}. *fixed-point-set* (*sigma w*))

lemma *carrier-F2-countable*: *countable* (*carrier F2*)
proof (*rule countable-subset*)
show *carrier F2* ⊆ (*UNIV* :: ((*bool* × *gen2*) *list*) *set*)
by *simp*
show *countable* (*UNIV* :: ((*bool* × *gen2*) *list*) *set*)
proof −
have *gen2-finite*: *finite* (*UNIV* :: *gen2 set*)
using *Gen2-finite Gen2-UNIV* **by** *simp*
have *alphabet-finite*: *finite* ((*UNIV* :: *bool set*) × (*UNIV* :: *gen2 set*))
by (*rule finite-cartesian-product*) (*simp-all add*: *gen2-finite*)
have *alphabet-countable*: *countable* (*UNIV* :: (*bool* × *gen2*) *set*)
by (*rule countable-finite*) (*use alphabet-finite* **in** ‹*simp only*: *UNIV-Times-UNIV*›)
have *countable* (*lists* (*UNIV* :: (*bool* × *gen2*) *set*))
by (*rule countable-lists*) (*rule alphabet-countable*)
then show *?thesis* **by** *simp*
qed
qed

lemma *bad-set-D-index-countable*: *countable* (*carrier F2* − {[]})
using *carrier-F2-countable* **by** *simp*

lemma *fixed-point-set-subset-sphere2*: *fixed-point-set f* ⊆ *sphere2*
by (*simp add*: *fixed-point-set-def*)

lemma *SO3-fixed-cross*:
assumes *f* ∈ *SO3* **and** *f x* = *x* **and** *f y* = *y*
shows *f* (*cross3 x y*) = *cross3 x y*
proof −
have *f-orth*: *orthogonal-transformation f*
using *assms*(*1*) **unfolding** *SO3-def rotation-def* **by** *auto*
have *f-det*: *det* (*matrix f*) = *1*
using *assms*(*1*) **unfolding** *SO3-def rotation-def* **by** *auto*
have *cross3* (*f x*) (*f y*) = *det* (*matrix f*) ∗$_R$ *f* (*cross3 x y*)
using *cross-orthogonal-transformation*[*OF f-orth*, *of x y*] .
with *assms*(*2*,*3*) *f-det* **show** *?thesis*
by *simp*
qed

lemma *SO3-fixed-cross-nonzero-imp-id*:
assumes *f* ∈ *SO3* **and** *f x* = *x* **and** *f y* = *y* **and** *cross3 x y* ≠ *0*
shows *f* = *id*

proof −
let *?n* = *cross3 x y*
let *?u* = *y* − ((*x* · *y*) / (*x* · *x*)) ∗$_R$ *x*
let *?S* = {*x*, *?u*, *?n*}
have *f-orth*: *orthogonal-transformation f*
using *assms*(*1*) **unfolding** *SO3-def rotation-def* **by** *auto*
have *f-lin*: *linear f*
by (*rule orthogonal-transformation-linear*[*OF f-orth*])
have *x-ne*: *x* ≠ *0*
using *assms*(*4*) **by** *auto*
have *xx-ne*: *x* · *x* ≠ *0*
using *x-ne* **by** *simp*
have *n-fix*: *f ?n* = *?n*
by (*rule SO3-fixed-cross*[*OF assms*(*1*,*2*,*3*)])
have *u-fix*: *f ?u* = *?u*
using *assms*(*2*,*3*) *f-lin*
by (*simp add*: *linear-diff linear-scale*)
have *cross-x-u*: *cross3 x ?u* = *?n*
by (*simp add*: *Cross3.right-diff-distrib cross-mult-right*)
have *u-ne*: *?u* ≠ *0*
proof
assume *?u* = *0*
hence *cross3 x ?u* = *0*
by *simp*
with *cross-x-u assms*(*4*) **show** *False*
by *simp*
qed
have *x-u-orth*: *orthogonal x ?u*
using *xx-ne* **by** (*simp add*: *orthogonal-def inner-diff-right inner-commute*)
have *n-x-orth*: *orthogonal ?n x*
by (*rule orthogonal-cross*)
have *n-y-orth*: *orthogonal ?n y*
by (*rule orthogonal-cross*)
have *n-u-orth*: *orthogonal ?n ?u*
using *n-x-orth n-y-orth*
by (*simp add*: *orthogonal-def inner-diff-right*)
have *pairwise-S*: *pairwise orthogonal ?S*
using *x-u-orth n-x-orth n-u-orth*
by (*auto simp*: *pairwise-def orthogonal-commute*)
have *zero-notin-S*: *0* ∉ *?S*
using *assms*(*4*) *x-ne u-ne* **by** *auto*
have *ind-S*: *independent ?S*
by (*rule pairwise-orthogonal-independent*[*OF pairwise-S zero-notin-S*])
have *x-neq-u*: *x* ≠ *?u*
proof
assume *x* = *?u*
hence *orthogonal x x*
using *x-u-orth* **by** *simp*
with *x-ne* **show** *False*

```
    by (simp add: orthogonal-def)
  qed
  have x-neq-n: x ≠ ?n
  proof
    assume x = ?n
    hence orthogonal x x
      using n-x-orth by (simp add: orthogonal-commute)
    with x-ne show False
      by (simp add: orthogonal-def)
  qed
  have u-neq-n: ?u ≠ ?n
  proof
    assume ?u = ?n
    hence orthogonal ?u ?u
      using n-u-orth by (simp add: orthogonal-commute)
    with u-ne show False
      by (simp add: orthogonal-def)
  qed
  have card-S: card ?S = CARD(3)
    using x-neq-u x-neq-n u-neq-n by simp
  have dim-S: dim ?S = CARD(3)
    using ind-S card-S by (simp add: dim-eq-card-independent)
  have span-S: span ?S = UNIV
  proof (rule iffD1[OF dim-eq-full])
    show dim ?S = DIM(real^3)
      using dim-S by simp
  qed
  have fixed-on-S: ⋀z. z ∈ ?S ⟹ f z = id z
    using assms(2) u-fix n-fix by auto
  have fixed-on-span: ⋀z. z ∈ span ?S ⟹ f z = id z
    by (rule linear-eq-on-span[OF f-lin linear-id fixed-on-S])
  show ?thesis
  proof
    fix z
    have z ∈ span ?S
      using span-S by simp
    thus f z = id z
      by (rule fixed-on-span)
  qed
qed

lemma nonidentity-SO3-fixed-points-cross-zero:
  assumes f ∈ SO3 and f ≠ id and x ∈ fixed-point-set f and y ∈ fixed-point-set
f
  shows cross3 x y = 0
proof (rule ccontr)
  assume cross-ne: cross3 x y ≠ 0
  have fx: f x = x and fy: f y = y
    using assms(3,4) by (simp-all add: fixed-point-set-def)
```

```
  have f = id
    by (rule SO3-fixed-cross-nonzero-imp-id[OF assms(1) fx fy cross-ne])
  with assms(2) show False
    by simp
qed

lemma collinear-sphere2-two-points:
  assumes a ∈ sphere2 and y ∈ sphere2 and cross3 a y = 0
  shows y = a ∨ y = − a
proof −
  have cross-col: cross3 a y = 0 ⟷ collinear {0, a, y}
    by (rule cross-eq-0)
  have norm-a: norm a = 1
    using assms(1) by (simp add: sphere2-def)
  have norm-y: norm y = 1
    using assms(2) by (simp add: sphere2-def)
  have a-ne: a ≠ 0
  proof
    assume a = 0
    with norm-a show False
      by simp
  qed
  have y-ne: y ≠ 0
  proof
    assume y = 0
    with norm-y show False
      by simp
  qed
  have col: collinear {0, a, y}
    using assms(3) cross-col by blast
  have cases: a = 0 ∨ y = 0 ∨ (∃ c. y = c ∗R a)
    using iffD1[OF collinear-lemma[of a y] col] .
  then obtain c where y-eq: y = c ∗R a
    using a-ne y-ne by blast
  have |c| = 1
    using norm-y by (simp add: y-eq norm-a)
  hence c-cases: c = 1 ∨ c = −1
  proof (cases c ≥ 0)
    case True
    with ‹|c| = 1› have c = 1
      by simp
    thus ?thesis ..
  next
    case False
    with ‹|c| = 1› have c = −1
      by simp
    thus ?thesis ..
  qed
  show ?thesis
```

```
    using c-cases y-eq by auto
qed

lemma bad-set-D-countable-if-fixed-point-sets-countable:
  assumes ⋀w. w ∈ carrier F2 − {[]} ⟹ countable (fixed-point-set (sigma w))
  shows countable bad-set-D
proof −
  have countable (⋃ w ∈ carrier F2 − {[]}. fixed-point-set (sigma w))
    by (rule countable-UN) (use bad-set-D-index-countable assms in auto)
  thus ?thesis
    by (simp add: bad-set-D-def)
qed

lemma nonidentity-SO3-fixed-point-set-countable:
  assumes f ∈ SO3 and f ≠ id
  shows countable (fixed-point-set f)
proof (cases fixed-point-set f = {})
  case True
  then show ?thesis by simp
next
  case False
  then obtain a where a: a ∈ fixed-point-set f
    by auto
  hence a-sphere: a ∈ sphere2
    by (simp add: fixed-point-set-def)
  have subset-two: fixed-point-set f ⊆ {a, − a}
  proof
    fix y
    assume y: y ∈ fixed-point-set f
    hence y-sphere: y ∈ sphere2
      by (simp add: fixed-point-set-def)
    have cross-zero: cross3 a y = 0
      by (rule nonidentity-SO3-fixed-points-cross-zero[OF assms a y])
    have y = a ∨ y = − a
      by (rule collinear-sphere2-two-points[OF a-sphere y-sphere cross-zero])
    thus y ∈ {a, − a}
      by simp
  qed
  have finite (fixed-point-set f)
    by (rule finite-subset[OF subset-two]) simp
  thus ?thesis
    by (rule countable-finite)
qed

lemma sigma-fixed-point-set-countable:
  assumes w ∈ carrier F2 − {[]}
  shows countable (fixed-point-set (sigma w))
proof −
  have sigma w ∈ SO3
```

using *assms sigma-image-in-SO3* **by** *auto*
moreover have *sigma w* $\neq$ *id*
using *assms sigma-nontrivial-word-not-id* **by** *auto*
ultimately show *?thesis*
by (*rule nonidentity-SO3-fixed-point-set-countable*)
qed

lemma *bad-set-D-countable*: *countable bad-set-D*
by (*rule bad-set-D-countable-if-fixed-point-sets-countable*)
(*rule sigma-fixed-point-set-countable*)

lemma *bad-set-D-subset*: *bad-set-D* $\subseteq$ *sphere2*
unfolding *bad-set-D-def fixed-point-set-def* **by** *auto*

lemma *sphere2-minus-bad-subset*: *sphere2* $-$ *bad-set-D* $\subseteq$ *sphere2*
by *auto*

interpretation *sigma-sphere-action*:
group-action carrier F2 [] (λ*w1 w2*. *w1* $\otimes_{F2}$ *w2*) λ*w x*. *sigma w x sphere2*
proof
show [] $\in$ *carrier F2*
by *simp*
show $\bigwedge$*g h*. *g* $\in$ *carrier F2* $\Longrightarrow$ *h* $\in$ *carrier F2* $\Longrightarrow$ *g* $\otimes_{F2}$ *h* $\in$ *carrier F2*
using *group.subgroup-self group-def monoid.m-closed free-group-is-group* **by**
blast
show $\bigwedge$*g x*. *g* $\in$ *carrier F2* $\Longrightarrow$ *x* $\in$ *sphere2* $\Longrightarrow$ *sigma g x* $\in$ *sphere2*
using *sigma-preserves-sphere2* **by** *blast*
show $\bigwedge$*x*. *x* $\in$ *sphere2* $\Longrightarrow$ *sigma* [] *x* = *x*
by *simp*
show $\bigwedge$*g h x*. *g* $\in$ *carrier F2* $\Longrightarrow$ *h* $\in$ *carrier F2* $\Longrightarrow$ *x* $\in$ *sphere2* $\Longrightarrow$
sigma (*g* $\otimes_{F2}$ *h*) *x* = *sigma g* (*sigma h x*)
using *sigma-homomorphism* **by** (*simp add*: *o-def*)
qed

lemma *F2-conjugate-nontrivial*:
assumes *w* $\in$ *carrier F2 v* $\in$ *carrier F2 v* $\neq$ []
shows (*inv*$_{F2}$ *w* $\otimes_{F2}$ *v*) $\otimes_{F2}$ *w* $\neq$ []
proof
interpret *G*: *group F2*
by (*rule free-group-is-group*)
assume *conj-eq*: (*inv*$_{F2}$ *w* $\otimes_{F2}$ *v*) $\otimes_{F2}$ *w* = []
have *inv-closed*: *inv*$_{F2}$ *w* $\in$ *carrier F2*
using *assms*(*1*) **by** (*rule G.inv-closed*)
have *vw-closed*: *v* $\otimes_{F2}$ *w* $\in$ *carrier F2*
by (*rule G.m-closed*) (*use assms* **in** *simp-all*)
have *assoc-eq*:
inv$_{F2}$ *w* $\otimes_{F2}$ (*v* $\otimes_{F2}$ *w*) =
(*inv*$_{F2}$ *w* $\otimes_{F2}$ *v*) $\otimes_{F2}$ *w*
using *inv-closed assms* **by** (*simp add*: *G.m-assoc*)

```
  have eq1: inv_F2 w ⊗_F2 (v ⊗_F2 w) = []
    using conj-eq assoc-eq by simp
  have v ⊗_F2 w = w ⊗_F2 []
    using G.inv-solve-left'[OF G.one-closed assms(1) vw-closed] eq1 by simp
  hence v ⊗_F2 w = w
    using assms by simp
  hence v = []
    using assms by simp
  with assms(3) show False by simp
qed

lemma sigma-inverse-left:
  assumes w ∈ carrier F2
  shows sigma (inv_F2 w) (sigma w x) = x
proof −
  interpret G: group F2
    by (rule free-group-is-group)
  have inv-closed: inv_F2 w ∈ carrier F2
    using assms by (rule G.inv-closed)
  have sigma (inv_F2 w ⊗_F2 w) x = sigma (inv_F2 w) (sigma w x)
    using sigma-homomorphism[OF inv-closed assms]
    by (simp add: o-def)
  moreover have inv_F2 w ⊗_F2 w = []
    using G.l-inv[OF assms] by simp
  ultimately show ?thesis
    by simp
qed

lemma sigma-conjugate-fixed-pullback:
  assumes w ∈ carrier F2
    and v ∈ carrier F2
    and sigma v (sigma w x) = sigma w x
  shows sigma ((inv_F2 w ⊗_F2 v) ⊗_F2 w) x = x
proof −
  interpret G: group F2
    by (rule free-group-is-group)
  have inv-closed: inv_F2 w ∈ carrier F2
    using assms(1) by (rule G.inv-closed)
  have iv-closed: inv_F2 w ⊗_F2 v ∈ carrier F2
    using inv-closed assms(2) by (rule G.m-closed)
  have sigma ((inv_F2 w ⊗_F2 v) ⊗_F2 w) x =
      sigma (inv_F2 w ⊗_F2 v) (sigma w x)
    using sigma-homomorphism[OF iv-closed assms(1)] by (simp add: o-def)
  also have ... = sigma (inv_F2 w) (sigma v (sigma w x))
    using sigma-homomorphism[OF inv-closed assms(2)] by (simp add: o-def)
  also have ... = sigma (inv_F2 w) (sigma w x)
    using assms(3) by simp
  also have ... = x
    using sigma-inverse-left[OF assms(1)] by simp
```

```
  finally show ?thesis .
qed

lemma sigma-preimage-bad-set:
  assumes w ∈ carrier F2
    and x ∈ sphere2
    and sigma w x ∈ bad-set-D
  shows x ∈ bad-set-D
proof −
  from assms(3) obtain v where v-carrier: v ∈ carrier F2
    and v-ne: v ≠ []
    and fixed: sigma v (sigma w x) = sigma w x
    unfolding bad-set-D-def fixed-point-set-def by auto
  define u where u = (inv_F2 w ⊗_F2 v) ⊗_F2 w
  have u-carrier: u ∈ carrier F2
  proof −
    interpret G: group F2
      by (rule free-group-is-group)
    have inv-closed: inv_F2 w ∈ carrier F2
      using assms(1) by (rule G.inv-closed)
    have inv_F2 w ⊗_F2 v ∈ carrier F2
      using inv-closed v-carrier by (rule G.m-closed)
    thus ?thesis
      unfolding u-def using assms(1) by (rule G.m-closed)
  qed
  have u-ne: u ≠ []
   using F2-conjugate-nontrivial[OF assms(1) v-carrier v-ne] by (simp add: u-def)
  have sigma u x = x
     using sigma-conjugate-fixed-pullback[OF assms(1) v-carrier fixed] by (simp
add: u-def)
  hence x ∈ fixed-point-set (sigma u)
    using assms(2) by (simp add: fixed-point-set-def)
  with u-carrier u-ne show ?thesis
    unfolding bad-set-D-def by blast
qed

lemma sigma-preserves-sphere2-minus-bad-set:
  assumes w ∈ carrier F2
    and x ∈ sphere2 − bad-set-D
  shows sigma w x ∈ sphere2 − bad-set-D
proof
  show sigma w x ∈ sphere2
    using assms sigma-preserves-sphere2 by blast
  show sigma w x ∉ bad-set-D
  proof
    assume sigma w x ∈ bad-set-D
    hence x ∈ bad-set-D
      using assms sigma-preimage-bad-set by auto
    with assms(2) show False by simp
```

qed
qed

interpretation *sigma-sphere-minus-bad-action*:
group-action carrier F2 [] (λ*w1 w2*. *w1* $\otimes_{F2}$ *w2*) λ*w x*. *sigma w x*
sphere2 − *bad-set-D*
proof
show [] ∈ *carrier F2*
by *simp*
show $\bigwedge$*g h*. *g* ∈ *carrier F2* ⟹ *h* ∈ *carrier F2* ⟹ *g* $\otimes_{F2}$ *h* ∈ *carrier F2*
using *group.subgroup-self group-def monoid.m-closed free-group-is-group* **by** *blast*
show $\bigwedge$*g x*. *g* ∈ *carrier F2* ⟹ *x* ∈ *sphere2* − *bad-set-D* ⟹
sigma g x ∈ *sphere2* − *bad-set-D*
by (*rule sigma-preserves-sphere2-minus-bad-set*)
show $\bigwedge$*x*. *x* ∈ *sphere2* − *bad-set-D* ⟹ *sigma* [] *x* = *x*
by *simp*
show $\bigwedge$*g h x*. *g* ∈ *carrier F2* ⟹ *h* ∈ *carrier F2* ⟹ *x* ∈ *sphere2* − *bad-set-D* ⟹
sigma (*g* $\otimes_{F2}$ *h*) *x* = *sigma g* (*sigma h x*)
using *sigma-homomorphism* **by** (*simp add*: *o-def*)
qed

24 Free action of F_2 on $S^2 \setminus D$

By construction, every point of $S^2 \setminus D$ avoids the fixed-point set of every non-trivial group element, hence is moved by every non-trivial *sigma w*.

The action is free away from the fixed-point bad set.

lemma *sigma-action-free-off-D*:
assumes *x* ∈ *sphere2* − *bad-set-D*
and *w* ∈ *carrier F2*
and *sigma w x* = *x*
shows *w* = []
proof (*rule ccontr*)
assume *w-ne*: *w* ≠ []
from *assms*(*1*) **have** *x-sphere*: *x* ∈ *sphere2* **and** *x-notbad*: *x* ∉ *bad-set-D*
by *auto*
from *x-sphere assms*(*3*) **have** *x* ∈ *fixed-point-set* (*sigma w*)
by (*simp add*: *fixed-point-set-def*)
with *assms*(*2*) *w-ne* **have** *x* ∈ *bad-set-D*
unfolding *bad-set-D-def* **by** *blast*
with *x-notbad* **show** *False* **..**
qed

lemma *sigma-sphere-minus-bad-action-free*:
sigma-sphere-minus-bad-action.free-on (*sphere2* − *bad-set-D*)
unfolding *sigma-sphere-minus-bad-action.free-on-def*

using *sigma-action-free-off-D* **by** *auto*

lemma *sigma-orbit-eqI*:
assumes *x*: $x \in$ *sphere2* $-$ *bad-set-D*
and *y*: $y \in$ *sphere2* $-$ *bad-set-D*
and *g*: $g \in$ *carrier F2*
and *y-eq*: $y =$ *sigma g x*
shows *sigma-sphere-minus-bad-action.orbit y* =
sigma-sphere-minus-bad-action.orbit x
proof
interpret *G*: *group F2*
by (*rule free-group-is-group*)
show *sigma-sphere-minus-bad-action.orbit y* $\subseteq$
sigma-sphere-minus-bad-action.orbit x
proof
fix *z*
assume $z \in$ *sigma-sphere-minus-bad-action.orbit y*
then obtain *h* **where** *h*: $h \in$ *carrier F2* **and** *z*: $z =$ *sigma h y*
unfolding *sigma-sphere-minus-bad-action.orbit-def* **by** *blast*
have *hg*: $h \otimes_{F2} g \in$ *carrier F2*
by (*rule G.m-closed*[*OF h g*])
have $z =$ *sigma* $(h \otimes_{F2} g)$ *x*
using *z y-eq sigma-homomorphism*[*OF h g*] **by** (*simp add*: *o-def*)
thus $z \in$ *sigma-sphere-minus-bad-action.orbit x*
unfolding *sigma-sphere-minus-bad-action.orbit-def* **using** *hg* **by** *blast*
qed
show *sigma-sphere-minus-bad-action.orbit x* $\subseteq$
sigma-sphere-minus-bad-action.orbit y
proof
fix *z*
assume $z \in$ *sigma-sphere-minus-bad-action.orbit x*
then obtain *h* **where** *h*: $h \in$ *carrier F2* **and** *z*: $z =$ *sigma h x*
unfolding *sigma-sphere-minus-bad-action.orbit-def* **by** *blast*
have *invg*: $inv_{F2}\ g \in$ *carrier F2*
by (*rule G.inv-closed*[*OF g*])
have *h-invg*: $h \otimes_{F2} inv_{F2}\ g \in$ *carrier F2*
by (*rule G.m-closed*[*OF h invg*])
have *x-eq*: $x =$ *sigma* $(inv_{F2}\ g)$ *y*
using *y-eq sigma-inverse-left*[*OF g*] **by** *simp*
have $z =$ *sigma* $(h \otimes_{F2} inv_{F2}\ g)$ *y*
using *z x-eq sigma-homomorphism*[*OF h invg*] **by** (*simp add*: *o-def*)
thus $z \in$ *sigma-sphere-minus-bad-action.orbit y*
unfolding *sigma-sphere-minus-bad-action.orbit-def* **using** *h-invg* **by** *blast*
qed
qed

lemma *sigma-orbit-eq-if-common-point*:
assumes *x*: $x \in$ *sphere2* $-$ *bad-set-D*
and *y*: $y \in$ *sphere2* $-$ *bad-set-D*

and *z*: $z \in$ *sigma-sphere-minus-bad-action.orbit x*
and *z'*: $z \in$ *sigma-sphere-minus-bad-action.orbit y*
shows *sigma-sphere-minus-bad-action.orbit x* =
sigma-sphere-minus-bad-action.orbit y
proof −
from *z* **obtain** *g* **where** *g*: $g \in$ *carrier F2* **and** *z-eq*: $z =$ *sigma g x*
unfolding *sigma-sphere-minus-bad-action.orbit-def* **by** *blast*
have *zX*: $z \in$ *sphere2* − *bad-set-D*
using *sigma-preserves-sphere2-minus-bad-set*[*OF g x*] *z-eq* **by** *simp*
have *orbit-z-x*: *sigma-sphere-minus-bad-action.orbit z* =
sigma-sphere-minus-bad-action.orbit x
using *sigma-orbit-eqI*[*OF x zX g z-eq*] **.**
from *z'* **obtain** *h* **where** *h*: $h \in$ *carrier F2* **and** *z-eq-y*: $z =$ *sigma h y*
unfolding *sigma-sphere-minus-bad-action.orbit-def* **by** *blast*
have *orbit-z-y*: *sigma-sphere-minus-bad-action.orbit z* =
sigma-sphere-minus-bad-action.orbit y
using *sigma-orbit-eqI*[*OF y zX h z-eq-y*] **.**
show *?thesis*
using *orbit-z-x orbit-z-y* **by** *simp*
qed

25 Transporting the free-group paradox to $S^2 \setminus D$

definition *hausdorff-rep* :: *real^3* ⇒ *real^3* **where**
hausdorff-rep x = (*SOME y*. $y \in$ *sigma-sphere-minus-bad-action.orbit x*)

definition *hausdorff-lift* :: ((*bool* × *gen2*) *list*) *set* ⇒ (*real^3*) *set* **where**
hausdorff-lift u =
{*sigma g* (*hausdorff-rep x*) | *g x*. $g \in u \wedge x \in$ *sphere2* − *bad-set-D*}

definition *hausdorff-sigma-image-set* ::
(*bool* × *gen2*) *list* ⇒ (*real^3*) *set* ⇒ (*real^3*) *set* **where**
hausdorff-sigma-image-set g u = *sigma g* ‘ *u*

lemma *map2-hausdorff-sigma-image-set-eq*:
map2 sigma-sphere-minus-bad-action.image-set gs As =
map2 hausdorff-sigma-image-set gs As
by (*induction gs arbitrary*: *As*)
(*auto simp*: *hausdorff-sigma-image-set-def sigma-sphere-minus-bad-action.image-set-def*
split: *list.splits*)

lemma *hausdorff-rep-in-orbit*:
assumes $x \in$ *sphere2* − *bad-set-D*
shows *hausdorff-rep x* ∈ *sigma-sphere-minus-bad-action.orbit x*
unfolding *hausdorff-rep-def*
by (*rule someI-ex*) (*use sigma-sphere-minus-bad-action.orbit-self*[*OF assms*] **in**
blast)

lemma *hausdorff-rep-in-X*:

assumes $x \in$ *sphere2* $-$ *bad-set-D*
shows *hausdorff-rep* $x \in$ *sphere2* $-$ *bad-set-D*
proof $-$
from *hausdorff-rep-in-orbit*[*OF assms*] **obtain** g
where g: $g \in$ *carrier F2* **and** *rep-eq*: *hausdorff-rep* $x =$ *sigma* g x
unfolding *sigma-sphere-minus-bad-action.orbit-def* **by** *blast*
show *?thesis*
using *sigma-preserves-sphere2-minus-bad-set*[*OF* g *assms*] *rep-eq* **by** *simp*
qed

lemma *hausdorff-rep-eq-if-orbit-eq*:
assumes *sigma-sphere-minus-bad-action.orbit* $x =$
sigma-sphere-minus-bad-action.orbit y
shows *hausdorff-rep* $x =$ *hausdorff-rep* y
using *assms* **by** (*simp add*: *hausdorff-rep-def*)

lemma *hausdorff-rep-idem*:
assumes x: $x \in$ *sphere2* $-$ *bad-set-D*
shows *hausdorff-rep* (*hausdorff-rep* x) = *hausdorff-rep* x
proof $-$
let *?r* = *hausdorff-rep* x
have *rX*: *?r* $\in$ *sphere2* $-$ *bad-set-D*
by (*rule hausdorff-rep-in-X*[*OF* x])
have *r-orbit-x*: *?r* $\in$ *sigma-sphere-minus-bad-action.orbit* x
by (*rule hausdorff-rep-in-orbit*[*OF* x])
have *r-orbit-r*: *?r* $\in$ *sigma-sphere-minus-bad-action.orbit ?r*
by (*rule sigma-sphere-minus-bad-action.orbit-self*[*OF rX*])
have *sigma-sphere-minus-bad-action.orbit ?r* =
sigma-sphere-minus-bad-action.orbit x
by (*rule sigma-orbit-eq-if-common-point*[*OF rX x r-orbit-r r-orbit-x*])
hence *hausdorff-rep ?r* = *hausdorff-rep* x
by (*rule hausdorff-rep-eq-if-orbit-eq*)
thus *?thesis*
by *simp*
qed

lemma *hausdorff-lift-subset-X*:
assumes $u \subseteq$ *carrier F2*
shows *hausdorff-lift* $u \subseteq$ *sphere2* $-$ *bad-set-D*
proof
fix y
assume $y \in$ *hausdorff-lift* u
then obtain g x **where** *gu*: $g \in u$ **and** x: $x \in$ *sphere2* $-$ *bad-set-D*
and *y-eq*: $y =$ *sigma* g (*hausdorff-rep* x)
unfolding *hausdorff-lift-def* **by** *blast*
have g: $g \in$ *carrier F2*
using *assms gu* **by** *auto*
have *repX*: *hausdorff-rep* $x \in$ *sphere2* $-$ *bad-set-D*
by (*rule hausdorff-rep-in-X*[*OF* x])

show $y \in$ *sphere2* $-$ *bad-set-D*
using *sigma-preserves-sphere2-minus-bad-set*[*OF g repX*] *y-eq* **by** *simp*
qed

lemma *F2-act-image-set-subset-carrier*:
assumes $g \in$ *carrier F2* **and** $u \subseteq$ *carrier F2*
shows *F2-act.image-set g u* $\subseteq$ *carrier F2*
unfolding *F2-act.image-set-def*
using *assms* **by** *auto*

lemma *sigma-free-same-point-eq*:
assumes *x*: $x \in$ *sphere2* $-$ *bad-set-D*
and *p*: $p \in$ *carrier F2*
and *q*: $q \in$ *carrier F2*
and *eq*: *sigma p x* $=$ *sigma q x*
shows $p = q$
proof $-$
interpret *G*: *group F2*
by (*rule free-group-is-group*)
have *invp*: $inv_{F2}\ p \in$ *carrier F2*
by (*rule G.inv-closed*[*OF p*])
have *invpq*: $inv_{F2}\ p \otimes_{F2} q \in$ *carrier F2*
by (*rule G.m-closed*[*OF invp q*])
have *fixed*: *sigma* $(inv_{F2}\ p \otimes_{F2} q)\ x = x$
proof $-$
have *sigma* $(inv_{F2}\ p \otimes_{F2} q)\ x =$
sigma $(inv_{F2}\ p)$ (*sigma q x*)
using *sigma-homomorphism*[*OF invp q*] **by** (*simp add*: *o-def*)
also have $\ldots =$ *sigma* $(inv_{F2}\ p)$ (*sigma p x*)
using *eq* **by** *simp*
also have $\ldots = x$
using *sigma-inverse-left*[*OF p*] **by** *simp*
finally show *?thesis* **.**
qed
have *invpq-unit*: $inv_{F2}\ p \otimes_{F2} q = []$
by (*rule sigma-action-free-off-D*[*OF x invpq fixed*])
have $q = p \otimes_{F2} []$
using *G.inv-solve-left'*[*OF G.one-closed p q*] *invpq-unit* **by** *simp*
thus *?thesis*
using *p* **by** *simp*
qed

lemma *hausdorff-lift-disjoint*:
assumes *disj*: $u \cap v = \{\}$
and *u*: $u \subseteq$ *carrier F2*
and *v*: $v \subseteq$ *carrier F2*
shows *hausdorff-lift u* $\cap$ *hausdorff-lift v* $= \{\}$
proof
show *hausdorff-lift u* $\cap$ *hausdorff-lift v* $\subseteq \{\}$

```
  proof
    fix z
    assume z: z ∈ hausdorff-lift u ∩ hausdorff-lift v
    then obtain p x where pu: p ∈ u and x: x ∈ sphere2 − bad-set-D
      and z-p: z = sigma p (hausdorff-rep x)
      unfolding hausdorff-lift-def by blast
    from z obtain q y where qv: q ∈ v and y: y ∈ sphere2 − bad-set-D
      and z-q: z = sigma q (hausdorff-rep y)
      unfolding hausdorff-lift-def by blast
    let ?rx = hausdorff-rep x
    let ?ry = hausdorff-rep y
    have p: p ∈ carrier F2
      using u pu by auto
    have q: q ∈ carrier F2
      using v qv by auto
    have rxX: ?rx ∈ sphere2 − bad-set-D
      by (rule hausdorff-rep-in-X[OF x])
    have ryX: ?ry ∈ sphere2 − bad-set-D
      by (rule hausdorff-rep-in-X[OF y])
    have z-orbit-rx: z ∈ sigma-sphere-minus-bad-action.orbit ?rx
      unfolding sigma-sphere-minus-bad-action.orbit-def
      using p z-p by blast
    have z-orbit-ry: z ∈ sigma-sphere-minus-bad-action.orbit ?ry
      unfolding sigma-sphere-minus-bad-action.orbit-def
      using q z-q by blast
    have orbit-eq: sigma-sphere-minus-bad-action.orbit ?rx =
        sigma-sphere-minus-bad-action.orbit ?ry
      by (rule sigma-orbit-eq-if-common-point[OF rxX ryX z-orbit-rx z-orbit-ry])
    have rep-rx: hausdorff-rep ?rx = ?rx
      by (rule hausdorff-rep-idem[OF x])
    have rep-ry: hausdorff-rep ?ry = ?ry
      by (rule hausdorff-rep-idem[OF y])
    have rx-eq-ry: ?rx = ?ry
      using hausdorff-rep-eq-if-orbit-eq[OF orbit-eq] rep-rx rep-ry by simp
    have p-eq-q: p = q
      by (rule sigma-free-same-point-eq[OF rxX p q]) (use z-p z-q rx-eq-ry in simp)
    have p ∈ u ∩ v
      using pu qv p-eq-q by auto
    with disj show z ∈ {}
      by simp
  qed
qed simp

lemma hausdorff-lift-pairwise-disjoint:
  assumes disj: pairwise-disjoint As
    and sub: ∀ u ∈ set As. u ⊆ carrier F2
  shows pairwise-disjoint (map hausdorff-lift As)
  unfolding pairwise-disjoint-def
proof (intro allI impI)
```

```
  fix i j
  assume i: i < length (map hausdorff-lift As)
    and j: j < length (map hausdorff-lift As)
    and ij: i ≠ j
  have iA: i < length As
    using i by simp
  have jA: j < length As
    using j by simp
  have Ai: As ! i ⊆ carrier F2
    using sub iA by simp
  have Aj: As ! j ⊆ carrier F2
    using sub jA by simp
  have As ! i ∩ As ! j = {}
    using disj iA jA ij by (simp add: pairwise-disjoint-def)
  hence hausdorff-lift (As ! i) ∩ hausdorff-lift (As ! j) = {}
    by (rule hausdorff-lift-disjoint[OF - Ai Aj])
  thus (map hausdorff-lift As) ! i ∩ (map hausdorff-lift As) ! j = {}
    using iA jA by simp
qed

lemma hausdorff-sigma-image-lift:
  assumes g: g ∈ carrier F2
    and u: u ⊆ carrier F2
  shows hausdorff-sigma-image-set g (hausdorff-lift u) =
    hausdorff-lift (F2-act.image-set g u)
proof
  show hausdorff-sigma-image-set g (hausdorff-lift u) ⊆
      hausdorff-lift (F2-act.image-set g u)
  proof
    fix z
    assume z ∈ hausdorff-sigma-image-set g (hausdorff-lift u)
    then obtain y where y: y ∈ hausdorff-lift u and z-eq: z = sigma g y
      unfolding hausdorff-sigma-image-set-def by blast
    from y obtain a x where au: a ∈ u and x: x ∈ sphere2 − bad-set-D
      and y-eq: y = sigma a (hausdorff-rep x)
      unfolding hausdorff-lift-def by blast
    have a: a ∈ carrier F2
      using u au by auto
    have ga-image: g ⊗_F2 a ∈ F2-act.image-set g u
      unfolding F2-act.image-set-def using au by auto
    have z = sigma (g ⊗_F2 a) (hausdorff-rep x)
      using sigma-homomorphism[OF g a] z-eq y-eq by (simp add: o-def)
    thus z ∈ hausdorff-lift (F2-act.image-set g u)
      unfolding hausdorff-lift-def using ga-image x by blast
  qed
  show hausdorff-lift (F2-act.image-set g u) ⊆
      hausdorff-sigma-image-set g (hausdorff-lift u)
  proof
    fix z
```

```
    assume z ∈ hausdorff-lift (F2-act.image-set g u)
    then obtain h x where h: h ∈ F2-act.image-set g u
      and x: x ∈ sphere2 − bad-set-D
      and z-eq: z = sigma h (hausdorff-rep x)
      unfolding hausdorff-lift-def by blast
    from h obtain a where au: a ∈ u and h-eq: h = g ⊗F2 a
      unfolding F2-act.image-set-def by auto
    have a: a ∈ carrier F2
      using u au by auto
    have y-lift: sigma a (hausdorff-rep x) ∈ hausdorff-lift u
      unfolding hausdorff-lift-def using au x by blast
    have z = sigma g (sigma a (hausdorff-rep x))
      using sigma-homomorphism[OF g a] z-eq h-eq by (simp add: o-def)
    thus z ∈ hausdorff-sigma-image-set g (hausdorff-lift u)
      unfolding hausdorff-sigma-image-set-def using y-lift by blast
  qed
qed

lemma hausdorff-translated-lift-pairwise-disjoint:
  assumes len: length As = length gs
    and gs: set gs ⊆ carrier F2
    and sub: ∀ u ∈ set As. u ⊆ carrier F2
    and disj: pairwise-disjoint (map2 F2-act.image-set gs As)
  shows pairwise-disjoint (map2 hausdorff-sigma-image-set gs (map hausdorff-lift
As))
  unfolding pairwise-disjoint-def
proof (intro allI impI)
  fix i j
  assume i: i < length (map2 hausdorff-sigma-image-set gs (map hausdorff-lift
As))
    and j: j < length (map2 hausdorff-sigma-image-set gs (map hausdorff-lift As))
    and ij: i ≠ j
  have iA: i < length As
    using i len by simp
  have jA: j < length As
    using j len by simp
  have ig: gs ! i ∈ carrier F2
  proof −
    have i < length gs
      using iA len by simp
    thus ?thesis
      using gs nth-mem by blast
  qed
  have jg: gs ! j ∈ carrier F2
  proof −
    have j < length gs
      using jA len by simp
    thus ?thesis
      using gs nth-mem by blast
```

```
  qed
  have Ai: As ! i ⊆ carrier F2
    using sub iA by simp
  have Aj: As ! j ⊆ carrier F2
    using sub jA by simp
  have img-i: F2-act.image-set (gs ! i) (As ! i) ⊆ carrier F2
    by (rule F2-act-image-set-subset-carrier[OF ig Ai])
  have img-j: F2-act.image-set (gs ! j) (As ! j) ⊆ carrier F2
    by (rule F2-act-image-set-subset-carrier[OF jg Aj])
  have img-disj: F2-act.image-set (gs ! i) (As ! i) ∩
      F2-act.image-set (gs ! j) (As ! j) = {}
    using disj iA jA ij len by (simp add: pairwise-disjoint-def)
  have lift-disj: hausdorff-lift (F2-act.image-set (gs ! i) (As ! i)) ∩
      hausdorff-lift (F2-act.image-set (gs ! j) (As ! j)) = {}
    by (rule hausdorff-lift-disjoint[OF img-disj img-i img-j])
  have trans-i: hausdorff-sigma-image-set (gs ! i) (hausdorff-lift (As ! i)) =
      hausdorff-lift (F2-act.image-set (gs ! i) (As ! i))
    by (rule hausdorff-sigma-image-lift[OF ig Ai])
  have trans-j: hausdorff-sigma-image-set (gs ! j) (hausdorff-lift (As ! j)) =
      hausdorff-lift (F2-act.image-set (gs ! j) (As ! j))
    by (rule hausdorff-sigma-image-lift[OF jg Aj])
  show (map2 hausdorff-sigma-image-set gs (map hausdorff-lift As)) ! i ∩
      (map2 hausdorff-sigma-image-set gs (map hausdorff-lift As)) ! j = {}
    using lift-disj trans-i trans-j iA jA len by simp
qed

lemma hausdorff-lift-cover:
  assumes len: length As = length gs
    and gs: set gs ⊆ carrier F2
    and sub: ∀ u ∈ set As. u ⊆ carrier F2
    and cover: carrier F2 =
      (⋃ i<length As. F2-act.image-set (gs ! i) (As ! i))
  shows sphere2 − bad-set-D =
    (⋃ i<length As. hausdorff-sigma-image-set (gs ! i) (hausdorff-lift (As ! i)))
proof −
  let ?X = sphere2 − bad-set-D
  let ?U = (⋃ i<length As. hausdorff-sigma-image-set (gs ! i) (hausdorff-lift (As !
i)))
  show ?X = ?U
  proof (rule subset-antisym)
    show ?X ⊆ ?U
    proof
      fix x
      assume x: x ∈ ?X
      interpret G: group F2
        by (rule free-group-is-group)
      from hausdorff-rep-in-orbit[OF x] obtain r
        where r: r ∈ carrier F2 and rep-eq: hausdorff-rep x = sigma r x
        unfolding sigma-sphere-minus-bad-action.orbit-def by blast
```

define h **where** $h = inv_{F2}\ r$
have h: $h \in carrier\ F2$
unfolding $h\text{-}def$ **by** $(rule\ G.inv\text{-}closed[OF\ r])$
have $x\text{-}eq$: $x = sigma\ h\ (hausdorff\text{-}rep\ x)$
unfolding $h\text{-}def$ **using** $rep\text{-}eq\ sigma\text{-}inverse\text{-}left[OF\ r]$ **by** $simp$
from $cover\ h$ **obtain** i **where** i: $i < length\ As$
and $h\text{-}img$: $h \in F2\text{-}act.image\text{-}set\ (gs\ !\ i)\ (As\ !\ i)$
by $blast$
from $h\text{-}img$ **obtain** a **where** aA: $a \in As\ !\ i$ **and** $h\text{-}eq$: $h = gs\ !\ i \otimes_{F2} a$
unfolding $F2\text{-}act.image\text{-}set\text{-}def$ **by** $auto$
have gi: $gs\ !\ i \in carrier\ F2$
proof $-$
have $i < length\ gs$
using $i\ len$ **by** $simp$
thus $?thesis$
using $gs\ nth\text{-}mem$ **by** $blast$
qed
have Ai: $As\ !\ i \subseteq carrier\ F2$
using $sub\ i$ **by** $simp$
have a: $a \in carrier\ F2$
using $Ai\ aA$ **by** $auto$
have $y\text{-}lift$: $sigma\ a\ (hausdorff\text{-}rep\ x) \in hausdorff\text{-}lift\ (As\ !\ i)$
unfolding $hausdorff\text{-}lift\text{-}def$ **using** $aA\ x$ **by** $blast$
have $sigma\ (gs\ !\ i)\ (sigma\ a\ (hausdorff\text{-}rep\ x)) = sigma\ h\ (hausdorff\text{-}rep\ x)$
using $sigma\text{-}homomorphism[OF\ gi\ a]\ h\text{-}eq$ **by** $(simp\ add{:}\ o\text{-}def)$
hence $x = sigma\ (gs\ !\ i)\ (sigma\ a\ (hausdorff\text{-}rep\ x))$
using $x\text{-}eq$ **by** $simp$
hence $x \in hausdorff\text{-}sigma\text{-}image\text{-}set\ (gs\ !\ i)\ (hausdorff\text{-}lift\ (As\ !\ i))$
unfolding $hausdorff\text{-}sigma\text{-}image\text{-}set\text{-}def$ **using** $y\text{-}lift$ **by** $blast$
thus $x \in ?U$
using i **by** $blast$
qed
show $?U \subseteq ?X$
proof
fix y
assume $y \in ?U$
then obtain i **where** i: $i < length\ As$
and $y\text{-}img$: $y \in hausdorff\text{-}sigma\text{-}image\text{-}set\ (gs\ !\ i)\ (hausdorff\text{-}lift\ (As\ !\ i))$
by $blast$
from $y\text{-}img$ **obtain** z **where** $z\text{-}lift$: $z \in hausdorff\text{-}lift\ (As\ !\ i)$
and $y\text{-}eq$: $y = sigma\ (gs\ !\ i)\ z$
unfolding $hausdorff\text{-}sigma\text{-}image\text{-}set\text{-}def$ **by** $blast$
have gi: $gs\ !\ i \in carrier\ F2$
proof $-$
have $i < length\ gs$
using $i\ len$ **by** $simp$
thus $?thesis$
using $gs\ nth\text{-}mem$ **by** $blast$
qed

```
    have Ai: As ! i ⊆ carrier F2
      using sub i by simp
    have zX: z ∈ ?X
      using hausdorff-lift-subset-X[OF Ai] z-lift by auto
    show y ∈ ?X
      using sigma-preserves-sphere2-minus-bad-set[OF gi zX] y-eq by simp
   qed
  qed
qed
```

26 The Hausdorff paradox

Define the rotation-action of F_2 on the sphere by evaluation: *sigma w · x = sigma w x*. Off the bad set, this is a free action, and the paradoxical decomposition of F_2 transports along orbits.

Hausdorff's theorem: $S^2 \setminus D$ is paradoxical under the action of SO(3).

```
theorem hausdorff-paradox-strong:
 ∃ P Q :: (real^3) set list. ∃ gP gQ :: ((bool × gen2) list) list.
     length P = length gP ∧ length Q = length gQ ∧
     set gP ⊆ carrier F2 ∧ set gQ ⊆ carrier F2 ∧
     set (map sigma gP) ⊆ SO3 ∧ set (map sigma gQ) ⊆ SO3 ∧
     pairwise-disjoint (P @ Q) ∧
     pairwise-disjoint (map2 hausdorff-sigma-image-set gP P) ∧
     pairwise-disjoint (map2 hausdorff-sigma-image-set gQ Q) ∧
     (∀ i < length P. P ! i ⊆ sphere2 − bad-set-D) ∧
     (∀ i < length Q. Q ! i ⊆ sphere2 − bad-set-D) ∧
     (sphere2 − bad-set-D) =
       (⋃ i<length P. P ! i) ∪ (⋃ i<length Q. Q ! i) ∧
     (sphere2 − bad-set-D) =
       (⋃ i<length P. hausdorff-sigma-image-set (gP ! i) (P ! i)) ∧
     (sphere2 − bad-set-D) =
       (⋃ i<length Q. hausdorff-sigma-image-set (gQ ! i) (Q ! i))
proof −
 from F2-paradoxical-partition obtain P0 Q0 :: ((bool × gen2) list) set list
   and gP gQ :: ((bool × gen2) list) list
   where lenP: length P0 = length gP
     and lenQ: length Q0 = length gQ
     and gP: set gP ⊆ carrier F2
     and gQ: set gQ ⊆ carrier F2
     and disj-all: pairwise-disjoint (P0 @ Q0)
     and disj-imP: pairwise-disjoint (map2 F2-act.image-set gP P0)
     and disj-imQ: pairwise-disjoint (map2 F2-act.image-set gQ Q0)
     and source-cover0: carrier F2 =
       (⋃ i<length P0. P0 ! i) ∪ (⋃ i<length Q0. Q0 ! i)
      and coverP: carrier F2 = (⋃ i<length P0. F2-act.image-set (gP ! i) (P0 !
i))
      and coverQ: carrier F2 = (⋃ i<length Q0. F2-act.image-set (gQ ! i) (Q0 !
```

i))
by *auto*
let *?P* = *map hausdorff-lift P0*
let *?Q* = *map hausdorff-lift Q0*
have *P0-sub*: ∀ *u* ∈ *set P0*. *u* ⊆ *carrier F2*
proof
fix *u*
assume *u* ∈ *set P0*
then obtain *i* **where** *i*: *i* < *length P0* **and** *u-eq*: *u* = *P0* ! *i*
by (*auto simp*: *in-set-conv-nth*)
show *u* ⊆ *carrier F2*
using *source-cover0 i u-eq* **by** *auto*
qed
have *Q0-sub*: ∀ *u* ∈ *set Q0*. *u* ⊆ *carrier F2*
proof
fix *u*
assume *u* ∈ *set Q0*
then obtain *i* **where** *i*: *i* < *length Q0* **and** *u-eq*: *u* = *Q0* ! *i*
by (*auto simp*: *in-set-conv-nth*)
show *u* ⊆ *carrier F2*
using *source-cover0 i u-eq* **by** *auto*
qed
have *all-sub*: ∀ *u* ∈ *set* (*P0* @ *Q0*). *u* ⊆ *carrier F2*
using *P0-sub Q0-sub* **by** *auto*
have *lenP'*: *length ?P* = *length gP*
using *lenP* **by** *simp*
have *lenQ'*: *length ?Q* = *length gQ*
using *lenQ* **by** *simp*
have *gP-SO3*: *set* (*map sigma gP*) ⊆ *SO3*
using *gP sigma-image-in-SO3* **by** *auto*
have *gQ-SO3*: *set* (*map sigma gQ*) ⊆ *SO3*
using *gQ sigma-image-in-SO3* **by** *auto*
have *source-disj*: *pairwise-disjoint* (*?P* @ *?Q*)
using *hausdorff-lift-pairwise-disjoint*[*OF disj-all all-sub*] **by** *simp*
have *trans-disjP*: *pairwise-disjoint* (*map2 hausdorff-sigma-image-set gP ?P*)
by (*rule hausdorff-translated-lift-pairwise-disjoint*[*OF lenP gP P0-sub disj-imP*])
have *trans-disjQ*: *pairwise-disjoint* (*map2 hausdorff-sigma-image-set gQ ?Q*)
by (*rule hausdorff-translated-lift-pairwise-disjoint*[*OF lenQ gQ Q0-sub disj-imQ*])
have *P-sub-X*: ∀ *i* < *length ?P*. *?P* ! *i* ⊆ *sphere2* − *bad-set-D*
proof (*intro allI impI*)
fix *i*
assume *i*: *i* < *length ?P*
have *P0* ! *i* ⊆ *carrier F2*
using *P0-sub i* **by** *simp*
thus *?P* ! *i* ⊆ *sphere2* − *bad-set-D*
using *i hausdorff-lift-subset-X* **by** *simp*
qed
have *Q-sub-X*: ∀ *i* < *length ?Q*. *?Q* ! *i* ⊆ *sphere2* − *bad-set-D*
proof (*intro allI impI*)

```
  fix i
  assume i: i < length ?Q
  have Q0 ! i ⊆ carrier F2
    using Q0-sub i by simp
  thus ?Q ! i ⊆ sphere2 − bad-set-D
    using i hausdorff-lift-subset-X by simp
qed
have source-cover': sphere2 − bad-set-D =
    (⋃i<length ?P. ?P ! i) ∪ (⋃i<length ?Q. ?Q ! i)
proof (rule subset-antisym)
  show sphere2 − bad-set-D ⊆
      (⋃i<length ?P. ?P ! i) ∪ (⋃i<length ?Q. ?Q ! i)
  proof
    fix x
    assume x: x ∈ sphere2 − bad-set-D
    interpret G: group F2
      by (rule free-group-is-group)
    from hausdorff-rep-in-orbit[OF x] obtain r
      where r: r ∈ carrier F2 and rep-eq: hausdorff-rep x = sigma r x
      unfolding sigma-sphere-minus-bad-action.orbit-def by blast
    define h where h = inv_F2 r
    have h: h ∈ carrier F2
      unfolding h-def by (rule G.inv-closed[OF r])
    have x-eq: x = sigma h (hausdorff-rep x)
      unfolding h-def using rep-eq sigma-inverse-left[OF r] by simp
    from source-cover0 h have h-cases:
      h ∈ (⋃i<length P0. P0 ! i) ∨ h ∈ (⋃i<length Q0. Q0 ! i)
      by blast
    thus x ∈ (⋃i<length ?P. ?P ! i) ∪ (⋃i<length ?Q. ?Q ! i)
    proof
      assume h ∈ (⋃i<length P0. P0 ! i)
      then obtain i where i: i < length P0 and hP: h ∈ P0 ! i
        by blast
      have x ∈ hausdorff-lift (P0 ! i)
        unfolding hausdorff-lift-def using hP x x-eq by blast
      hence x ∈ ?P ! i
        using i by simp
      hence x ∈ (⋃i<length ?P. ?P ! i)
        using i by auto
      thus ?thesis
        by blast
    next
      assume h ∈ (⋃i<length Q0. Q0 ! i)
      then obtain i where i: i < length Q0 and hQ: h ∈ Q0 ! i
        by blast
      have x ∈ hausdorff-lift (Q0 ! i)
        unfolding hausdorff-lift-def using hQ x x-eq by blast
      hence x ∈ ?Q ! i
        using i by simp
```

```
      hence x ∈ (⋃ i<length ?Q. ?Q ! i)
        using i by auto
      thus ?thesis
        by blast
    qed
  qed
  show (⋃ i<length ?P. ?P ! i) ∪ (⋃ i<length ?Q. ?Q ! i)
      ⊆ sphere2 − bad-set-D
    using P-sub-X Q-sub-X by auto
 qed
 have coverP': sphere2 − bad-set-D =
    (⋃ i<length ?P. hausdorff-sigma-image-set (gP ! i) (?P ! i))
  using hausdorff-lift-cover[OF lenP gP P0-sub coverP] by simp
 have coverQ': sphere2 − bad-set-D =
    (⋃ i<length ?Q. hausdorff-sigma-image-set (gQ ! i) (?Q ! i))
  using hausdorff-lift-cover[OF lenQ gQ Q0-sub coverQ] by simp
 show ?thesis
  apply (rule exI[of - ?P])
  apply (rule exI[of - ?Q])
  apply (rule exI[of - gP])
  apply (rule exI[of - gQ])
  using lenP' lenQ' gP gQ gP-SO3 gQ-SO3 source-disj trans-disjP trans-disjQ
    P-sub-X Q-sub-X source-cover' coverP' coverQ'
  by blast
qed

theorem hausdorff-paradoxical:
 sigma-sphere-minus-bad-action.paradoxical (sphere2 − bad-set-D)
proof −
 from hausdorff-paradox-strong obtain P Q :: (real^3) set list
  and gP gQ :: ((bool × gen2) list) list
  where length P = length gP and length Q = length gQ
   and set gP ⊆ carrier F2 and set gQ ⊆ carrier F2
   and pairwise-disjoint (P @ Q)
   and pairwise-disjoint (map2 hausdorff-sigma-image-set gP P)
   and pairwise-disjoint (map2 hausdorff-sigma-image-set gQ Q)
   and P-sub: ∀ i < length P. P ! i ⊆ sphere2 − bad-set-D
   and Q-sub: ∀ i < length Q. Q ! i ⊆ sphere2 − bad-set-D
   and coverP: sphere2 − bad-set-D =
     (⋃ i<length P. hausdorff-sigma-image-set (gP ! i) (P ! i))
   and coverQ: sphere2 − bad-set-D =
     (⋃ i<length Q. hausdorff-sigma-image-set (gQ ! i) (Q ! i))
  by auto
 have pieces-sub: (⋃ i<length P. P ! i) ∪ (⋃ i<length Q. Q ! i)
    ⊆ sphere2 − bad-set-D
  using P-sub Q-sub by auto
 have imageP-eq: map2 sigma-sphere-minus-bad-action.image-set gP P =
    map2 hausdorff-sigma-image-set gP P
  by (rule map2-hausdorff-sigma-image-set-eq)
```

```
  have imageQ-eq: map2 sigma-sphere-minus-bad-action.image-set gQ Q =
      map2 hausdorff-sigma-image-set gQ Q
    by (rule map2-hausdorff-sigma-image-set-eq)
  have coverP-action: sphere2 − bad-set-D =
      (⋃ i<length P. sigma-sphere-minus-bad-action.image-set (gP ! i) (P ! i))
    using coverP
   by (simp add: hausdorff-sigma-image-set-def sigma-sphere-minus-bad-action.image-set-def)
  have coverQ-action: sphere2 − bad-set-D =
      (⋃ i<length Q. sigma-sphere-minus-bad-action.image-set (gQ ! i) (Q ! i))
    using coverQ
   by (simp add: hausdorff-sigma-image-set-def sigma-sphere-minus-bad-action.image-set-def)
  have imageP-disj-action: pairwise-disjoint (map2 sigma-sphere-minus-bad-action.image-set
gP P)
    using ‹pairwise-disjoint (map2 hausdorff-sigma-image-set gP P)› imageP-eq
    by metis
  have imageQ-disj-action: pairwise-disjoint (map2 sigma-sphere-minus-bad-action.image-set
gQ Q)
    using ‹pairwise-disjoint (map2 hausdorff-sigma-image-set gQ Q)› imageQ-eq
    by metis
  show ?thesis
    unfolding sigma-sphere-minus-bad-action.paradoxical-def
    apply (rule exI[of - P])
    apply (rule exI[of - Q])
    apply (rule exI[of - gP])
    apply (rule exI[of - gQ])
    using ‹length P = length gP› ‹length Q = length gQ›
      ‹set gP ⊆ carrier F2› ‹set gQ ⊆ carrier F2›
      ‹pairwise-disjoint (P @ Q)›
     imageP-disj-action imageQ-disj-action pieces-sub coverP-action coverQ-action
    by blast
qed

theorem hausdorff-paradox-rot-strong:
  ∃ P Q :: (real^3) set list. ∃ gP gQ :: ((real^3) ⇒ (real^3)) list.
      length P = length gP ∧ length Q = length gQ ∧
      set gP ⊆ SO3 ∧ set gQ ⊆ SO3 ∧
      pairwise-disjoint (P @ Q) ∧
      pairwise-disjoint (map2 (λg A. g ‘ A) gP P) ∧
      pairwise-disjoint (map2 (λg A. g ‘ A) gQ Q) ∧
      (∀ i < length P. P ! i ⊆ sphere2 − bad-set-D) ∧
      (∀ i < length Q. Q ! i ⊆ sphere2 − bad-set-D) ∧
      (sphere2 − bad-set-D) =
        (⋃ i<length P. P ! i) ∪ (⋃ i<length Q. Q ! i) ∧
      (sphere2 − bad-set-D) =
        (⋃ i<length P. (gP ! i) ‘ (P ! i)) ∧
       (sphere2 − bad-set-D) =
         (⋃ i<length Q. (gQ ! i) ‘ (Q ! i))
proof −
  from hausdorff-paradox-strong obtain P Q :: (real^3) set list
```

and *wP wQ* :: ((*bool* × *gen2*) *list*) *list*
where *lenP*: *length P* = *length wP*
and *lenQ*: *length Q* = *length wQ*
and *wP-SO3*: *set* (*map sigma wP*) ⊆ *SO3*
and *wQ-SO3*: *set* (*map sigma wQ*) ⊆ *SO3*
and *disj*: *pairwise-disjoint* (*P* @ *Q*)
and *disjP*: *pairwise-disjoint* (*map2 hausdorff-sigma-image-set wP P*)
and *disjQ*: *pairwise-disjoint* (*map2 hausdorff-sigma-image-set wQ Q*)
and *P-sub*: ∀ *i* < *length P*. *P* ! *i* ⊆ *sphere2* − *bad-set-D*
and *Q-sub*: ∀ *i* < *length Q*. *Q* ! *i* ⊆ *sphere2* − *bad-set-D*
and *source-cover*: *sphere2* − *bad-set-D* =
(⋃ *i*<*length P*. *P* ! *i*) ∪ (⋃ *i*<*length Q*. *Q* ! *i*)
and *coverP*: *sphere2* − *bad-set-D* =
(⋃ *i*<*length P*. *hausdorff-sigma-image-set* (*wP* ! *i*) (*P* ! *i*))
and *coverQ*: *sphere2* − *bad-set-D* =
(⋃ *i*<*length Q*. *hausdorff-sigma-image-set* (*wQ* ! *i*) (*Q* ! *i*))
by *auto*
let *?gP* = *map sigma wP*
let *?gQ* = *map sigma wQ*
have *map2P-eq*:
map2 (λ*g A*. *g* ‘ *A*) *?gP P* = *map2 hausdorff-sigma-image-set wP P*
using *lenP*
proof (*induction wP arbitrary*: *P*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons w ws*)
then obtain *A Ps* **where** *P*: *P* = *A* # *Ps*
by (*cases P*) *auto*
with *Cons* **show** *?case*
by (*simp add*: *hausdorff-sigma-image-set-def*)
qed
have *map2Q-eq*:
map2 (λ*g A*. *g* ‘ *A*) *?gQ Q* = *map2 hausdorff-sigma-image-set wQ Q*
using *lenQ*
proof (*induction wQ arbitrary*: *Q*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons w ws*)
then obtain *A Qs* **where** *Q*: *Q* = *A* # *Qs*
by (*cases Q*) *auto*
with *Cons* **show** *?case*
by (*simp add*: *hausdorff-sigma-image-set-def*)
qed
have *disjP-rot*: *pairwise-disjoint* (*map2* (λ*g A*. *g* ‘ *A*) *?gP P*)
using *disjP map2P-eq* **by** *simp*
have *disjQ-rot*: *pairwise-disjoint* (*map2* (λ*g A*. *g* ‘ *A*) *?gQ Q*)
using *disjQ map2Q-eq* **by** *simp*

```
  have coverP-rot: sphere2 − bad-set-D =
     (⋃i<length P. (?gP ! i) ‘ (P ! i))
    using coverP lenP by (simp add: hausdorff-sigma-image-set-def)
  have coverQ-rot: sphere2 − bad-set-D =
     (⋃i<length Q. (?gQ ! i) ‘ (Q ! i))
    using coverQ lenQ by (simp add: hausdorff-sigma-image-set-def)
  show ?thesis
    apply (rule exI[of - P])
    apply (rule exI[of - Q])
    apply (rule exI[of - ?gP])
    apply (rule exI[of - ?gQ])
    using lenP lenQ wP-SO3 wQ-SO3 disj disjP-rot disjQ-rot
      P-sub Q-sub source-cover coverP-rot coverQ-rot
    by simp
qed

theorem hausdorff-paradox:
  ∃ P Q :: (real^3) set list. ∃ gP gQ :: ((real^3) ⇒ (real^3)) list.
       length P = length gP ∧ length Q = length gQ ∧
       set gP ⊆ SO3 ∧ set gQ ⊆ SO3 ∧
       (∀ i < length P. P ! i ⊆ sphere2 − bad-set-D) ∧
       (∀ i < length Q. Q ! i ⊆ sphere2 − bad-set-D) ∧
       (sphere2 − bad-set-D) =
         (⋃i<length P. (gP ! i) ‘ (P ! i)) ∧
        (sphere2 − bad-set-D) =
          (⋃i<length Q. (gQ ! i) ‘ (Q ! i))
proof −
  from hausdorff-paradox-strong obtain P Q :: (real^3) set list
    and wP wQ :: ((bool × gen2) list) list
    where lenP: length P = length wP
      and lenQ: length Q = length wQ
      and wP-SO3: set (map sigma wP) ⊆ SO3
      and wQ-SO3: set (map sigma wQ) ⊆ SO3
      and P-sub: ∀ i < length P. P ! i ⊆ sphere2 − bad-set-D
      and Q-sub: ∀ i < length Q. Q ! i ⊆ sphere2 − bad-set-D
      and coverP: sphere2 − bad-set-D =
        (⋃i<length P. hausdorff-sigma-image-set (wP ! i) (P ! i))
      and coverQ: sphere2 − bad-set-D =
        (⋃i<length Q. hausdorff-sigma-image-set (wQ ! i) (Q ! i))
    by auto
  let ?gP = map sigma wP
  let ?gQ = map sigma wQ
  have coverP-rot: sphere2 − bad-set-D =
     (⋃i<length P. (?gP ! i) ‘ (P ! i))
    using coverP lenP by (simp add: hausdorff-sigma-image-set-def)
  have coverQ-rot: sphere2 − bad-set-D =
     (⋃i<length Q. (?gQ ! i) ‘ (Q ! i))
    using coverQ lenQ by (simp add: hausdorff-sigma-image-set-def)
  show ?thesis
```

apply (*rule exI*[*of* - *P*])
apply (*rule exI*[*of* - *Q*])
apply (*rule exI*[*of* - *?gP*])
apply (*rule exI*[*of* - *?gQ*])
using *lenP lenQ wP-SO3 wQ-SO3 P-sub Q-sub coverP-rot coverQ-rot*
by *simp*
qed

end

theory *Sphere-Decomposition*
imports *Hausdorff-Paradox*
begin

Equidecomposability of $S^2 \setminus D$ and S^2.

definition *Rz-angle-mat* :: *real* ⇒ *real^3^3* **where**
Rz-angle-mat t = *vector* [
vector [*cos t*, − *sin t*, *0*],
vector [*sin t*, *cos t*, *0*],
vector [*0*, *0*, *1*]]

definition *Rz-angle* :: *real* ⇒ *real^3* ⇒ *real^3* **where**
Rz-angle t v = *Rz-angle-mat t* ∗*v v*

definition *e1-3* :: *real^3* **where**
e1-3 = *vector* [*1*, *0*, *0*]

definition *e3-3* :: *real^3* **where**
e3-3 = *vector* [*0*, *0*, *1*]

lemma *Rz-angle-mat-orthogonal*: *orthogonal-matrix* (*Rz-angle-mat t*)
unfolding *orthogonal-matrix Rz-angle-mat-def*
by (*simp add*: *matrix-matrix-mult-def transpose-def mat-def vec-eq-iff vector-def sum-3 forall-3*
power2-eq-square algebra-simps)

lemma *det-Rz-angle-mat* [*simp*]: *det* (*Rz-angle-mat t*) = *1*
unfolding *Rz-angle-mat-def*
by (*simp add*: *det-3 vector-def power2-eq-square algebra-simps*)

lemma *Rz-angle-in-SO3*: *Rz-angle t* ∈ *SO3*
unfolding *SO3-def rotation-def Rz-angle-def*
by (*simp add*: *orthogonal-transformation-matrix Rz-angle-mat-orthogonal matrix-vector-mul-linear*)

lemma *Rz-angle-mat-add*:
Rz-angle-mat a ∗∗ *Rz-angle-mat b* = *Rz-angle-mat* (*a* + *b*)
unfolding *Rz-angle-mat-def*

by (*simp add*: *matrix-matrix-mult-def vec-eq-iff vector-def sum-3 forall-3 cos-add sin-add algebra-simps*)

lemma *Rz-angle-compose*:
Rz-angle a ∘ *Rz-angle b* = *Rz-angle* (*a* + *b*)
by (*rule ext*) (*simp add*: *Rz-angle-def matrix-vector-mul-assoc Rz-angle-mat-add*)

lemma *Rz-angle-funpow*:
(*Rz-angle t* ^^ *n*) = *Rz-angle* (*real n* ∗ *t*)
proof (*induction n*)
case *0*
show *?case*
by (*rule ext*) (*simp add*: *Rz-angle-def Rz-angle-mat-def mat-def matrix-vector-mult-def vec-eq-iff vector-def forall-3 sum-3*)
next
case (*Suc n*)
have (*Rz-angle t* ^^ *Suc n*) = *Rz-angle t* ∘ *Rz-angle* (*real n* ∗ *t*)
using *Suc.IH* **by** *simp*
also have ... = *Rz-angle* (*t* + *real n* ∗ *t*)
by (*rule Rz-angle-compose*)
also have *t* + *real n* ∗ *t* = *real* (*Suc n*) ∗ *t*
by (*simp add*: *algebra-simps*)
finally show *?case* .
qed

lemma *Rz-angle-components* [*simp*]:
(*Rz-angle t x*) $ *1* = *cos t* ∗ *x* $ *1* − *sin t* ∗ *x* $ *2*
(*Rz-angle t x*) $ *2* = *sin t* ∗ *x* $ *1* + *cos t* ∗ *x* $ *2*
(*Rz-angle t x*) $ *3* = *x* $ *3*
by (*simp-all add*: *Rz-angle-def Rz-angle-mat-def matrix-vector-mult-def vector-def sum-3*)

lemma *Rz-angle-eq-non-axis-sin-cos*:
assumes *Rz-angle a x* = *Rz-angle b x* **and** *x* $ *1* ≠ *0* ∨ *x* $ *2* ≠ *0*
shows *sin a* = *sin b* ∧ *cos a* = *cos b*
proof −
let *?X* = *x* $ *1*
let *?Y* = *x* $ *2*
let *?C* = *cos a* − *cos b*
let *?S* = *sin a* − *sin b*
have *comp1*: (*Rz-angle a x*) $ *1* = (*Rz-angle b x*) $ *1*
using *arg-cong*[*OF assms*(*1*), *of* λ*v*. *v* $ *1*] .
have *comp2*: (*Rz-angle a x*) $ *2* = (*Rz-angle b x*) $ *2*
using *arg-cong*[*OF assms*(*1*), *of* λ*v*. *v* $ *2*] .
have *eq1*: *?C* ∗ *?X* − *?S* ∗ *?Y* = *0*
using *comp1* **by** (*simp add*: *algebra-simps*)
have *eq2*: *?S* ∗ *?X* + *?C* ∗ *?Y* = *0*
using *comp2* **by** (*simp add*: *algebra-simps*)
have *nonzero*: $?X^2 + ?Y^2 \neq 0$

```
    using assms(2) by (simp add: power2-eq-square sum-squares-eq-zero-iff)
  have C-eq: ?C * (?X^2 + ?Y^2) =
      (?C * ?X - ?S * ?Y) * ?X + (?S * ?X + ?C * ?Y) * ?Y
    by (simp add: power2-eq-square algebra-simps)
  have Cprod0: ?C * (?X^2 + ?Y^2) = 0
    using C-eq eq1 eq2 by simp
  hence C-or: ?C = 0 ∨ ?X^2 + ?Y^2 = 0
    by (simp add: mult-eq-0-iff)
  have C0: ?C = 0
    using C-or nonzero by blast
  have S-eq: ?S * (?X^2 + ?Y^2) =
      (?S * ?X + ?C * ?Y) * ?X - (?C * ?X - ?S * ?Y) * ?Y
    by (simp add: power2-eq-square algebra-simps)
  have Sprod0: ?S * (?X^2 + ?Y^2) = 0
    using S-eq eq1 eq2 by simp
  hence S-or: ?S = 0 ∨ ?X^2 + ?Y^2 = 0
    by (simp add: mult-eq-0-iff)
  have S0: ?S = 0
    using S-or nonzero by blast
  from S0 C0 show ?thesis
    by simp
qed

lemma Rz-angle-collision-angles-countable:
  assumes nonaxis: x $ 1 ≠ 0 ∨ x $ 2 ≠ 0
  shows countable {t. Rz-angle t x = y}
proof (cases {t. Rz-angle t x = y} = {})
  case True
  then show ?thesis by simp
next
  case False
  then obtain a where a: Rz-angle a x = y
    by auto
  have subset: {t. Rz-angle t x = y} ⊆ (λn::int. a + 2 * pi * n) ` UNIV
  proof
    fix t
    assume t: t ∈ {t. Rz-angle t x = y}
    hence eq: Rz-angle t x = Rz-angle a x
      using a by simp
    have trig: sin t = sin a ∧ cos t = cos a
      by (rule Rz-angle-eq-non-axis-sin-cos[OF eq nonaxis])
    have ∃ n::int. t = a + 2 * pi * n
      using iffD1[OF sin-cos-eq-iff[of t a] trig] .
    then obtain n :: int where n: t = a + 2 * pi * n
      by blast
    show t ∈ (λn::int. a + 2 * pi * n) ` UNIV
      using n by simp
  qed
  have countable ((λn::int. a + 2 * pi * n) ` UNIV)
```

```
    by simp
  thus ?thesis
    by (rule countable-subset[OF subset])
qed

lemma Rz-angle-scaled-collision-angles-countable:
  assumes k > 0 and nonaxis: x $ 1 ≠ 0 ∨ x $ 2 ≠ 0
  shows countable {t. Rz-angle (real k * t) x = y}
proof −
  let ?f = λt::real. real k * t
  let ?A = {u. Rz-angle u x = y}
  have count-A: countable ?A
    by (rule Rz-angle-collision-angles-countable[OF nonaxis])
  have inj-f: inj-on ?f UNIV
    using assms(1) by (auto simp: inj-on-def)
  have count-pre: countable {t ∈ UNIV. ?f t ∈ ?A}
    by (rule countable-image-inj-gen[OF inj-f count-A])
  thus ?thesis
    by simp
qed

lemma e1-3-in-sphere2: e1-3 ∈ sphere2
  by (simp add: e1-3-def sphere2-def norm-eq-1 inner-vec-def sum-3 vector-def)

lemma e1-3-nonaxis: e1-3 $ 1 ≠ 0 ∨ e1-3 $ 2 ≠ 0
  by (simp add: e1-3-def)

lemma e3-3-in-sphere2: e3-3 ∈ sphere2
  by (simp add: e3-3-def sphere2-def norm-eq-1 inner-vec-def sum-3 vector-def)

lemma z-axis-sphere2-cases:
  assumes x ∈ sphere2 and x $ 1 = 0 and x $ 2 = 0
  shows x = e3-3 ∨ x = − e3-3
proof −
  have x3-sq: (x $ 3)^2 = 1
    using assms unfolding sphere2-def
    by (simp add: norm-eq-1 inner-vec-def sum-3 power2-eq-square)
  hence x $ 3 = 1 ∨ x $ 3 = −1
    by (simp add: power2-eq-1-iff)
  thus ?thesis
    using assms(2,3)
    by (auto simp: e3-3-def vec-eq-iff vector-def forall-3)
qed

lemma SO3-funpow:
  assumes R ∈ SO3
  shows (R ^^ n) ∈ SO3
proof (induction n)
  case 0
```

```
  show ?case
    using id-in-SO3 by (simp add: id-def)
next
  case (Suc n)
  have R ∘ (R ^^ n) ∈ SO3
    by (rule SO3-closed-compose[OF assms Suc.IH])
  thus ?case
    by (simp add: o-def)
qed

lemma SO3-inj:
  assumes f ∈ SO3
  shows inj f
proof −
  have orthogonal-transformation f
    using assms unfolding SO3-def rotation-def by auto
  thus ?thesis
    by (rule orthogonal-transformation-inj)
qed

lemma SO3-surj:
  assumes f ∈ SO3
  shows surj f
proof −
  have orthogonal-transformation f
    using assms unfolding SO3-def rotation-def by auto
  thus ?thesis
    by (rule orthogonal-transformation-surj)
qed

lemma SO3-linear:
  assumes f ∈ SO3
  shows linear f
proof −
  have orthogonal-transformation f
    using assms unfolding SO3-def rotation-def by auto
  thus ?thesis
    by (rule orthogonal-transformation-linear)
qed

lemma SO3-inverse:
  assumes f ∈ SO3
  shows Hilbert-Choice.inv f ∈ SO3
proof −
  have f-orth: orthogonal-transformation f
    and f-det: det (matrix f) = 1
    using assms unfolding SO3-def rotation-def by auto
  have inv-orth: orthogonal-transformation (Hilbert-Choice.inv f)
    by (rule orthogonal-transformation-inv[OF f-orth])
```

```
  have f-lin: linear f
    by (rule orthogonal-transformation-linear[OF f-orth])
  have inv-lin: linear (Hilbert-Choice.inv f)
    by (rule orthogonal-transformation-linear[OF inv-orth])
  have surj-f: surj f
    by (rule orthogonal-transformation-surj[OF f-orth])
  have comp-id: f ∘ Hilbert-Choice.inv f = id
    using surj-f by (auto simp: fun-eq-iff surj-f-inv-f)
  have det (matrix (f ∘ Hilbert-Choice.inv f)) =
      det (matrix f) * det (matrix (Hilbert-Choice.inv f))
    using matrix-compose[OF inv-lin f-lin] by (simp add: det-mul)
  also have ... = det (matrix (Hilbert-Choice.inv f))
    using f-det by simp
  finally have det (matrix (Hilbert-Choice.inv f)) = 1
    using comp-id by simp
  with inv-orth show ?thesis
    unfolding SO3-def rotation-def by simp
qed

lemma conjugate-funpow:
  assumes bijS: bij S
  shows ((S ∘ R ∘ Hilbert-Choice.inv S) ^^ n) =
    S ∘ (R ^^ n) ∘ Hilbert-Choice.inv S
proof (induction n)
  case 0
  show ?case
    using bijS by (auto simp: fun-eq-iff bij-is-surj surj-f-inv-f)
next
  case (Suc n)
  have injS: inj S
    using bijS by (simp add: bij-is-inj)
  show ?case
  proof (rule ext)
    fix x
    have n-eq: ((S ∘ R ∘ Hilbert-Choice.inv S) ^^ n) x =
        S ((R ^^ n) (Hilbert-Choice.inv S x))
      using fun-cong[OF Suc.IH, of x] by (simp add: o-def)
    have ((S ∘ R ∘ Hilbert-Choice.inv S) ^^ Suc n) x =
        S (R (Hilbert-Choice.inv S (((S ∘ R ∘ Hilbert-Choice.inv S) ^^ n) x)))
      by simp
    also have ... = S (R (Hilbert-Choice.inv S
        (S ((R ^^ n) (Hilbert-Choice.inv S x)))))
      using n-eq by simp
    also have ... = S ((R ^^ Suc n) (Hilbert-Choice.inv S x))
      using injS by (simp add: inv-f-f)
    finally show ((S ∘ R ∘ Hilbert-Choice.inv S) ^^ Suc n) x =
        (S ∘ (R ^^ Suc n) ∘ Hilbert-Choice.inv S) x
      by simp
  qed
```

qed

lemma *conjugate-funpow-image*:
assumes *bijS*: *bij S*
shows $((S \circ R \circ \mathit{Hilbert\text{-}Choice.inv}\ S) \mathbin{\frown\frown} n)\ `\ D =$
$S\ `\ ((R \mathbin{\frown\frown} n)\ `\ (\mathit{Hilbert\text{-}Choice.inv}\ S\ `\ D))$
using *conjugate-funpow*[*OF bijS*, *of n R*]
by *auto*

lemma *exists-antipodal-axis-avoiding-countable*:
assumes *count-D*: *countable D*
shows $\exists a \in \mathit{sphere2}.\ a \notin D \wedge -a \notin D$
proof −
let *?A* = $D \cup \mathit{uminus}\ `\ D$
have *count-A*: *countable ?A*
using *count-D* **by** *simp*
let *?bad* = $(\bigcup y \in \mathit{?A}.\ \{t.\ \mathit{Rz\text{-}angle}\ t\ \mathit{e1\text{-}3} = y\})$
have *count-bad*: *countable ?bad*
proof (*intro countable-UN count-A*)
fix *y*
assume $y \in \mathit{?A}$
show *countable* $\{t.\ \mathit{Rz\text{-}angle}\ t\ \mathit{e1\text{-}3} = y\}$
by (*rule Rz-angle-collision-angles-countable*[*OF e1-3-nonaxis*])
qed
obtain *t* **where** *t-not-bad*: $t \notin \mathit{?bad}$
using *real-interval-avoid-countable-set*[*OF zero-less-one count-bad*] **by** *blast*
define *a* **where** $a = \mathit{Rz\text{-}angle}\ t\ \mathit{e1\text{-}3}$
have *a-sphere*: $a \in \mathit{sphere2}$
unfolding *a-def*
by (*rule rotation-preserves-sphere2*[*OF Rz-angle-in-SO3 e1-3-in-sphere2*])
have *a-not-A*: $a \notin \mathit{?A}$
using *t-not-bad* **unfolding** *a-def* **by** *blast*
hence *a-not-D*: $a \notin D$
by *simp*
have *neg-a-not-D*: $-a \notin D$
proof
assume $-a \in D$
hence $\mathit{uminus}\ (-a) \in \mathit{uminus}\ `\ D$
by *blast*
hence $a \in \mathit{uminus}\ `\ D$
by *simp*
with *a-not-A* **show** *False*
by *simp*
qed
from *a-sphere a-not-D neg-a-not-D* **show** *?thesis*
by *blast*
qed

lemma *exists-Rz-angle-absorbing*:

```
  assumes count-D: countable D
    and nonaxis: ⋀x. x ∈ D ⟹ x $ 1 ≠ 0 ∨ x $ 2 ≠ 0
  shows ∃t. ∀n m. n ≠ m ⟶ ((Rz-angle t ^^ n) ‘ D) ∩ ((Rz-angle t ^^ m) ‘ D)
= {}
proof −
  let ?bad = (⋃k ∈ {k::nat. k > 0}. ⋃x ∈ D. ⋃y ∈ D.
    {t. Rz-angle (real k ∗ t) x = y})
  have count-bad: countable ?bad
  proof (intro countable-UN countable-Collect countableI-type count-D)
    fix k x y
    assume k ∈ {k::nat. 0 < k} and x ∈ D and y ∈ D
    thus countable {t. Rz-angle (real k ∗ t) x = y}
      by (intro Rz-angle-scaled-collision-angles-countable nonaxis) auto
  qed
  obtain t where t-not-bad: t ∉ ?bad
    using real-interval-avoid-countable-set[OF zero-less-one count-bad] by blast
  have disj: ((Rz-angle t ^^ n) ‘ D) ∩ ((Rz-angle t ^^ m) ‘ D) = {} if n ≠ m for
n m
  proof (rule ccontr)
    assume ((Rz-angle t ^^ n) ‘ D) ∩ ((Rz-angle t ^^ m) ‘ D) ≠ {}
    then obtain x y z where x: x ∈ D and y: y ∈ D
      and z: z = (Rz-angle t ^^ n) x z = (Rz-angle t ^^ m) y
      by blast
    have False if nm: n < m
    proof −
      let ?k = m − n
      have k-pos: ?k > 0
        using nm by simp
      have m-eq: m = n + ?k
        using nm by simp
      have eq: (Rz-angle t ^^ n) x = (Rz-angle t ^^ m) y
        using z by simp
      have tail: (Rz-angle t ^^ m) y =
          (Rz-angle t ^^ n) ((Rz-angle t ^^ ?k) y)
        using m-eq by (metis comp-apply funpow-add)
      have (Rz-angle t ^^ n) x = (Rz-angle t ^^ n) ((Rz-angle t ^^ ?k) y)
        using eq tail by simp
      moreover have inj (Rz-angle t ^^ n)
        by (rule SO3-inj[OF SO3-funpow[OF Rz-angle-in-SO3]])
      ultimately have x-eq: x = (Rz-angle t ^^ ?k) y
        by (meson injD)
      have Rz-angle (real ?k ∗ t) y = x
        using x-eq Rz-angle-funpow[of ?k t] by (simp add: fun-eq-iff)
      hence t ∈ {u. Rz-angle (real ?k ∗ u) y = x}
        by simp
      hence t ∈ ?bad
        using k-pos x y by blast
      with t-not-bad show False
        by contradiction
```

```
    qed
    moreover have False if mn: m < n
    proof −
      let ?k = n − m
      have k-pos: ?k > 0
        using mn by simp
      have n-eq: n = m + ?k
        using mn by simp
      have eq: (Rz-angle t ^^ m) y = (Rz-angle t ^^ n) x
        using z by simp
      have tail: (Rz-angle t ^^ n) x =
          (Rz-angle t ^^ m) ((Rz-angle t ^^ ?k) x)
        using n-eq by (metis comp-apply funpow-add)
      have (Rz-angle t ^^ m) y = (Rz-angle t ^^ m) ((Rz-angle t ^^ ?k) x)
        using eq tail by simp
      moreover have inj (Rz-angle t ^^ m)
        by (rule SO3-inj[OF SO3-funpow[OF Rz-angle-in-SO3]])
      ultimately have y-eq: y = (Rz-angle t ^^ ?k) x
        by (meson injD)
      have Rz-angle (real ?k * t) x = y
        using y-eq Rz-angle-funpow[of ?k t] by (simp add: fun-eq-iff)
      hence t ∈ {u. Rz-angle (real ?k * u) x = y}
        by simp
      hence t ∈ ?bad
        using k-pos x y by blast
      with t-not-bad show False
        by contradiction
    qed
    ultimately show False
      using that by linarith
  qed
  show ?thesis
    using disj by blast
qed

lemma exists-absorbing-rotation:
  ∃ R ∈ SO3. ∀ n m. n ≠ m ⟶ ((R^^n) ‘ bad-set-D) ∩ ((R^^m) ‘ bad-set-D) =
{}
proof −
  obtain a where a-sphere: a ∈ sphere2 and a-not-D: a ∉ bad-set-D
    and neg-a-not-D: − a ∉ bad-set-D
    using exists-antipodal-axis-avoiding-countable[OF bad-set-D-countable] by blast
  have norm-eq: norm e3-3 = norm a
    using e3-3-in-sphere2 a-sphere unfolding sphere2-def by simp
  have card3: 2 ≤ CARD(3)
    by simp
  obtain S where S-orth: orthogonal-transformation S
    and S-det: det (matrix S) = 1
    and S-e3: S e3-3 = a
```

```
    using rotation-exists[OF card3 norm-eq] by blast
  have S-SO3: S ∈ SO3
    using S-orth S-det unfolding SO3-def rotation-def by simp
  have invS-SO3: Hilbert-Choice.inv S ∈ SO3
    by (rule SO3-inverse[OF S-SO3])
  have S-bij: bij S
    by (rule orthogonal-transformation-bij[OF S-orth])
  have S-inj: inj S
    using S-bij by (simp add: bij-is-inj)
  have S-surj: surj S
    using S-bij by (simp add: bij-is-surj)
  have S-lin: linear S
    by (rule orthogonal-transformation-linear[OF S-orth])
  let ?D' = Hilbert-Choice.inv S ‘ bad-set-D
  have count-D': countable ?D'
    using bad-set-D-countable by simp
  have nonaxis-D': x $ 1 ≠ 0 ∨ x $ 2 ≠ 0 if xD: x ∈ ?D' for x
  proof (rule ccontr)
    assume ¬ (x $ 1 ≠ 0 ∨ x $ 2 ≠ 0)
    hence x1: x $ 1 = 0 and x2: x $ 2 = 0
      by auto
   from xD obtain d where dD: d ∈ bad-set-D and x-def: x = Hilbert-Choice.inv
S d
      by auto
    have d-sphere: d ∈ sphere2
      using dD bad-set-D-subset by auto
    have x-sphere: x ∈ sphere2
      unfolding x-def
      by (rule rotation-preserves-sphere2[OF invS-SO3 d-sphere])
    have axis-cases: x = e3-3 ∨ x = − e3-3
      by (rule z-axis-sphere2-cases[OF x-sphere x1 x2])
    have Sx-d: S x = d
      using x-def S-surj by (simp add: surj-f-inv-f)
    from axis-cases show False
    proof
      assume x = e3-3
      hence d = a
        using Sx-d S-e3 by simp
      with dD a-not-D show False
        by simp
    next
      assume x = − e3-3
      hence S x = − a
        using S-lin S-e3 by (simp add: linear-neg)
      hence d = − a
        using Sx-d by simp
      with dD neg-a-not-D show False
        by simp
    qed
```

```
  qed
  obtain t where t-disj:
    ∀ n m. n ≠ m ⟶
      ((Rz-angle t ^^ n) ‘ ?D′) ∩ ((Rz-angle t ^^ m) ‘ ?D′) = {}
    using exists-Rz-angle-absorbing[OF count-D′ nonaxis-D′] by blast
  let ?R = S ∘ Rz-angle t ∘ Hilbert-Choice.inv S
  have inner-SO3: Rz-angle t ∘ Hilbert-Choice.inv S ∈ SO3
    by (rule SO3-closed-compose[OF Rz-angle-in-SO3 invS-SO3])
  have R-SO3: ?R ∈ SO3
  proof −
    have S ∘ (Rz-angle t ∘ Hilbert-Choice.inv S) ∈ SO3
      by (rule SO3-closed-compose[OF S-SO3 inner-SO3])
    thus ?thesis
      by (simp add: o-assoc)
  qed
  have R-disj: ((?R ^^ n) ‘ bad-set-D) ∩ ((?R ^^ m) ‘ bad-set-D) = {}
    if nm: n ≠ m for n m
  proof −
    have n-img: ((?R ^^ n) ‘ bad-set-D) =
        S ‘ ((Rz-angle t ^^ n) ‘ ?D′)
      by (rule conjugate-funpow-image[OF S-bij])
    have m-img: ((?R ^^ m) ‘ bad-set-D) =
        S ‘ ((Rz-angle t ^^ m) ‘ ?D′)
      by (rule conjugate-funpow-image[OF S-bij])
    have z-disj: ((Rz-angle t ^^ n) ‘ ?D′) ∩ ((Rz-angle t ^^ m) ‘ ?D′) = {}
      using t-disj nm by blast
    show ?thesis
      unfolding n-img m-img
      using z-disj S-inj by (auto dest: injD)
  qed
  from R-SO3 R-disj show ?thesis
    by blast
qed

lemma SO3-funpow-preserves-sphere2:
  assumes R ∈ SO3 and x ∈ sphere2
  shows (R ^^ n) x ∈ sphere2
  by (rule rotation-preserves-sphere2[OF SO3-funpow[OF assms(1)] assms(2)])

lemma funpow-Suc-image:
  ((R ^^ Suc n) ‘ S) = R ‘ ((R ^^ n) ‘ S)
  by auto

lemma absorbing-rotation-shift:
  assumes disj: ∀ n m. n ≠ m ⟶ ((R^^n) ‘ D) ∩ ((R^^m) ‘ D) = {}
  defines E ≡ (⋃ n. (R^^n) ‘ D)
  shows R ‘ E = E − D
proof
  show R ‘ E ⊆ E − D
```

```
  proof
    fix y
    assume y ∈ R ‘ E
    then obtain x n d where x: x = (R ^^ n) d and d: d ∈ D and y: y = R x
      unfolding E-def by blast
    have y-pow: y ∈ (R ^^ Suc n) ‘ D
      using x d y by auto
    hence y ∈ E
      unfolding E-def by blast
    moreover have y ∉ D
    proof
      assume y ∈ D
      hence y ∈ (R ^^ 0) ‘ D
        by simp
      with y-pow have ((R ^^ Suc n) ‘ D) ∩ ((R ^^ 0) ‘ D) ≠ {}
        by blast
      moreover have Suc n ≠ 0
        by simp
      ultimately show False
        using disj by blast
    qed
    ultimately show y ∈ E − D
      by simp
  qed
next
  show E − D ⊆ R ‘ E
  proof
    fix y
    assume y: y ∈ E − D
    then obtain n d where y-pow: y = (R ^^ n) d and d: d ∈ D
      unfolding E-def by blast
    from y have y-not-D: y ∉ D
      by simp
    show y ∈ R ‘ E
    proof (cases n)
      case 0
      with y-pow d y-not-D show ?thesis
        by simp
    next
      case (Suc k)
      have (R ^^ k) d ∈ E
        unfolding E-def using d by blast
      moreover have y = R ((R ^^ k) d)
        using y-pow Suc by simp
      ultimately show ?thesis
        by blast
    qed
  qed
qed
```

```
lemma sphere2-absorb-bad-set:
  ∃ P Q :: (real^3) set list. ∃ gs :: ((real^3) ⇒ (real^3)) list.
      length P = length Q ∧ length P = length gs ∧
      set gs ⊆ SO3 ∧
      sphere2 = (⋃ i<length P. P ! i) ∧
      (sphere2 − bad-set-D) = (⋃ i<length Q. Q ! i) ∧
      (∀ i < length P. Q ! i = (gs ! i) ‘ (P ! i))
proof −
  obtain R where R-SO3: R ∈ SO3
    and disj: ∀ n m. n ≠ m ⟶ ((R^^n) ‘ bad-set-D) ∩ ((R^^m) ‘ bad-set-D) =
{}
    using exists-absorbing-rotation by auto
  define E where E = (⋃ n. (R^^n) ‘ bad-set-D)
  have R-E: R ‘ E = E − bad-set-D
    unfolding E-def by (rule absorbing-rotation-shift[OF disj])
  have D-subset-E: bad-set-D ⊆ E
  proof
    fix x
    assume x: x ∈ bad-set-D
    have x ∈ (R ^^ 0) ‘ bad-set-D
      using x by (auto simp add: image-def)
    thus x ∈ E
      unfolding E-def by blast
  qed
  have E-subset-sphere2: E ⊆ sphere2
  proof
    fix x
    assume x ∈ E
    then obtain n d where x: x = (R ^^ n) d and d: d ∈ bad-set-D
      unfolding E-def by blast
    from d have d ∈ sphere2
      using bad-set-D-subset by blast
    with R-SO3 x show x ∈ sphere2
      using SO3-funpow-preserves-sphere2 by blast
  qed
  let ?P = [E, sphere2 − E]
  let ?Q = [R ‘ E, sphere2 − E]
  let ?gs = [R, id]
  have P-union: sphere2 = (⋃ i<length ?P. ?P ! i)
  proof
    show sphere2 ⊆ (⋃ i<length ?P. ?P ! i)
    proof
      fix x
      assume x: x ∈ sphere2
      show x ∈ (⋃ i<length ?P. ?P ! i)
      proof (cases x ∈ E)
        case True
        then show ?thesis
```

```
      by force
    next
      case False
      with x show ?thesis
        by force
    qed
  qed
next
  show (⋃ i<length ?P. ?P ! i) ⊆ sphere2
  proof
    fix x
    assume x: x ∈ (⋃ i<length ?P. ?P ! i)
    then obtain i where i: i < length ?P and xi: x ∈ ?P ! i
      by blast
    have i = 0 ∨ i = 1
      using i by auto
    thus x ∈ sphere2
    proof
      assume i = 0
      with xi E-subset-sphere2 show ?thesis
        by auto
    next
      assume i = 1
      with xi show ?thesis
        by auto
    qed
  qed
qed
have Q-union: sphere2 − bad-set-D = (⋃ i<length ?Q. ?Q ! i)
proof
  show sphere2 − bad-set-D ⊆ (⋃ i<length ?Q. ?Q ! i)
  proof
    fix x
    assume x: x ∈ sphere2 − bad-set-D
    show x ∈ (⋃ i<length ?Q. ?Q ! i)
    proof (cases x ∈ E)
      case True
      with x R-E show ?thesis
        by force
    next
      case False
      with x show ?thesis
        by force
    qed
  qed
next
  show (⋃ i<length ?Q. ?Q ! i) ⊆ sphere2 − bad-set-D
  proof
    fix x
```

```
    assume x: x ∈ (⋃ i<length ?Q. ?Q ! i)
    then obtain i where i: i < length ?Q and xi: x ∈ ?Q ! i
      by blast
    have i = 0 ∨ i = 1
      using i by auto
    thus x ∈ sphere2 − bad-set-D
    proof
      assume i = 0
      with xi R-E E-subset-sphere2 show ?thesis
        by auto
    next
      assume i = 1
      with xi D-subset-E show ?thesis
        by auto
    qed
  qed
  have image-eq: ∀ i < length ?P. ?Q ! i = (?gs ! i) ‘ (?P ! i)
  proof (intro allI impI)
    fix i
    assume i: i < length ?P
    have i = 0 ∨ i = 1
      using i by auto
    thus ?Q ! i = (?gs ! i) ‘ (?P ! i)
      by auto
  qed
  show ?thesis
    apply (rule exI[of - ?P])
    apply (rule exI[of - ?Q])
    apply (rule exI[of - ?gs])
    using R-SO3 id-in-SO3 P-union Q-union image-eq
    by auto
qed
```

Combining the Hausdorff paradox with the absorption argument yields the paradoxical decomposition of the full sphere.

```
lemma sphere-indexed-union-append-singleton:
  fixes P :: 'a set list and S :: 'a set
  shows (⋃ i<length (P @ [S]). (P @ [S]) ! i) =
    (⋃ i<length P. P ! i) ∪ S
proof −
  have (⋃ i<length (P @ [S]). (P @ [S]) ! i) =
      (⋃ i<Suc (length P). (P @ [S]) ! i)
    by simp
  also have ... =
      (⋃ i<length P. (P @ [S]) ! i) ∪ (P @ [S]) ! length P
    by (simp add: lessThan-Suc Un-commute)
  also have ... = (⋃ i<length P. P ! i) ∪ S
  proof −
```

```
    have (⋃i<length P. (P @ [S]) ! i) = (⋃i<length P. P ! i)
      by (rule SUP-cong) (simp-all add: nth-append)
    thus ?thesis by simp
  qed
  finally show ?thesis .
qed

lemma sphere-indexed-image-union-append-singleton:
  fixes P :: 'a set list and G :: ('a ⇒ 'b) list and S :: 'a set
    and g :: 'a ⇒ 'b
  assumes length P = length G
  shows (⋃i<length (P @ [S]). (G @ [g]) ! i ‘ ((P @ [S]) ! i)) =
    (⋃i<length P. G ! i ‘ (P ! i)) ∪ g ‘ S
proof −
  have (⋃i<length (P @ [S]). (G @ [g]) ! i ‘ ((P @ [S]) ! i)) =
      (⋃i<Suc (length P). (G @ [g]) ! i ‘ ((P @ [S]) ! i))
    by simp
  also have ... =
      (⋃i<length P. (G @ [g]) ! i ‘ ((P @ [S]) ! i)) ∪
      (G @ [g]) ! length P ‘ ((P @ [S]) ! length P)
    by (simp add: lessThan-Suc Un-commute)
  also have ... = (⋃i<length P. G ! i ‘ (P ! i)) ∪ g ‘ S
    using assms by (simp add: nth-append)
  finally show ?thesis .
qed

lemma sphere2-absorb-cover:
  assumes R-SO3: R ∈ SO3
    and disj: ∀ n m. n ≠ m ⟶ ((R^^n) ‘ bad-set-D) ∩ ((R^^m) ‘ bad-set-D) =
{}
    and len: length P = length g
    and g-SO3: set g ⊆ SO3
    and P-sub: ∀ i < length P. P ! i ⊆ sphere2 − bad-set-D
    and X-cover: sphere2 − bad-set-D = (⋃i<length P. (g ! i) ‘ (P ! i))
  shows ∃ P' :: (real^3) set list. ∃ g' :: ((real^3) ⇒ (real^3)) list.
      length P' = length g' ∧
      set g' ⊆ SO3 ∧
      (∀ i < length P'. P' ! i ⊆ sphere2) ∧
      sphere2 = (⋃i<length P'. P' ! i) ∧
      sphere2 = (⋃i<length P'. (g' ! i) ‘ (P' ! i))
proof −
  define E where E = (⋃n. (R^^n) ‘ bad-set-D)
  have R-E: R ‘ E = E − bad-set-D
    unfolding E-def by (rule absorbing-rotation-shift[OF disj])
  have D-subset-E: bad-set-D ⊆ E
  proof
    fix x
    assume x: x ∈ bad-set-D
    have x ∈ (R ^^ 0) ‘ bad-set-D
```

```
    using x by (auto simp add: image-def)
  thus x ∈ E
    unfolding E-def by blast
qed
have E-subset-sphere2: E ⊆ sphere2
proof
  fix x
  assume x ∈ E
  then obtain n d where x: x = (R ^^ n) d and d: d ∈ bad-set-D
    unfolding E-def by blast
  from d have d ∈ sphere2
    using bad-set-D-subset by blast
  with R-SO3 x show x ∈ sphere2
    using SO3-funpow-preserves-sphere2 by blast
qed
have invR-SO3: Hilbert-Choice.inv R ∈ SO3
  by (rule SO3-inverse[OF R-SO3])
have R-inj: inj R
  by (rule SO3-inj[OF R-SO3])
define P0 where P0 = P @ P
define g0 where g0 = map (λh. Hilbert-Choice.inv R ∘ h) g @ g
define Rest where Rest = sphere2 − (⋃ i<length P0. P0 ! i)
define P' where P' = P0 @ [Rest]
define g' where g' = g0 @ [id]
have len0: length P0 = length g0
  using len by (simp add: P0-def g0-def)
have len': length P' = length g'
  using len0 by (simp add: P'-def g'-def)
have g0-SO3: set g0 ⊆ SO3
proof
  fix h
  assume h ∈ set g0
  hence h ∈ (λk. Hilbert-Choice.inv R ∘ k) ` set g ∨ h ∈ set g
    by (auto simp: g0-def)
  thus h ∈ SO3
  proof
    assume h ∈ (λk. Hilbert-Choice.inv R ∘ k) ` set g
    then obtain k where k: k ∈ set g and h: h = Hilbert-Choice.inv R ∘ k
      by blast
    have k-SO3: k ∈ SO3
      using g-SO3 k by auto
    show ?thesis
      unfolding h by (rule SO3-closed-compose[OF invR-SO3 k-SO3])
  next
    assume h ∈ set g
    thus ?thesis
      using g-SO3 by auto
  qed
qed
```

have *g'-SO3*: *set g'* ⊆ *SO3*
using *g0-SO3 id-in-SO3* **by** (*auto simp*: *g'-def*)
have *P0-sub*: ∀ *i* < *length P0*. *P0* ! *i* ⊆ *sphere2*
proof (*intro allI impI*)
fix *i*
assume *i*: *i* < *length P0*
show *P0* ! *i* ⊆ *sphere2*
proof (*cases i* < *length P*)
case *True*
have *P0-i*: *P0* ! *i* = *P* ! *i*
using *True* **by** (*simp add*: *P0-def nth-append*)
have *P* ! *i* ⊆ *sphere2*
using *P-sub True* **by** *blast*
thus *?thesis*
by (*simp add*: *P0-i*)
next
case *False*
hence *i2*: *i* − *length P* < *length P*
using *i* **by** (*simp add*: *P0-def*)
have *P0* ! *i* = *P* ! (*i* − *length P*)
using *False i* **by** (*simp add*: *P0-def nth-append*)
with *P-sub i2* **show** *?thesis*
by *auto*
qed
qed
have *P'-sub*: ∀ *i* < *length P'*. *P'* ! *i* ⊆ *sphere2*
proof (*intro allI impI*)
fix *i*
assume *i*: *i* < *length P'*
show *P'* ! *i* ⊆ *sphere2*
proof (*cases i* < *length P0*)
case *True*
have *P'-i*: *P'* ! *i* = *P0* ! *i*
using *True* **by** (*simp add*: *P'-def nth-append*)
have *P0* ! *i* ⊆ *sphere2*
using *P0-sub True* **by** *blast*
thus *?thesis*
by (*simp add*: *P'-i*)
next
case *False*
with *i* **have** *i* = *length P0*
by (*simp add*: *P'-def*)
thus *?thesis*
by (*auto simp*: *P'-def Rest-def*)
qed
qed
have *P'-cover*: *sphere2* = (⋃ *i*<*length P'*. *P'* ! *i*)
proof −
have (⋃ *i*<*length P'*. *P'* ! *i*) =

```
    (⋃ i<length P0. P0 ! i) ∪ Rest
   unfolding P'-def by (rule sphere-indexed-union-append-singleton)
  also have ... = sphere2
  proof
   show (⋃ i<length P0. P0 ! i) ∪ Rest ⊆ sphere2
    using P0-sub by (auto simp: Rest-def)
   show sphere2 ⊆ (⋃ i<length P0. P0 ! i) ∪ Rest
    by (auto simp: Rest-def)
  qed
  finally show ?thesis
   by simp
qed
have image0-cover: sphere2 ⊆ (⋃ i<length P0. (g0 ! i) ‘ (P0 ! i))
proof
  fix y
  assume y-sphere: y ∈ sphere2
  show y ∈ (⋃ i<length P0. (g0 ! i) ‘ (P0 ! i))
  proof (cases y ∈ E)
   case True
   hence Ry-in-RE: R y ∈ R ‘ E
    by blast
   have Ry-E-minus-D: R y ∈ E − bad-set-D
    using Ry-in-RE R-E by simp
   have Ry-X: R y ∈ sphere2 − bad-set-D
   proof
    show R y ∈ sphere2
     using Ry-E-minus-D E-subset-sphere2 by blast
    show R y ∉ bad-set-D
     using Ry-E-minus-D by simp
   qed
   then obtain i x where i: i < length P and x: x ∈ P ! i
    and Ry: R y = (g ! i) x
    using X-cover by blast
   have i0: i < length P0
    using i by (simp add: P0-def)
   have g0-i: g0 ! i = Hilbert-Choice.inv R ∘ (g ! i)
    using i len by (simp add: g0-def nth-append)
   have P0-i: P0 ! i = P ! i
    using i by (simp add: P0-def nth-append)
   have (g0 ! i) x = y
   proof −
    have inv-Ry: Hilbert-Choice.inv R (R y) = y
     by (rule inv-f-f[OF R-inj])
    have (g0 ! i) x = Hilbert-Choice.inv R ((g ! i) x)
     by (simp add: g0-i)
    also have ... = Hilbert-Choice.inv R (R y)
     using Ry by simp
    also have ... = y
     by (rule inv-Ry)
```

```
    finally show ?thesis .
  qed
  moreover have (g0 ! i) x ∈ (g0 ! i) ‘ (P0 ! i)
    using x P0-i by blast
  ultimately show ?thesis
    using i0 by blast
next
  case False
  have y-X: y ∈ sphere2 − bad-set-D
  proof
    show y ∈ sphere2
      by (rule y-sphere)
    show y ∉ bad-set-D
      using False D-subset-E by auto
  qed
  then obtain i x where i: i < length P and x: x ∈ P ! i
    and y-eq: y = (g ! i) x
    using X-cover by blast
  let ?j = length P + i
  have j0: ?j < length P0
    using i by (simp add: P0-def)
  have g0-j: g0 ! ?j = g ! i
    using i len by (simp add: g0-def nth-append)
  have P0-j: P0 ! ?j = P ! i
    using i by (simp add: P0-def nth-append)
  show ?thesis
  proof −
    have (g0 ! ?j) x ∈ (g0 ! ?j) ‘ (P0 ! ?j)
      using x P0-j by blast
    moreover have (g0 ! ?j) x = y
      using y-eq g0-j by simp
    ultimately show ?thesis
      using j0 by blast
  qed
qed
qed
have image0-subset: (⋃ i<length P0. (g0 ! i) ‘ (P0 ! i)) ⊆ sphere2
proof
  fix y
  assume y ∈ (⋃ i<length P0. (g0 ! i) ‘ (P0 ! i))
  then obtain i x where i: i < length P0 and x: x ∈ P0 ! i
    and y: y = (g0 ! i) x
    by blast
  have g0-i: g0 ! i ∈ SO3
  proof −
    have g0 ! i ∈ set g0
      using i len0 by (simp add: nth-mem)
    thus ?thesis
      using g0-SO3 by blast
```

```
    qed
    have x-sphere: x ∈ sphere2
      using P0-sub i x by auto
    show y ∈ sphere2
      using rotation-preserves-sphere2[OF g0-i x-sphere] y by simp
  qed
  have image0-eq: sphere2 = (⋃ i<length P0. (g0 ! i) ‘ (P0 ! i))
  proof
    show sphere2 ⊆ (⋃ i<length P0. (g0 ! i) ‘ (P0 ! i))
      by (rule image0-cover)
    show (⋃ i<length P0. (g0 ! i) ‘ (P0 ! i)) ⊆ sphere2
      by (rule image0-subset)
  qed
  have image-cover: sphere2 = (⋃ i<length P'. (g' ! i) ‘ (P' ! i))
  proof −
    have (⋃ i<length P'. (g' ! i) ‘ (P' ! i)) =
        (⋃ i<length P0. (g0 ! i) ‘ (P0 ! i)) ∪ id ‘ Rest
      unfolding P'-def g'-def
      by (rule sphere-indexed-image-union-append-singleton[OF len0])
    also have ... = sphere2
    proof −
      have id ‘ Rest ⊆ sphere2
        by (auto simp: Rest-def)
      with image0-eq show ?thesis
        by blast
    qed
    finally show ?thesis
      by simp
  qed
  show ?thesis
  proof (intro exI conjI)
    show length P' = length g'
      by (rule len')
    show set g' ⊆ SO3
      by (rule g'-SO3)
    show ∀ i<length P'. P' ! i ⊆ sphere2
      by (rule P'-sub)
    show sphere2 = (⋃ i<length P'. P' ! i)
      by (rule P'-cover)
    show sphere2 = (⋃ i<length P'. (g' ! i) ‘ (P' ! i))
      by (rule image-cover)
  qed
qed

theorem sphere2-paradoxical-strong:
  ∃ P Q :: (real^3) set list. ∃ gP gQ :: ((real^3) ⇒ (real^3)) list.
      length P = length gP ∧ length Q = length gQ ∧
      set gP ⊆ SO3 ∧ set gQ ⊆ SO3 ∧
      pairwise-disjoint (P @ Q) ∧
```

pairwise-disjoint (*map2* (λ*g A*. *g* ‘ *A*) *gP P*) ∧
pairwise-disjoint (*map2* (λ*g A*. *g* ‘ *A*) *gQ Q*) ∧
(∀ *i* < *length P*. *P* ! *i* ⊆ *sphere2*) ∧
(∀ *i* < *length Q*. *Q* ! *i* ⊆ *sphere2*) ∧
sphere2 = (⋃ *i*<*length P*. *P* ! *i*) ∪ (⋃ *i*<*length Q*. *Q* ! *i*) ∧
sphere2 = (⋃ *i*<*length P*. (*gP* ! *i*) ‘ (*P* ! *i*)) ∧
sphere2 = (⋃ *i*<*length Q*. (*gQ* ! *i*) ‘ (*Q* ! *i*))
proof −
obtain *R* **where** *R-SO3*: *R* ∈ *SO3*
and *disj*: ∀ *n m*. *n* ≠ *m* ⟶ ((*R* ^^ *n*) ‘ *bad-set-D*) ∩ ((*R* ^^ *m*) ‘ *bad-set-D*) = {}
using *exists-absorbing-rotation* **by** *blast*
define *E* **where** *E* = (⋃ *n*. (*R* ^^ *n*) ‘ *bad-set-D*)
have *R-E*: *R* ‘ *E* = *E* − *bad-set-D*
unfolding *E-def* **by** (*rule absorbing-rotation-shift*[*OF disj*])
have *D-subset-E*: *bad-set-D* ⊆ *E*
proof
fix *x*
assume *x*: *x* ∈ *bad-set-D*
have *x* ∈ (*R* ^^ *0*) ‘ *bad-set-D*
using *x* **by** (*auto simp add*: *image-def*)
thus *x* ∈ *E*
unfolding *E-def* **by** *blast*
qed
have *E-subset-sphere2*: *E* ⊆ *sphere2*
proof
fix *x*
assume *x* ∈ *E*
then obtain *n d* **where** *x*: *x* = (*R* ^^ *n*) *d* **and** *d*: *d* ∈ *bad-set-D*
unfolding *E-def* **by** *blast*
from *d* **have** *d* ∈ *sphere2*
using *bad-set-D-subset* **by** *blast*
with *R-SO3 x* **show** *x* ∈ *sphere2*
using *SO3-funpow-preserves-sphere2* **by** *blast*
qed
have *invR-SO3*: *Hilbert-Choice.inv R* ∈ *SO3*
by (*rule SO3-inverse*[*OF R-SO3*])
have *R-inj*: *inj R*
by (*rule SO3-inj*[*OF R-SO3*])
let *?X* = *sphere2* − *bad-set-D*
let *?Y* = *sphere2*
let *?A* = [*E*, *sphere2* − *E*]
let *?B* = [*R* ‘ *E*, *sphere2* − *E*]
let *?e* = [*R*, *id*]
let *?t* = [*Hilbert-Choice.inv R*, *id*]
have *A-cover*: *?Y* = (⋃ *k*<*length ?A*. *?A* ! *k*)
using *E-subset-sphere2* **by** (*auto simp*: *lessThan-Suc*)
have *A-disj*: *pairwise-disjoint ?A*
unfolding *pairwise-disjoint-def*

```
proof (intro allI impI)
  fix i j
  assume i: i < length ?A and j: j < length ?A and ij: i ≠ j
  have (i = 0 ∧ j = 1) ∨ (i = 1 ∧ j = 0)
    using i j ij by auto
  thus ?A ! i ∩ ?A ! j = {}
    by auto
qed
have B-cover: ?X = (⋃k<length ?B. ?B ! k)
proof
  show ?X ⊆ (⋃k<length ?B. ?B ! k)
  proof
    fix x
    assume x: x ∈ ?X
    show x ∈ (⋃k<length ?B. ?B ! k)
    proof (cases x ∈ E)
      case True
      with x R-E show ?thesis
        by force
    next
      case False
      with x show ?thesis
        by force
    qed
  qed
  show (⋃k<length ?B. ?B ! k) ⊆ ?X
  proof
    fix x
    assume x: x ∈ (⋃k<length ?B. ?B ! k)
    then obtain k where k: k < length ?B and xk: x ∈ ?B ! k
      by blast
    have k = 0 ∨ k = 1
      using k by auto
    thus x ∈ ?X
    proof
      assume k = 0
      with xk R-E E-subset-sphere2 show ?thesis
        by auto
    next
      assume k = 1
      with xk D-subset-E show ?thesis
        by auto
    qed
  qed
qed
have B-disj: pairwise-disjoint ?B
  unfolding pairwise-disjoint-def
proof (intro allI impI)
  fix i j
```

```
  assume i: i < length ?B and j: j < length ?B and ij: i ≠ j
  have (i = 0 ∧ j = 1) ∨ (i = 1 ∧ j = 0)
    using i j ij by auto
  thus ?B ! i ∩ ?B ! j = {}
    using R-E by auto
qed
have e-left: ⋀k x. ⟦k < length ?A; x ∈ ?A ! k⟧
    ⟹ (?e ! k) x ∈ ?B ! k ∧ (?t ! k) ((?e ! k) x) = x
proof −
  fix k x
  assume k: k < length ?A and x: x ∈ ?A ! k
  have k = 0 ∨ k = 1
    using k by auto
  thus (?e ! k) x ∈ ?B ! k ∧ (?t ! k) ((?e ! k) x) = x
  proof
    assume k = 0
    with x show ?thesis
      using inv-f-f[OF R-inj, of x] by auto
  next
    assume k = 1
    with x show ?thesis
      by auto
  qed
qed
have t-left: ⋀k y. ⟦k < length ?B; y ∈ ?B ! k⟧
    ⟹ (?t ! k) y ∈ ?A ! k ∧ (?e ! k) ((?t ! k) y) = y
proof −
  fix k y
  assume k: k < length ?B and y: y ∈ ?B ! k
  have k = 0 ∨ k = 1
    using k by auto
  thus (?t ! k) y ∈ ?A ! k ∧ (?e ! k) ((?t ! k) y) = y
  proof
    assume k = 0
    then obtain x where x: x ∈ E and y-eq: y = R x
      using y by auto
    have inv-eq: Hilbert-Choice.inv R y = x
      using y-eq inv-f-f[OF R-inj, of x] by simp
    with x y-eq ‹k = 0› show ?thesis
      by simp
  next
    assume k = 1
    with y show ?thesis
      by auto
  qed
qed

from hausdorff-paradox-rot-strong obtain P Q :: (real^3) set list
  and gP gQ :: ((real^3) ⇒ (real^3)) list
```

```
  where lenP: length P = length gP
    and lenQ: length Q = length gQ
    and gP-SO3: set gP ⊆ SO3
    and gQ-SO3: set gQ ⊆ SO3
    and source-disj: pairwise-disjoint (P @ Q)
    and imageP-disj: pairwise-disjoint (map2 (λg A. g ‘ A) gP P)
    and imageQ-disj: pairwise-disjoint (map2 (λg A. g ‘ A) gQ Q)
    and P-sub: ∀ i < length P. P ! i ⊆ ?X
    and Q-sub: ∀ i < length Q. Q ! i ⊆ ?X
    and source-cover: ?X = (⋃ i<length P. P ! i) ∪ (⋃ i<length Q. Q ! i)
    and imageP-cover: ?X = (⋃ i<length P. (gP ! i) ‘ (P ! i))
    and imageQ-cover: ?X = (⋃ i<length Q. (gQ ! i) ‘ (Q ! i))
  by auto
have gP-inj: ⋀i. i < length P ⟹ inj (gP ! i)
proof −
  fix i assume i: i < length P
  have gP ! i ∈ SO3
  proof −
    have i < length gP
      using lenP i by simp
    hence gP ! i ∈ set gP
      by (rule nth-mem)
    thus ?thesis
      using gP-SO3 by blast
  qed
  thus inj (gP ! i)
    by (rule SO3-inj)
qed
have gQ-inj: ⋀i. i < length Q ⟹ inj (gQ ! i)
proof −
  fix i assume i: i < length Q
  have gQ ! i ∈ SO3
  proof −
    have i < length gQ
      using lenQ i by simp
    hence gQ ! i ∈ set gQ
      by (rule nth-mem)
    thus ?thesis
      using gQ-SO3 by blast
  qed
  thus inj (gQ ! i)
    by (rule SO3-inj)
qed
have mapsP: ⋀k i l. ⟦k < length ?A; i < length P; l < length ?B⟧
    ⟹ transfer-map ?t gP ?e k i l ∈ SO3
proof −
  fix k i l
  assume k: k < length ?A and i: i < length P and l: l < length ?B
  have gi: gP ! i ∈ SO3
```

```
  proof −
    have i < length gP
      using lenP i by simp
    hence gP ! i ∈ set gP
      by (rule nth-mem)
    thus ?thesis
      using gP-SO3 by blast
  qed
  have ek: ?e ! k ∈ SO3
  proof −
    have k = 0 ∨ k = 1
      using k by auto
    thus ?thesis
      using R-SO3 id-in-SO3 by auto
  qed
  have tl: ?t ! l ∈ SO3
  proof −
    have l = 0 ∨ l = 1
      using l by auto
    thus ?thesis
      using invR-SO3 id-in-SO3 by auto
  qed
  have left: (?t ! l) ∘ (gP ! i) ∈ SO3
    by (rule SO3-closed-compose[OF tl gi])
  show transfer-map ?t gP ?e k i l ∈ SO3
    unfolding transfer-map-def
    by (rule SO3-closed-compose[OF left ek])
qed
have mapsQ: ⋀k i l. ⟦k < length ?A; i < length Q; l < length ?B⟧
  ⟹ transfer-map ?t gQ ?e k i l ∈ SO3
proof −
  fix k i l
  assume k: k < length ?A and i: i < length Q and l: l < length ?B
  have gi: gQ ! i ∈ SO3
  proof −
    have i < length gQ
      using lenQ i by simp
    hence gQ ! i ∈ set gQ
      by (rule nth-mem)
    thus ?thesis
      using gQ-SO3 by blast
  qed
  have ek: ?e ! k ∈ SO3
  proof −
    have k = 0 ∨ k = 1
      using k by auto
    thus ?thesis
      using R-SO3 id-in-SO3 by auto
  qed
```

```
    have tl: ?t ! l ∈ SO3
    proof −
      have l = 0 ∨ l = 1
        using l by auto
      thus ?thesis
        using invR-SO3 id-in-SO3 by auto
    qed
    have left: (?t ! l) ∘ (gQ ! i) ∈ SO3
      by (rule SO3-closed-compose[OF tl gi])
    show transfer-map ?t gQ ?e k i l ∈ SO3
      unfolding transfer-map-def
      by (rule SO3-closed-compose[OF left ek])
  qed
  have lenAB: length ?A = length ?B
    by simp
  from transfer-partitioned-paradox[
      OF lenAB A-cover A-disj B-cover B-disj e-left t-left lenP lenQ source-disj
         source-cover imageP-disj imageQ-disj imageP-cover imageQ-cover gP-inj
gQ-inj
       mapsP mapsQ]
  obtain P' Q' :: (real^3) set list and hP hQ :: ((real^3) ⇒ (real^3)) list
    where lenP': length P' = length hP
      and lenQ': length Q' = length hQ
      and hP-SO3: set hP ⊆ SO3
      and hQ-SO3: set hQ ⊆ SO3
      and source-disj': pairwise-disjoint (P' @ Q')
      and imageP-disj': pairwise-disjoint (map2 (λg A. g ‘ A) hP P')
      and imageQ-disj': pairwise-disjoint (map2 (λg A. g ‘ A) hQ Q')
      and source-cover': ?Y = (⋃i<length P'. P' ! i) ∪ (⋃i<length Q'. Q' ! i)
      and imageP-cover': ?Y = (⋃i<length P'. (hP ! i) ‘ (P' ! i))
      and imageQ-cover': ?Y = (⋃i<length Q'. (hQ ! i) ‘ (Q' ! i))
    by auto
  have P'-sub: ∀ i < length P'. P' ! i ⊆ sphere2
    using source-cover' by blast
  have Q'-sub: ∀ i < length Q'. Q' ! i ⊆ sphere2
    using source-cover' by blast
  show ?thesis
  proof (intro exI conjI)
    show length P' = length hP
      by (rule lenP')
    show length Q' = length hQ
      by (rule lenQ')
    show set hP ⊆ SO3
      by (rule hP-SO3)
    show set hQ ⊆ SO3
      by (rule hQ-SO3)
    show pairwise-disjoint (P' @ Q')
      by (rule source-disj')
    show pairwise-disjoint (map2 (λg A. g ‘ A) hP P')
```

by (*rule imageP-disj'*)
show *pairwise-disjoint* (*map2* (λ*g A*. *g* ‘ *A*) *hQ Q'*)
by (*rule imageQ-disj'*)
show ∀ *i*<*length P'*. *P'* ! *i* ⊆ *sphere2*
by (*rule P'-sub*)
show ∀ *i*<*length Q'*. *Q'* ! *i* ⊆ *sphere2*
by (*rule Q'-sub*)
show *sphere2* = (⋃ *i*<*length P'*. *P'* ! *i*) ∪ (⋃ *i*<*length Q'*. *Q'* ! *i*)
by (*rule source-cover'*)
show *sphere2* = (⋃ *i*<*length P'*. (*hP* ! *i*) ‘ (*P'* ! *i*))
by (*rule imageP-cover'*)
show *sphere2* = (⋃ *i*<*length Q'*. (*hQ* ! *i*) ‘ (*Q'* ! *i*))
by (*rule imageQ-cover'*)
qed
qed

theorem *sphere2-paradoxical*:
∃ *P Q* :: (*real^3*) *set list*. ∃ *gP gQ* :: ((*real^3*) ⇒ (*real^3*)) *list*.
length P = *length gP* ∧ *length Q* = *length gQ* ∧
set gP ⊆ *SO3* ∧ *set gQ* ⊆ *SO3* ∧
(∀ *i* < *length P*. *P* ! *i* ⊆ *sphere2*) ∧
(∀ *i* < *length Q*. *Q* ! *i* ⊆ *sphere2*) ∧
sphere2 = (⋃ *i*<*length P*. *P* ! *i*) ∧
sphere2 = (⋃ *i*<*length Q*. *Q* ! *i*) ∧
sphere2 = (⋃ *i*<*length P*. (*gP* ! *i*) ‘ (*P* ! *i*)) ∧
sphere2 = (⋃ *i*<*length Q*. (*gQ* ! *i*) ‘ (*Q* ! *i*))
proof −
obtain *R* **where** *R-SO3*: *R* ∈ *SO3*
and *disj*: ∀ *n m*. *n* ≠ *m* ⟶ ((*R*^^*n*) ‘ *bad-set-D*) ∩ ((*R*^^*m*) ‘ *bad-set-D*) = {}
using *exists-absorbing-rotation* **by** *blast*
from *hausdorff-paradox* **obtain** *P Q* :: (*real^3*) *set list*
and *hP hQ* :: ((*real^3*) ⇒ (*real^3*)) *list*
where *hd*:
length P = *length hP* ∧ *length Q* = *length hQ* ∧
set hP ⊆ *SO3* ∧ *set hQ* ⊆ *SO3* ∧
(∀ *i* < *length P*. *P* ! *i* ⊆ *sphere2* − *bad-set-D*) ∧
(∀ *i* < *length Q*. *Q* ! *i* ⊆ *sphere2* − *bad-set-D*) ∧
sphere2 − *bad-set-D* = (⋃ *i*<*length P*. (*hP* ! *i*) ‘ (*P* ! *i*)) ∧
sphere2 − *bad-set-D* = (⋃ *i*<*length Q*. (*hQ* ! *i*) ‘ (*Q* ! *i*))
by (*elim exE*)
from *hd* **have** *lenP*: *length P* = *length hP*
by (*elim conjE*)
from *hd* **have** *lenQ*: *length Q* = *length hQ*
by (*elim conjE*)
from *hd* **have** *hP-SO3*: *set hP* ⊆ *SO3*
by (*elim conjE*)
from *hd* **have** *hQ-SO3*: *set hQ* ⊆ *SO3*
by (*elim conjE*)

from *hd* **have** *P-sub*: $\forall\, i < \mathit{length}\ P.\ P\ !\ i \subseteq \mathit{sphere2} - \mathit{bad\text{-}set\text{-}D}$
by (*elim conjE*)
from *hd* **have** *Q-sub*: $\forall\, i < \mathit{length}\ Q.\ Q\ !\ i \subseteq \mathit{sphere2} - \mathit{bad\text{-}set\text{-}D}$
by (*elim conjE*)
from *hd* **have** *coverP*: $\mathit{sphere2} - \mathit{bad\text{-}set\text{-}D} = (\bigcup i{<}\mathit{length}\ P.\ (hP\ !\ i)\ `\ (P\ !\ i))$
by (*elim conjE*)
from *hd* **have** *coverQ*: $\mathit{sphere2} - \mathit{bad\text{-}set\text{-}D} = (\bigcup i{<}\mathit{length}\ Q.\ (hQ\ !\ i)\ `\ (Q\ !\ i))$
by (*elim conjE*)
from *sphere2-absorb-cover*[*OF R-SO3 disj lenP hP-SO3 P-sub coverP*]
obtain P' hP' **where** P'*-all*:
$\mathit{length}\ P' = \mathit{length}\ hP' \wedge$
$\mathit{set}\ hP' \subseteq \mathit{SO3} \wedge$
$(\forall\, i < \mathit{length}\ P'.\ P'\ !\ i \subseteq \mathit{sphere2}) \wedge$
$\mathit{sphere2} = (\bigcup i{<}\mathit{length}\ P'.\ P'\ !\ i) \wedge$
$\mathit{sphere2} = (\bigcup i{<}\mathit{length}\ P'.\ (hP'\ !\ i)\ `\ (P'\ !\ i))$
by (*elim exE*)
from P'*-all* **have** P'*-len*: $\mathit{length}\ P' = \mathit{length}\ hP'$
by (*elim conjE*)
from P'*-all* **have** P'*-SO3*: $\mathit{set}\ hP' \subseteq \mathit{SO3}$
by (*elim conjE*)
from P'*-all* **have** P'*-sub*: $\forall\, i < \mathit{length}\ P'.\ P'\ !\ i \subseteq \mathit{sphere2}$
by (*elim conjE*)
from P'*-all* **have** P'*-src*: $\mathit{sphere2} = (\bigcup i{<}\mathit{length}\ P'.\ P'\ !\ i)$
by (*elim conjE*)
from P'*-all* **have** P'*-img*: $\mathit{sphere2} = (\bigcup i{<}\mathit{length}\ P'.\ (hP'\ !\ i)\ `\ (P'\ !\ i))$
by (*elim conjE*)
from *sphere2-absorb-cover*[*OF R-SO3 disj lenQ hQ-SO3 Q-sub coverQ*]
obtain Q' hQ' **where** Q'*-all*:
$\mathit{length}\ Q' = \mathit{length}\ hQ' \wedge$
$\mathit{set}\ hQ' \subseteq \mathit{SO3} \wedge$
$(\forall\, i < \mathit{length}\ Q'.\ Q'\ !\ i \subseteq \mathit{sphere2}) \wedge$
$\mathit{sphere2} = (\bigcup i{<}\mathit{length}\ Q'.\ Q'\ !\ i) \wedge$
$\mathit{sphere2} = (\bigcup i{<}\mathit{length}\ Q'.\ (hQ'\ !\ i)\ `\ (Q'\ !\ i))$
by (*elim exE*)
from Q'*-all* **have** Q'*-len*: $\mathit{length}\ Q' = \mathit{length}\ hQ'$
by (*elim conjE*)
from Q'*-all* **have** Q'*-SO3*: $\mathit{set}\ hQ' \subseteq \mathit{SO3}$
by (*elim conjE*)
from Q'*-all* **have** Q'*-sub*: $\forall\, i < \mathit{length}\ Q'.\ Q'\ !\ i \subseteq \mathit{sphere2}$
by (*elim conjE*)
from Q'*-all* **have** Q'*-src*: $\mathit{sphere2} = (\bigcup i{<}\mathit{length}\ Q'.\ Q'\ !\ i)$
by (*elim conjE*)
from Q'*-all* **have** Q'*-img*: $\mathit{sphere2} = (\bigcup i{<}\mathit{length}\ Q'.\ (hQ'\ !\ i)\ `\ (Q'\ !\ i))$
by (*elim conjE*)
show *?thesis*
proof (*intro exI conjI*)
show $\mathit{length}\ P' = \mathit{length}\ hP'$
by (*rule P'-len*)
show $\mathit{length}\ Q' = \mathit{length}\ hQ'$

```
    by (rule Q'-len)
  show set hP' ⊆ SO3
    by (rule P'-SO3)
  show set hQ' ⊆ SO3
    by (rule Q'-SO3)
  show ∀ i<length P'. P' ! i ⊆ sphere2
    by (rule P'-sub)
  show ∀ i<length Q'. Q' ! i ⊆ sphere2
    by (rule Q'-sub)
  show sphere2 = (⋃ i<length P'. P' ! i)
    by (rule P'-src)
  show sphere2 = (⋃ i<length Q'. Q' ! i)
    by (rule Q'-src)
  show sphere2 = (⋃ i<length P'. (hP' ! i) ` (P' ! i))
    by (rule P'-img)
  show sphere2 = (⋃ i<length Q'. (hQ' ! i) ` (Q' ! i))
    by (rule Q'-img)
 qed
qed

end


theory Ball-Decomposition
 imports Sphere-Decomposition
begin
```

Radial cone construction: take a sphere subset and form the open radial cone in the ball.

```
definition cone-of :: (real^3) set ⇒ (real^3) set where
  cone-of S = {t *R x | t x. 0 < t ∧ t ≤ 1 ∧ x ∈ S}

lemma cone-of-in-ball3:
 assumes S ⊆ sphere2
 shows cone-of S ⊆ ball3
proof
 fix y assume y ∈ cone-of S
 then obtain t x where y-eq: y = t *R x and t-pos: 0 < t and t-le: t ≤ 1
   and x-in: x ∈ S
   unfolding cone-of-def by auto
 from x-in assms have x-sphere: x ∈ sphere2 by auto
 hence x-norm: norm x = 1 by (simp add: sphere2-def)
 have norm y = norm (t *R x) by (simp add: y-eq)
 also have ... = |t| * norm x by simp
 also have ... = t using t-pos x-norm by simp
 also have ... ≤ 1 using t-le .
 finally have norm y ≤ 1 .
 thus y ∈ ball3 by (simp add: ball3-def)
qed
```

lemma *cone-of-in-ball3-minus-origin*:
assumes $S \subseteq$ *sphere2*
shows *cone-of* $S \subseteq$ *ball3* $- \{0\}$
proof
fix y **assume** *y-in*: $y \in$ *cone-of* S
then obtain t x **where** *y-eq*: $y = t *_R x$ **and** *t-pos*: $0 < t$ **and** *t-le*: $t \leq 1$
and *x-in*: $x \in S$
unfolding *cone-of-def* **by** *auto*
from *cone-of-in-ball3*[*OF assms*] *y-in* **have** *y-ball*: $y \in$ *ball3*
by *blast*
from *x-in assms* **have** *x-sphere*: $x \in$ *sphere2*
by *auto*
hence *x-norm*: *norm* $x = 1$
by (*simp add*: *sphere2-def*)
have *norm* $y =$ *norm* $(t *_R x)$
by (*simp add*: *y-eq*)
also have $\ldots = t$
using *t-pos x-norm* **by** *simp*
finally have *norm* $y > 0$
using *t-pos* **by** *simp*
hence $y \neq 0$
by *auto*
with *y-ball* **show** $y \in$ *ball3* $- \{0\}$
by *simp*
qed

lemma *ball3-minus-origin-eq-cone-sphere2*:
ball3 $- \{0$::*real^3*$\} =$ *cone-of sphere2*
proof
show *ball3* $- \{0$::*real^3*$\} \subseteq$ *cone-of sphere2*
proof
fix y **assume** *y-in*: $y \in$ *ball3* $- \{0$::*real^3*$\}$
hence *y-ball*: $y \in$ *ball3* **and** *y-ne*: $y \neq 0$
by *auto*
define t **where** $t =$ *norm* y
define x **where** $x = (1 \ / \ t) *_R y$
have *t-pos*: $0 < t$
using *y-ne* **by** (*simp add*: *t-def*)
have *t-le*: $t \leq 1$
using *y-ball* **by** (*simp add*: *ball3-def t-def*)
have *x-sphere*: $x \in$ *sphere2*
proof $-$
have *norm* $x =$ *norm* $((1 \ / \ t) *_R y)$
by (*simp add*: *x-def*)
also have $\ldots = 1$
using *t-pos* **by** (*simp add*: *t-def*)
finally show *?thesis*
by (*simp add*: *sphere2-def*)

```
    qed
    have y-eq: y = t *R x
      using t-pos by (simp add: x-def)
    show y ∈ cone-of sphere2
      unfolding cone-of-def
      using y-eq t-pos t-le x-sphere by blast
  qed
  show cone-of sphere2 ⊆ ball3 - {0::real^3}
    by (rule cone-of-in-ball3-minus-origin) simp
qed

lemma cone-of-UN:
  cone-of (⋃ i∈I. S i) = (⋃ i∈I. cone-of (S i))
  unfolding cone-of-def by blast

lemma SO3-image-cone-of:
  assumes g ∈ SO3
  shows g ‘ cone-of S = cone-of (g ‘ S)
proof
  show g ‘ cone-of S ⊆ cone-of (g ‘ S)
  proof
    fix y assume y ∈ g ‘ cone-of S
    then obtain x where y-eq: y = g x and x-in: x ∈ cone-of S
      by blast
    then obtain t z where x-eq: x = t *R z and t-pos: 0 < t and t-le: t ≤ 1
      and z-in: z ∈ S
      unfolding cone-of-def by blast
    have y = t *R g z
    using assms by (simp add: y-eq x-eq SO3-def rotation-def orthogonal-transformation-scaleR)
    thus y ∈ cone-of (g ‘ S)
      unfolding cone-of-def using t-pos t-le z-in by blast
  qed
  show cone-of (g ‘ S) ⊆ g ‘ cone-of S
  proof
    fix y assume y ∈ cone-of (g ‘ S)
    then obtain t z where y-eq: y = t *R z and t-pos: 0 < t and t-le: t ≤ 1
      and z-in: z ∈ g ‘ S
      unfolding cone-of-def by blast
    then obtain x where x-in: x ∈ S and z-eq: z = g x
      by blast
    have y = g (t *R x)
    using assms by (simp add: y-eq z-eq SO3-def rotation-def orthogonal-transformation-scaleR)
    moreover have t *R x ∈ cone-of S
      unfolding cone-of-def using t-pos t-le x-in by blast
    ultimately show y ∈ g ‘ cone-of S
      by blast
  qed
qed
```

lemma *cone-of-disjoint*:
assumes *S-sub*: $S \subseteq$ *sphere2*
and *T-sub*: $T \subseteq$ *sphere2*
and *disj*: $S \cap T = \{\}$
shows *cone-of* $S \cap$ *cone-of* $T = \{\}$
proof
show *cone-of* $S \cap$ *cone-of* $T \subseteq \{\}$
proof
fix y
assume y: $y \in$ *cone-of* $S \cap$ *cone-of* T
then obtain t x **where** *y-tx*: $y = t *_R x$ **and** *t-pos*: $0 < t$
and *xS*: $x \in S$
unfolding *cone-of-def* **by** *blast*
from y **obtain** s z **where** *y-sz*: $y = s *_R z$ **and** *s-pos*: $0 < s$
and *zT*: $z \in T$
unfolding *cone-of-def* **by** *blast*
have *x-norm*: *norm* $x = 1$
using *S-sub sphere2-def xS* **by** *auto*
have *z-norm*: *norm* $z = 1$
using *T-sub sphere2-def zT* **by** *auto*
have *t-eq*: $t = s$
proof −
have $t =$ *norm* $(t *_R x)$
using *t-pos x-norm* **by** *simp*
also have $\ldots =$ *norm* $(s *_R z)$
using *y-tx y-sz* **by** *simp*
also have $\ldots = s$
using *s-pos z-norm* **by** *simp*
finally show *?thesis* **.**
qed
have $x = z$
using *y-tx y-sz t-eq t-pos* **by** (*simp add*: *scaleR-cancel-left*)
hence $x \in S \cap T$
using *xS zT* **by** *simp*
with *disj* **show** $y \in \{\}$
by *simp*
qed
show $\{\} \subseteq$ *cone-of* $S \cap$ *cone-of* T
by *simp*
qed

lemma *cone-of-pairwise-disjoint*:
assumes *disj*: *pairwise-disjoint* P
and *sub*: $\forall i <$ *length* P. $P ! i \subseteq$ *sphere2*
shows *pairwise-disjoint* (*map cone-of* P)
unfolding *pairwise-disjoint-def*
proof (*intro allI impI*)
fix i j
assume i: $i <$ *length* (*map cone-of* P) **and** j: $j <$ *length* (*map cone-of* P) **and**

ij: $i \neq j$
have $P\ !\ i \cap P\ !\ j = \{\}$
using *disj i j ij* **by** (*simp add*: *pairwise-disjoint-def*)
moreover have $P\ !\ i \subseteq$ *sphere2* $P\ !\ j \subseteq$ *sphere2*
using *sub i j* **by** *auto*
ultimately show *map cone-of* $P\ !\ i \cap$ *map cone-of* $P\ !\ j = \{\}$
using *i j* **by** (*simp add*: *cone-of-disjoint*)
qed

lemma *SO3-cone-images-pairwise-disjoint*:
assumes *len*: *length* $P =$ *length g*
and *g-SO3*: *set g* $\subseteq$ *SO3*
and *P-sub*: $\forall\, i <$ *length P*. $P\ !\ i \subseteq$ *sphere2*
and *disj*: *pairwise-disjoint* (*map2* ($\lambda h\ A.\ h$ ‘ A) *g P*)
shows *pairwise-disjoint* (*map2* ($\lambda h\ A.\ h$ ‘ A) *g* (*map cone-of P*))
unfolding *pairwise-disjoint-def*
proof (*intro allI impI*)
fix *i j*
assume *i*: $i <$ *length* (*map2* ($\lambda h\ A.\ h$ ‘ A) *g* (*map cone-of P*))
and *j*: $j <$ *length* (*map2* ($\lambda h\ A.\ h$ ‘ A) *g* (*map cone-of P*))
and *ij*: $i \neq j$
have *iP*: $i <$ *length P* **and** *jP*: $j <$ *length P*
using *i j len* **by** *auto*
have *gi*: $g\ !\ i \in$ *SO3* **and** *gj*: $g\ !\ j \in$ *SO3*
proof −
have $i <$ *length g*
using *iP len* **by** *simp*
hence $g\ !\ i \in$ *set g*
by (*rule nth-mem*)
thus $g\ !\ i \in$ *SO3*
using *g-SO3* **by** *blast*
next
have $j <$ *length g*
using *jP len* **by** *simp*
hence $g\ !\ j \in$ *set g*
by (*rule nth-mem*)
thus $g\ !\ j \in$ *SO3*
using *g-SO3* **by** *blast*
qed
have *img-disj*: $(g\ !\ i$ ‘ $(P\ !\ i)) \cap (g\ !\ j$ ‘ $(P\ !\ j)) = \{\}$
using *disj len iP jP ij* **by** (*simp add*: *pairwise-disjoint-def*)
have *img-i-sub*: $g\ !\ i$ ‘ $(P\ !\ i) \subseteq$ *sphere2*
using *P-sub iP rotation-preserves-sphere2*[*OF gi*] **by** *auto*
have *img-j-sub*: $g\ !\ j$ ‘ $(P\ !\ j) \subseteq$ *sphere2*
using *P-sub jP rotation-preserves-sphere2*[*OF gj*] **by** *auto*
have *cone-disj*: *cone-of* $(g\ !\ i$ ‘ $(P\ !\ i)) \cap$ *cone-of* $(g\ !\ j$ ‘ $(P\ !\ j)) = \{\}$
by (*rule cone-of-disjoint*[*OF img-i-sub img-j-sub img-disj*])
have *map2* ($\lambda h\ A.\ h$ ‘ A) *g* (*map cone-of P*) ! *i* =
cone-of $(g\ !\ i$ ‘ $(P\ !\ i))$

using *len iP SO3-image-cone-of*[*OF gi*, *of P ! i*] **by** *simp*
moreover have *map2* (λ*h A*. *h ‘ A*) *g* (*map cone-of P*) ! *j* =
cone-of (*g ! j ‘* (*P ! j*))
using *len jP SO3-image-cone-of*[*OF gj*, *of P ! j*] **by** *simp*
ultimately show *map2* (λ*h A*. *h ‘ A*) *g* (*map cone-of P*) ! *i* ∩
map2 (λ*h A*. *h ‘ A*) *g* (*map cone-of P*) ! *j* = {}
using *cone-disj* **by** *simp*
qed

lemma *zero-in-ball3* [*simp*]: (*0*::*real^3*) ∈ *ball3*
by (*simp add*: *ball3-def*)

lemma *indexed-image-union-append-singleton*:
fixes *P* :: *'a set list* **and** *G* :: (*'a* ⇒ *'b*) *list* **and** *S* :: *'a set*
and *g* :: *'a* ⇒ *'b*
assumes *length P* = *length G*
shows (⋃*i*<*length* (*P* @ [*S*]). (*G* @ [*g*]) ! *i ‘* ((*P* @ [*S*]) ! *i*)) =
(⋃*i*<*length P*. *G* ! *i ‘* (*P* ! *i*)) ∪ *g ‘ S*
proof −
have (⋃*i*<*length* (*P* @ [*S*]). (*G* @ [*g*]) ! *i ‘* ((*P* @ [*S*]) ! *i*)) =
(⋃*i*<*Suc* (*length P*). (*G* @ [*g*]) ! *i ‘* ((*P* @ [*S*]) ! *i*))
by *simp*
also have ... =
(⋃*i*<*length P*. (*G* @ [*g*]) ! *i ‘* ((*P* @ [*S*]) ! *i*)) ∪
(*G* @ [*g*]) ! *length P ‘* ((*P* @ [*S*]) ! *length P*)
by (*simp add*: *lessThan-Suc Un-commute*)
also have ... = (⋃*i*<*length P*. *G* ! *i ‘* (*P* ! *i*)) ∪ *g ‘ S*
using *assms* **by** (*simp add*: *nth-append*)
finally show *?thesis* **.**
qed

lemma *indexed-union-append-singleton*:
fixes *P* :: *'a set list* **and** *S* :: *'a set*
shows (⋃*i*<*length* (*P* @ [*S*]). (*P* @ [*S*]) ! *i*) =
(⋃*i*<*length P*. *P* ! *i*) ∪ *S*
proof −
have (⋃*i*<*length* (*P* @ [*S*]). (*P* @ [*S*]) ! *i*) =
(⋃*i*<*Suc* (*length P*). (*P* @ [*S*]) ! *i*)
by *simp*
also have ... =
(⋃*i*<*length P*. (*P* @ [*S*]) ! *i*) ∪ (*P* @ [*S*]) ! *length P*
by (*simp add*: *lessThan-Suc Un-commute*)
finally show *?thesis*
by (*metis sphere-indexed-union-append-singleton*)
qed

The punctured ball is paradoxical via radial cones over the sphere pieces.

theorem *ball3-minus-origin-paradoxical*:
∃ *P Q* :: (*real^3*) *set list*. ∃ *gP gQ* :: ((*real^3*) ⇒ (*real^3*)) *list*.

$length\ P = length\ gP \wedge length\ Q = length\ gQ \wedge$
$set\ gP \subseteq SO3 \wedge set\ gQ \subseteq SO3 \wedge$
$(\forall\, i < length\ P.\ P\ !\ i \subseteq ball3 - \{0\}) \wedge$
$(\forall\, i < length\ Q.\ Q\ !\ i \subseteq ball3 - \{0\}) \wedge$
$ball3 - \{0\} = (\bigcup i{<}length\ P.\ P\ !\ i) \wedge$
$ball3 - \{0\} = (\bigcup i{<}length\ Q.\ Q\ !\ i) \wedge$
$ball3 - \{0\} = (\bigcup i{<}length\ P.\ (gP\ !\ i)\ `\ (P\ !\ i)) \wedge$
$ball3 - \{0\} = (\bigcup i{<}length\ Q.\ (gQ\ !\ i)\ `\ (Q\ !\ i))$

proof −
from *sphere2-paradoxical* **obtain** $P\ Q :: (real\hat{}3)\ set\ list$
and $gP\ gQ :: ((real\hat{}3 \Rightarrow real\hat{}3))\ list$
where *decomp*:
$length\ P = length\ gP \wedge length\ Q = length\ gQ \wedge$
$set\ gP \subseteq SO3 \wedge set\ gQ \subseteq SO3 \wedge$
$(\forall\, i < length\ P.\ P\ !\ i \subseteq sphere2) \wedge$
$(\forall\, i < length\ Q.\ Q\ !\ i \subseteq sphere2) \wedge$
$sphere2 = (\bigcup i{<}length\ P.\ P\ !\ i) \wedge$
$sphere2 = (\bigcup i{<}length\ Q.\ Q\ !\ i) \wedge$
$sphere2 = (\bigcup i{<}length\ P.\ (gP\ !\ i)\ `\ (P\ !\ i)) \wedge$
$sphere2 = (\bigcup i{<}length\ Q.\ (gQ\ !\ i)\ `\ (Q\ !\ i))$
by (*elim exE*)
from *decomp* **have** *lenP*: $length\ P = length\ gP$
by (*elim conjE*)
from *decomp* **have** *lenQ*: $length\ Q = length\ gQ$
by (*elim conjE*)
from *decomp* **have** *gP-SO3*: $set\ gP \subseteq SO3$
by (*elim conjE*)
from *decomp* **have** *gQ-SO3*: $set\ gQ \subseteq SO3$
by (*elim conjE*)
from *decomp* **have** *P-sub-sphere*: $\forall\, i < length\ P.\ P\ !\ i \subseteq sphere2$
by (*elim conjE*)
from *decomp* **have** *Q-sub-sphere*: $\forall\, i < length\ Q.\ Q\ !\ i \subseteq sphere2$
by (*elim conjE*)
from *decomp* **have** *P-src-sphere*: $sphere2 = (\bigcup i{<}length\ P.\ P\ !\ i)$
by (*elim conjE*)
from *decomp* **have** *Q-src-sphere*: $sphere2 = (\bigcup i{<}length\ Q.\ Q\ !\ i)$
by (*elim conjE*)
from *decomp* **have** *P-cov-sphere*: $sphere2 = (\bigcup i{<}length\ P.\ (gP\ !\ i)\ `\ (P\ !\ i))$
by (*elim conjE*)
from *decomp* **have** *Q-cov-sphere*: $sphere2 = (\bigcup i{<}length\ Q.\ (gQ\ !\ i)\ `\ (Q\ !\ i))$
by (*elim conjE*)

define P' **where** $P' = map\ cone\text{-}of\ P$
define Q' **where** $Q' = map\ cone\text{-}of\ Q$
have *lenP'*: $length\ P' = length\ gP$
by (*simp add*: P'-*def lenP*)
have *lenQ'*: $length\ Q' = length\ gQ$
by (*simp add*: Q'-*def lenQ*)
have P'-*sub*: $\forall\, i < length\ P'.\ P'\ !\ i \subseteq ball3 - \{0\}$

using *P-sub-sphere* **by** (*simp add*: *P'-def cone-of-in-ball3-minus-origin*)
have *Q'-sub*: $\forall\, i < \mathit{length}\ Q'.\ Q' ! i \subseteq \mathit{ball3} - \{0\}$
using *Q-sub-sphere* **by** (*simp add*: *Q'-def cone-of-in-ball3-minus-origin*)
have *P'-src*: $\mathit{ball3} - \{0\} = (\bigcup i{<}\mathit{length}\ P'.\ P' ! i)$
proof −
have $(\bigcup i{<}\mathit{length}\ P'.\ P' ! i) = (\bigcup i{<}\mathit{length}\ P.\ \mathit{cone\text{-}of}\ (P ! i))$
by (*simp add*: *P'-def*)
also have $\ldots = \mathit{cone\text{-}of}\ (\bigcup i{<}\mathit{length}\ P.\ P ! i)$
by (*simp add*: *cone-of-UN*)
also have $\ldots = \mathit{cone\text{-}of\ sphere2}$
using *P-src-sphere* **by** *simp*
also have $\ldots = \mathit{ball3} - \{0\}$
using *ball3-minus-origin-eq-cone-sphere2* **by** *simp*
finally show *?thesis* **by** *simp*
qed
have *Q'-src*: $\mathit{ball3} - \{0\} = (\bigcup i{<}\mathit{length}\ Q'.\ Q' ! i)$
proof −
have $(\bigcup i{<}\mathit{length}\ Q'.\ Q' ! i) = (\bigcup i{<}\mathit{length}\ Q.\ \mathit{cone\text{-}of}\ (Q ! i))$
by (*simp add*: *Q'-def*)
also have $\ldots = \mathit{cone\text{-}of}\ (\bigcup i{<}\mathit{length}\ Q.\ Q ! i)$
by (*simp add*: *cone-of-UN*)
also have $\ldots = \mathit{cone\text{-}of\ sphere2}$
using *Q-src-sphere* **by** *simp*
also have $\ldots = \mathit{ball3} - \{0\}$
using *ball3-minus-origin-eq-cone-sphere2* **by** *simp*
finally show *?thesis* **by** *simp*
qed
have *P'-cov*: $\mathit{ball3} - \{0\} = (\bigcup i{<}\mathit{length}\ P'.\ (gP ! i)\ `\ (P' ! i))$
proof −
have $(\bigcup i{<}\mathit{length}\ P'.\ (gP ! i)\ `\ (P' ! i)) =$
$(\bigcup i{<}\mathit{length}\ P.\ \mathit{cone\text{-}of}\ ((gP ! i)\ `\ (P ! i)))$
proof −
have $\bigwedge i.\ i < \mathit{length}\ P \Longrightarrow (gP ! i)\ `\ \mathit{cone\text{-}of}\ (P ! i) =$
$\mathit{cone\text{-}of}\ ((gP ! i)\ `\ (P ! i))$
by (*metis SO3-image-cone-of gP-SO3 lenP nth-mem subset-eq*)
thus *?thesis*
by (*simp add*: *P'-def*)
qed
also have $\ldots = \mathit{cone\text{-}of}\ (\bigcup i{<}\mathit{length}\ P.\ (gP ! i)\ `\ (P ! i))$
by (*simp add*: *cone-of-UN*)
also have $\ldots = \mathit{cone\text{-}of\ sphere2}$
using *P-cov-sphere* **by** *simp*
also have $\ldots = \mathit{ball3} - \{0\}$
using *ball3-minus-origin-eq-cone-sphere2* **by** *simp*
finally show *?thesis* **by** *simp*
qed
have *Q'-cov*: $\mathit{ball3} - \{0\} = (\bigcup i{<}\mathit{length}\ Q'.\ (gQ ! i)\ `\ (Q' ! i))$
proof −
have $(\bigcup i{<}\mathit{length}\ Q'.\ (gQ ! i)\ `\ (Q' ! i)) =$

```
      (⋃i<length Q. cone-of ((gQ ! i) ‘ (Q ! i)))
    proof −
      have ⋀i. i < length Q ⟹ (gQ ! i) ‘ cone-of (Q ! i) =
          cone-of ((gQ ! i) ‘ (Q ! i))
      proof −
        fix i assume i-lt: i < length Q
        have gQ ! i ∈ set gQ
          using i-lt lenQ by (simp add: nth-mem)
        hence gQ ! i ∈ SO3
          using gQ-SO3 by blast
        thus (gQ ! i) ‘ cone-of (Q ! i) = cone-of ((gQ ! i) ‘ (Q ! i))
          by (rule SO3-image-cone-of)
      qed
      thus ?thesis
        by (simp add: Q'-def)
    qed
    also have ... = cone-of (⋃i<length Q. (gQ ! i) ‘ (Q ! i))
      by (simp add: cone-of-UN)
    also have ... = cone-of sphere2
      using Q-cov-sphere by simp
    also have ... = ball3 − {0}
      using ball3-minus-origin-eq-cone-sphere2 by simp
    finally show ?thesis by simp
  qed
  show ?thesis
    apply (rule exI[of - P'])
    apply (rule exI[of - Q'])
    apply (rule exI[of - gP])
    apply (rule exI[of - gQ])
    using lenP' lenQ' gP-SO3 gQ-SO3 P'-sub Q'-sub P'-src Q'-src P'-cov Q'-cov
    by simp
qed

theorem ball3-minus-origin-paradoxical-strong:
  ∃ P Q :: (real^3) set list. ∃ gP gQ :: ((real^3) ⇒ (real^3)) list.
      length P = length gP ∧ length Q = length gQ ∧
      set gP ⊆ SO3 ∧ set gQ ⊆ SO3 ∧
      pairwise-disjoint (P @ Q) ∧
      pairwise-disjoint (map2 (λg A. g ‘ A) gP P) ∧
      pairwise-disjoint (map2 (λg A. g ‘ A) gQ Q) ∧
      (∀ i < length P. P ! i ⊆ ball3 − {0}) ∧
      (∀ i < length Q. Q ! i ⊆ ball3 − {0}) ∧
      ball3 − {0} = (⋃i<length P. P ! i) ∪ (⋃i<length Q. Q ! i) ∧
      ball3 − {0} = (⋃i<length P. (gP ! i) ‘ (P ! i)) ∧
      ball3 − {0} = (⋃i<length Q. (gQ ! i) ‘ (Q ! i))
proof −
  from sphere2-paradoxical-strong obtain P Q :: (real^3) set list
    and gP gQ :: ((real^3 ⇒ real^3)) list
    where lenP: length P = length gP
```

and *lenQ*: *length Q* = *length gQ*
and *gP-SO3*: *set gP* ⊆ *SO3*
and *gQ-SO3*: *set gQ* ⊆ *SO3*
and *source-disj*: *pairwise-disjoint* (*P* @ *Q*)
and *imageP-disj*: *pairwise-disjoint* (*map2* (λ*g A*. *g* ‘ *A*) *gP P*)
and *imageQ-disj*: *pairwise-disjoint* (*map2* (λ*g A*. *g* ‘ *A*) *gQ Q*)
and *P-sub-sphere*: ∀ *i* < *length P*. *P* ! *i* ⊆ *sphere2*
and *Q-sub-sphere*: ∀ *i* < *length Q*. *Q* ! *i* ⊆ *sphere2*
and *source-cover*: *sphere2* = (⋃ *i*<*length P*. *P* ! *i*) ∪ (⋃ *i*<*length Q*. *Q* ! *i*)
and *P-cov-sphere*: *sphere2* = (⋃ *i*<*length P*. (*gP* ! *i*) ‘ (*P* ! *i*))
and *Q-cov-sphere*: *sphere2* = (⋃ *i*<*length Q*. (*gQ* ! *i*) ‘ (*Q* ! *i*))
by *auto*

define *P′* **where** *P′* = *map cone-of P*
define *Q′* **where** *Q′* = *map cone-of Q*
have *lenP′*: *length P′* = *length gP*
by (*simp add*: *P′-def lenP*)
have *lenQ′*: *length Q′* = *length gQ*
by (*simp add*: *Q′-def lenQ*)
have *P′-sub*: ∀ *i* < *length P′*. *P′* ! *i* ⊆ *ball3* − {*0*}
using *P-sub-sphere* **by** (*simp add*: *P′-def cone-of-in-ball3-minus-origin*)
have *Q′-sub*: ∀ *i* < *length Q′*. *Q′* ! *i* ⊆ *ball3* − {*0*}
using *Q-sub-sphere* **by** (*simp add*: *Q′-def cone-of-in-ball3-minus-origin*)
have *all-sub*: ∀ *i* < *length* (*P* @ *Q*). (*P* @ *Q*) ! *i* ⊆ *sphere2*
using *P-sub-sphere Q-sub-sphere* **by** (*auto simp*: *nth-append*)
have *source-disj′*: *pairwise-disjoint* (*P′* @ *Q′*)
unfolding *P′-def Q′-def*
using *cone-of-pairwise-disjoint*[*OF source-disj all-sub*] **by** *simp*
have *imageP-disj′*: *pairwise-disjoint* (*map2* (λ*g A*. *g* ‘ *A*) *gP P′*)
unfolding *P′-def*
by (*rule SO3-cone-images-pairwise-disjoint*[*OF lenP gP-SO3 P-sub-sphere imageP-disj*])
have *imageQ-disj′*: *pairwise-disjoint* (*map2* (λ*g A*. *g* ‘ *A*) *gQ Q′*)
unfolding *Q′-def*
by (*rule SO3-cone-images-pairwise-disjoint*[*OF lenQ gQ-SO3 Q-sub-sphere imageQ-disj*])

have *P′-src*: (⋃ *i*<*length P′*. *P′* ! *i*) = *cone-of* (⋃ *i*<*length P*. *P* ! *i*)
proof −
have (⋃ *i*<*length P′*. *P′* ! *i*) = (⋃ *i*<*length P*. *cone-of* (*P* ! *i*))
by (*simp add*: *P′-def*)
also have … = *cone-of* (⋃ *i*<*length P*. *P* ! *i*)
by (*simp add*: *cone-of-UN*)
finally show *?thesis* **.**
qed
have *Q′-src*: (⋃ *i*<*length Q′*. *Q′* ! *i*) = *cone-of* (⋃ *i*<*length Q*. *Q* ! *i*)
using *Q′-def cone-of-UN* **by** *fastforce*
have *source-cover′*: *ball3* − {*0*} = (⋃ *i*<*length P′*. *P′* ! *i*) ∪ (⋃ *i*<*length Q′*. *Q′* ! *i*)

proof −
  have $(\bigcup i{<}\mathit{length}\ P'.\ P'\ !\ i) \cup (\bigcup i{<}\mathit{length}\ Q'.\ Q'\ !\ i) =$
      $\mathit{cone\text{-}of}\ ((\bigcup i{<}\mathit{length}\ P.\ P\ !\ i) \cup (\bigcup i{<}\mathit{length}\ Q.\ Q\ !\ i))$
    using P'*-src* Q'*-src* **by** (*auto simp*: *cone-of-def*)
  also have $\ldots = \mathit{cone\text{-}of}\ \mathit{sphere2}$
    using *source-cover* **by** *simp*
  also have $\ldots = \mathit{ball3} - \{0\}$
    using *ball3-minus-origin-eq-cone-sphere2* **by** *simp*
  finally show *?thesis* **by** *simp*
qed

have P'*-cov*: $\mathit{ball3} - \{0\} = (\bigcup i{<}\mathit{length}\ P'.\ (gP\ !\ i)\ `\ (P'\ !\ i))$
proof −
  have $(\bigcup i{<}\mathit{length}\ P'.\ (gP\ !\ i)\ `\ (P'\ !\ i)) =$
      $(\bigcup i{<}\mathit{length}\ P.\ \mathit{cone\text{-}of}\ ((gP\ !\ i)\ `\ (P\ !\ i)))$
  proof −
    have $\bigwedge i.\ i < \mathit{length}\ P \Longrightarrow (gP\ !\ i)\ `\ \mathit{cone\text{-}of}\ (P\ !\ i) =$
        $\mathit{cone\text{-}of}\ ((gP\ !\ i)\ `\ (P\ !\ i))$
    proof −
      fix i **assume** *i-lt*: $i < \mathit{length}\ P$
      have $gP\ !\ i \in \mathit{SO3}$
        using *gP-SO3 i-lt lenP* **by** *auto*
      thus $(gP\ !\ i)\ `\ \mathit{cone\text{-}of}\ (P\ !\ i) = \mathit{cone\text{-}of}\ ((gP\ !\ i)\ `\ (P\ !\ i))$
        by (*rule SO3-image-cone-of*)
    qed
    thus *?thesis*
      by (*simp add*: P'*-def*)
  qed
  also have $\ldots = \mathit{cone\text{-}of}\ (\bigcup i{<}\mathit{length}\ P.\ (gP\ !\ i)\ `\ (P\ !\ i))$
    by (*simp add*: *cone-of-UN*)
  also have $\ldots = \mathit{cone\text{-}of}\ \mathit{sphere2}$
    using *P-cov-sphere* **by** *simp*
  also have $\ldots = \mathit{ball3} - \{0\}$
    using *ball3-minus-origin-eq-cone-sphere2* **by** *simp*
  finally show *?thesis* **by** *simp*
qed
have Q'*-cov*: $\mathit{ball3} - \{0\} = (\bigcup i{<}\mathit{length}\ Q'.\ (gQ\ !\ i)\ `\ (Q'\ !\ i))$
proof −
  have $(\bigcup i{<}\mathit{length}\ Q'.\ (gQ\ !\ i)\ `\ (Q'\ !\ i)) =$
      $(\bigcup i{<}\mathit{length}\ Q.\ \mathit{cone\text{-}of}\ ((gQ\ !\ i)\ `\ (Q\ !\ i)))$
  proof −
    have $\bigwedge i.\ i < \mathit{length}\ Q \Longrightarrow (gQ\ !\ i)\ `\ \mathit{cone\text{-}of}\ (Q\ !\ i) =$
        $\mathit{cone\text{-}of}\ ((gQ\ !\ i)\ `\ (Q\ !\ i))$
    proof −
      fix i **assume** *i-lt*: $i < \mathit{length}\ Q$
      have $gQ\ !\ i \in \mathit{SO3}$
        using *gQ-SO3 i-lt lenQ* **by** *auto*
      thus $(gQ\ !\ i)\ `\ \mathit{cone\text{-}of}\ (Q\ !\ i) = \mathit{cone\text{-}of}\ ((gQ\ !\ i)\ `\ (Q\ !\ i))$
        by (*rule SO3-image-cone-of*)

```
    qed
    thus ?thesis
      by (simp add: Q'-def)
  qed
  also have ... = cone-of (⋃ i<length Q. (gQ ! i) ‘ (Q ! i))
    by (simp add: cone-of-UN)
  also have ... = cone-of sphere2
    using Q-cov-sphere by simp
  also have ... = ball3 − {0}
    using ball3-minus-origin-eq-cone-sphere2 by simp
  finally show ?thesis by simp
qed
show ?thesis
proof (intro exI conjI)
  show length P' = length gP
    by (rule lenP')
  show length Q' = length gQ
    by (rule lenQ')
  show set gP ⊆ SO3
    by (rule gP-SO3)
  show set gQ ⊆ SO3
    by (rule gQ-SO3)
  show pairwise-disjoint (P' @ Q')
    by (rule source-disj')
  show pairwise-disjoint (map2 (λg A. g ‘ A) gP P')
    by (rule imageP-disj')
  show pairwise-disjoint (map2 (λg A. g ‘ A) gQ Q')
    by (rule imageQ-disj')
  show ∀ i<length P'. P' ! i ⊆ ball3 − {0}
    by (rule P'-sub)
  show ∀ i<length Q'. Q' ! i ⊆ ball3 − {0}
    by (rule Q'-sub)
  show ball3 − {0} = (⋃ i<length P'. P' ! i) ∪ (⋃ i<length Q'. Q' ! i)
    by (rule source-cover')
  show ball3 − {0} = (⋃ i<length P'. (gP ! i) ‘ (P' ! i))
    by (rule P'-cov)
  show ball3 − {0} = (⋃ i<length Q'. (gQ ! i) ‘ (Q' ! i))
    by (rule Q'-cov)
qed
qed
```

Adjoining the origin gives a useful SO(3)-only finite-cover decomposition of the full ball. The partition-level absorption is completed in the final theory, where off-centre isometries are available.

```
theorem ball3-paradoxical:
  ∃ P Q :: (real^3) set list. ∃ gP gQ :: ((real^3) ⇒ (real^3)) list.
      length P = length gP ∧ length Q = length gQ ∧
      set gP ⊆ SO3 ∧ set gQ ⊆ SO3 ∧
      (∀ i < length P. P ! i ⊆ ball3) ∧
```

$(\forall\, i < \mathit{length}\ Q.\ Q\ !\ i \subseteq \mathit{ball3}) \wedge$
$\mathit{ball3} = (\bigcup i{<}\mathit{length}\ P.\ P\ !\ i) \wedge$
$\mathit{ball3} = (\bigcup i{<}\mathit{length}\ Q.\ Q\ !\ i) \wedge$
$\mathit{ball3} = (\bigcup i{<}\mathit{length}\ P.\ (gP\ !\ i)\ `\ (P\ !\ i)) \wedge$
$\mathit{ball3} = (\bigcup i{<}\mathit{length}\ Q.\ (gQ\ !\ i)\ `\ (Q\ !\ i))$

proof −
from *ball3-minus-origin-paradoxical* **obtain** $P\ Q :: (\mathit{real\hat{}3})\ \mathit{set}\ \mathit{list}$
and $gP\ gQ :: ((\mathit{real\hat{}3} \Rightarrow \mathit{real\hat{}3}))\ \mathit{list}$
where *decomp*:
$\mathit{length}\ P = \mathit{length}\ gP \wedge \mathit{length}\ Q = \mathit{length}\ gQ \wedge$
$\mathit{set}\ gP \subseteq \mathit{SO3} \wedge \mathit{set}\ gQ \subseteq \mathit{SO3} \wedge$
$(\forall\, i < \mathit{length}\ P.\ P\ !\ i \subseteq \mathit{ball3} - \{0\}) \wedge$
$(\forall\, i < \mathit{length}\ Q.\ Q\ !\ i \subseteq \mathit{ball3} - \{0\}) \wedge$
$\mathit{ball3} - \{0\} = (\bigcup i{<}\mathit{length}\ P.\ P\ !\ i) \wedge$
$\mathit{ball3} - \{0\} = (\bigcup i{<}\mathit{length}\ Q.\ Q\ !\ i) \wedge$
$\mathit{ball3} - \{0\} = (\bigcup i{<}\mathit{length}\ P.\ (gP\ !\ i)\ `\ (P\ !\ i)) \wedge$
$\mathit{ball3} - \{0\} = (\bigcup i{<}\mathit{length}\ Q.\ (gQ\ !\ i)\ `\ (Q\ !\ i))$
by (*elim exE*)
from *decomp* **have** *lenP*: $\mathit{length}\ P = \mathit{length}\ gP$
by (*elim conjE*)
from *decomp* **have** *lenQ*: $\mathit{length}\ Q = \mathit{length}\ gQ$
by (*elim conjE*)
from *decomp* **have** *gP-SO3*: $\mathit{set}\ gP \subseteq \mathit{SO3}$
by (*elim conjE*)
from *decomp* **have** *gQ-SO3*: $\mathit{set}\ gQ \subseteq \mathit{SO3}$
by (*elim conjE*)
from *decomp* **have** *P-sub*: $\forall\, i < \mathit{length}\ P.\ P\ !\ i \subseteq \mathit{ball3} - \{0\}$
by (*elim conjE*)
from *decomp* **have** *Q-sub*: $\forall\, i < \mathit{length}\ Q.\ Q\ !\ i \subseteq \mathit{ball3} - \{0\}$
by (*elim conjE*)
from *decomp* **have** *P-src*: $\mathit{ball3} - \{0\} = (\bigcup i{<}\mathit{length}\ P.\ P\ !\ i)$
by (*elim conjE*)
from *decomp* **have** *Q-src*: $\mathit{ball3} - \{0\} = (\bigcup i{<}\mathit{length}\ Q.\ Q\ !\ i)$
by (*elim conjE*)
from *decomp* **have** *P-cov*: $\mathit{ball3} - \{0\} = (\bigcup i{<}\mathit{length}\ P.\ (gP\ !\ i)\ `\ (P\ !\ i))$
by (*elim conjE*)
from *decomp* **have** *Q-cov*: $\mathit{ball3} - \{0\} = (\bigcup i{<}\mathit{length}\ Q.\ (gQ\ !\ i)\ `\ (Q\ !\ i))$
by (*elim conjE*)
define P' **where** $P' = P\ @\ [\{0::\mathit{real\hat{}3}\}]$
define Q' **where** $Q' = Q\ @\ [\{0::\mathit{real\hat{}3}\}]$
define gP' **where** $gP' = gP\ @\ [\mathit{id}]$
define gQ' **where** $gQ' = gQ\ @\ [\mathit{id}]$
have *lenP'*: $\mathit{length}\ P' = \mathit{length}\ gP'$
by (*simp add*: *P'-def gP'-def lenP*)
have *lenQ'*: $\mathit{length}\ Q' = \mathit{length}\ gQ'$
by (*simp add*: *Q'-def gQ'-def lenQ*)
have *P'-sub*: $\forall\, i < \mathit{length}\ P'.\ P'\ !\ i \subseteq \mathit{ball3}$
using *P-sub* **by** (*auto simp add*: *P'-def nth-append*)
have *Q'-sub*: $\forall\, i < \mathit{length}\ Q'.\ Q'\ !\ i \subseteq \mathit{ball3}$

```
  using Q-sub by (auto simp add: Q'-def nth-append)
have gP'-SO3: set gP' ⊆ SO3
  using gP-SO3 by (simp add: gP'-def id-in-SO3)
have gQ'-SO3: set gQ' ⊆ SO3
  using gQ-SO3 by (simp add: gQ'-def id-in-SO3)
have P'-src: ball3 = (⋃i<length P'. P' ! i)
proof −
  have append-union:
    (⋃i<length (P @ [{0::real^3}]). (P @ [{0}]) ! i) =
      (⋃i<length P. P ! i) ∪ {0}
    using indexed-union-append-singleton[where P = P and S = {0::real^3}] .
  have (⋃i<length P'. P' ! i) = (ball3 − {0}) ∪ {0}
    unfolding P'-def
    using append-union P-src by simp
  also have ... = ball3
    by auto
  finally show ?thesis by simp
qed
have Q'-src: ball3 = (⋃i<length Q'. Q' ! i)
proof −
  have append-union:
    (⋃i<length (Q @ [{0::real^3}]). (Q @ [{0}]) ! i) =
      (⋃i<length Q. Q ! i) ∪ {0}
    using indexed-union-append-singleton[where P = Q and S = {0::real^3}] .
  have (⋃i<length Q'. Q' ! i) = (ball3 − {0}) ∪ {0}
    unfolding Q'-def
    using append-union Q-src by simp
  also have ... = ball3
    by auto
  finally show ?thesis by simp
qed
have P'-cov: ball3 = (⋃i<length P'. (gP' ! i) ‘ (P' ! i))
proof −
  have append-union:
    (⋃i<length (P @ [{0::real^3}]). (gP @ [id]) ! i ‘ ((P @ [{0}]) ! i)) =
      (⋃i<length P. gP ! i ‘ (P ! i)) ∪ {0}
    using indexed-image-union-append-singleton[
      where P = P and G = gP and S = {0::real^3} and g = id, OF lenP]
    by simp
  have union-eq: (⋃i<length P'. (gP' ! i) ‘ (P' ! i)) = ball3
  proof −
    have (⋃i<length P'. (gP' ! i) ‘ (P' ! i)) =
       (⋃i<length P. gP ! i ‘ (P ! i)) ∪ {0}
      unfolding P'-def gP'-def
      using append-union .
    also have ... = (ball3 − {0}) ∪ {0}
      using P-cov by simp
    also have ... = ball3
      by auto
```

```
    finally show ?thesis .
  qed
  thus ?thesis by simp
qed
have Q'-cov: ball3 = (⋃i<length Q'. (gQ' ! i) ‘ (Q' ! i))
proof −
  have append-union:
    (⋃i<length (Q @ [{0::real^3}]). (gQ @ [id]) ! i ‘ ((Q @ [{0}]) ! i)) =
      (⋃i<length Q. gQ ! i ‘ (Q ! i)) ∪ {0}
    using indexed-image-union-append-singleton[
      where P = Q and G = gQ and S = {0::real^3} and g = id, OF lenQ]
    by simp
  have union-eq: (⋃i<length Q'. (gQ' ! i) ‘ (Q' ! i)) = ball3
  proof −
    have (⋃i<length Q'. (gQ' ! i) ‘ (Q' ! i)) =
        (⋃i<length Q. gQ ! i ‘ (Q ! i)) ∪ {0}
      unfolding Q'-def gQ'-def
      using append-union .
    also have ... = (ball3 − {0}) ∪ {0}
      using Q-cov by simp
    also have ... = ball3
      by auto
    finally show ?thesis .
  qed
  thus ?thesis by simp
qed
show ?thesis
  apply (rule exI[of - P'])
  apply (rule exI[of - Q'])
  apply (rule exI[of - gP'])
  apply (rule exI[of - gQ'])
  using lenP' lenQ' gP'-SO3 gQ'-SO3 P'-sub Q'-sub P'-src Q'-src P'-cov Q'-cov
  by simp
qed

end


theory Banach-Tarski-Theorem
  imports Ball-Decomposition
begin
```

Two copies of the ball, embedded in disjoint regions of $\mathbb{R}^3$.

```
definition shift3 :: real^3 where
  shift3 = vector [3, 0, 0]

definition ball3-copy-left :: (real^3) set where
  ball3-copy-left = ball3
```

definition *ball3-copy-right* :: (*real^3*) *set* **where**
ball3-copy-right = (+) *shift3* ‘ *ball3*

lemma *norm-shift3* [*simp*]: *norm shift3* = *3*
unfolding *shift3-def*
by (*simp add*: *norm-eq-sqrt-inner inner-vec-def sum-3*)

lemma *ball3-copies-disjoint*:
ball3-copy-left ∩ *ball3-copy-right* = {}
proof
show *ball3-copy-left* ∩ *ball3-copy-right* ⊆ {}
proof
fix *z*
assume *z* ∈ *ball3-copy-left* ∩ *ball3-copy-right*
then obtain *y* **where** *z-ball*: *z* ∈ *ball3* **and** *y-ball*: *y* ∈ *ball3* **and** *z-eq*: *z* = *shift3* + *y*
unfolding *ball3-copy-left-def ball3-copy-right-def* **by** *auto*
have *z-norm*: *norm z* ≤ *1* **and** *y-norm*: *norm y* ≤ *1*
using *z-ball y-ball* **by** (*simp-all add*: *ball3-def*)
have *tri*: *norm* (*z* − *y*) ≤ *norm z* + *norm y*
using *norm-triangle-ineq4* **by** *blast*
have *3* = *norm* (*z* − *y*)
using *z-eq* **by** *simp*
also have ... ≤ *norm z* + *norm y*
by (*rule tri*)
also have ... ≤ *2*
using *z-norm y-norm* **by** *simp*
finally show *z* ∈ {}
by *simp*
qed
qed *simp*

Isometries of $\mathbb{R}^3$ are the motions used in the final assembly. Rotations from $SO(3)$, translations, and their compositions are isometries.

definition *isometry* :: (*real^3* ⇒ *real^3*) ⇒ *bool* **where**
isometry f ⟷ (∀ *x y*. *dist* (*f x*) (*f y*) = *dist x y*)

lemma *id-isometry* [*simp*]: *isometry id*
by (*simp add*: *isometry-def*)

lemma *translation-isometry* [*simp*]: *isometry* ((+) *a*)
by (*simp add*: *isometry-def dist-norm*)

lemma *SO3-isometry*:
assumes *g* ∈ *SO3*
shows *isometry g*
using *assms*
by (*auto simp add*: *SO3-def rotation-def isometry-def orthogonal-transformation-isometry*)

lemma *isometry-compose*:
assumes *isometry f isometry g*
shows *isometry* (*f* ∘ *g*)
using *assms* **by** (*simp add*: *isometry-def*)

lemma *translation-compose-SO3-isometry* [*simp*]:
assumes *g* ∈ *SO3*
shows *isometry* ((+) *a* ∘ *g*)
using *assms* **by** (*intro isometry-compose translation-isometry SO3-isometry*)

lemma *offcenter-rotation-isometry*:
assumes *R* ∈ *SO3*
shows *isometry* (λ*x*. *c* + *R* (*x* − *c*))
proof −
have (λ*x*. *c* + *R* (*x* − *c*)) = (+) *c* ∘ *R* ∘ (+) (− *c*)
by (*rule ext*) *simp*
thus *?thesis*
using *assms* **by** (*simp add*: *isometry-compose SO3-isometry*)
qed

lemma *offcenter-rotation-inverse-left*:
fixes *c* :: *real^3*
assumes *inv-left*: *Rinv* ∘ *R* = *id*
defines *T* ≡ λ*x*. *c* + *R* (*x* − *c*)
and *U* ≡ λ*x*. *c* + *Rinv* (*x* − *c*)
shows *U* (*T x*) = *x*
using *fun-cong*[*OF inv-left*, *of x* − *c*]
by (*simp add*: *T-def U-def*)

lemma *offcenter-rotation-inverse-right*:
fixes *c* :: *real^3*
assumes *inv-right*: *R* ∘ *Rinv* = *id*
defines *T* ≡ λ*x*. *c* + *R* (*x* − *c*)
and *U* ≡ λ*x*. *c* + *Rinv* (*x* − *c*)
shows *T* (*U x*) = *x*
using *fun-cong*[*OF inv-right*, *of x* − *c*]
by (*simp add*: *T-def U-def*)

lemma *offcenter-rotation-funpow-zero*:
fixes *c* :: *real^3* **and** *R* :: *real^3* ⇒ *real^3*
defines *T* ≡ λ*x*. *c* + *R* (*x* − *c*)
shows (*T* ^^ *n*) *0* = *c* + (*R* ^^ *n*) (− *c*)
proof (*induction n*)
case *0*
show *?case*
by *simp*
next
case (*Suc n*)
have (*T* ^^ *Suc n*) *0* = *T* ((*T* ^^ *n*) *0*)

by *simp*
also have ... = $T\ (c + (R\ ^\frown\!^\frown\ n)\ (-\ c))$
using *Suc.IH* **by** *simp*
also have ... = $c + R\ ((R\ ^\frown\!^\frown\ n)\ (-\ c))$
by (*simp add*: *T-def*)
also have ... = $c + (R\ ^\frown\!^\frown\ Suc\ n)\ (-\ c)$
by *simp*
finally show *?case* **.**
qed

lemma *sigma-replicate-b*:
$sigma\ (replicate\ n\ (False,\ B)) = (Rz\ ^\frown\!^\frown\ n)$
by (*induction n*) (*simp-all add*: *letter-to-rot-def*)

lemma *canceled-replicate-b*: $canceled\ (replicate\ n\ (False,\ B))$
unfolding *canceled-def*
proof
assume $Domainp\ cancels\text{-}to\text{-}1\ (replicate\ n\ (False,\ B))$
then obtain w **where** $cancels\text{-}to\text{-}1\ (replicate\ n\ (False,\ B))\ w$
by *auto*
then obtain i **where** i: $Suc\ i < length\ (replicate\ n\ (False,\ B))$
and $canceling\ (replicate\ n\ (False,\ B)\ !\ i)\ (replicate\ n\ (False,\ B)\ !\ Suc\ i)$
by (*auto simp*: *cancels-to-1-def cancels-to-1-at-def*)
hence $canceling\ (False,\ B)\ (False,\ B)$
by *simp*
thus *False*
by (*simp add*: *canceling-def*)
qed

lemma *replicate-b-in-F2*: $replicate\ n\ (False,\ B) \in carrier\ F2$
by (*intro canceled-in-F2 canceled-replicate-b*)

theorem *ball3-paradoxical-strong*:
$\exists\, P\ Q :: (real\hat{}3)\ set\ list.\ \exists\, gP\ gQ :: ((real\hat{}3) \Rightarrow (real\hat{}3))\ list.$
$length\ P = length\ gP \wedge length\ Q = length\ gQ \wedge$
$(\forall\, g \in set\ gP.\ isometry\ g) \wedge (\forall\, g \in set\ gQ.\ isometry\ g) \wedge$
$pairwise\text{-}disjoint\ (P\ @\ Q) \wedge$
$pairwise\text{-}disjoint\ (map2\ (\lambda g\ A.\ g\ `\ A)\ gP\ P) \wedge$
$pairwise\text{-}disjoint\ (map2\ (\lambda g\ A.\ g\ `\ A)\ gQ\ Q) \wedge$
$(\forall\, i < length\ P.\ P\ !\ i \subseteq ball3) \wedge$
$(\forall\, i < length\ Q.\ Q\ !\ i \subseteq ball3) \wedge$
$ball3 = (\bigcup i{<}length\ P.\ P\ !\ i) \cup (\bigcup i{<}length\ Q.\ Q\ !\ i) \wedge$
$ball3 = (\bigcup i{<}length\ P.\ (gP\ !\ i)\ `\ (P\ !\ i)) \wedge$
$ball3 = (\bigcup i{<}length\ Q.\ (gQ\ !\ i)\ `\ (Q\ !\ i))$
proof −
obtain a **where** $a\text{-}sphere$: $a \in sphere2$ **and** $a\text{-}not\text{-}D$: $a \notin bad\text{-}set\text{-}D$
using *exists-antipodal-axis-avoiding-countable*[*OF bad-set-D-countable*] **by** *blast*
let $?c = -\ (1\ /\ 2) *_R a$
define T **where** $T = (\lambda x.\ ?c + Rz\ (x - ?c))$

define U **where** $U = (\lambda x.\ ?c + Rz\text{-}inv\ (x - ?c))$
have *T-iso*: *isometry T*
unfolding *T-def* **by** (*rule offcenter-rotation-isometry*[*OF Rz-in-SO3*])
have *U-iso*: *isometry U*
unfolding *U-def* **by** (*rule offcenter-rotation-isometry*[*OF Rz-inv-in-SO3*])
have *U-T*: $\bigwedge x.\ U\ (T\ x) = x$
unfolding *T-def U-def*
by (*rule offcenter-rotation-inverse-left*[*OF Rz-inv-left*])
have *T-U*: $\bigwedge x.\ T\ (U\ x) = x$
unfolding *T-def U-def*
by (*rule offcenter-rotation-inverse-right*[*OF Rz-inv-right*])

have *no-Rz-power-fix*: $\bigwedge n.\ n > 0 \Longrightarrow (Rz \frown\frown n)\ a \neq a$
proof
fix $n :: nat$
assume *n-pos*: $n > 0$ **and** *fixed*: $(Rz \frown\frown n)\ a = a$
let $?w = replicate\ n\ (False, B)$
have *w-carrier*: $?w \in carrier\ F2$
by (*rule replicate-b-in-F2*)
have *w-ne*: $?w \neq []$
using *n-pos* **by** *auto*
have *sigma ?w a* $= a$
using *fixed* **by** (*simp add*: *sigma-replicate-b*)
hence $a \in fixed\text{-}point\text{-}set\ (sigma\ ?w)$
using *a-sphere* **by** (*simp add*: *fixed-point-set-def*)
hence $a \in bad\text{-}set\text{-}D$
unfolding *bad-set-D-def* **using** *w-carrier w-ne* **by** *blast*
with *a-not-D* **show** *False*
by *contradiction*
qed

have *Rz-power-a-inj*: $\bigwedge n\ m.\ (Rz \frown\frown n)\ a = (Rz \frown\frown m)\ a \Longrightarrow n = m$
proof −
fix $n\ m :: nat$
assume *eq*: $(Rz \frown\frown n)\ a = (Rz \frown\frown m)\ a$
show $n = m$
proof (*cases n m rule*: *linorder-cases*)
case *less*
let $?d = m - n$
have *d-pos*: $?d > 0$
using *less* **by** *simp*
have *m-eq*: $m = n + ?d$
using *less* **by** *simp*
have $(Rz \frown\frown n)\ a = (Rz \frown\frown n)\ ((Rz \frown\frown ?d)\ a)$
using *eq m-eq* **by** (*metis comp-apply funpow-add*)
hence $a = (Rz \frown\frown ?d)\ a$
using *SO3-inj*[*OF SO3-funpow*[*OF Rz-in-SO3*, *of n*]] **by** (*auto simp*: *inj-def*)
with *no-Rz-power-fix*[*OF d-pos*] **show** *?thesis*
by *simp*

```
  next
    case equal
    then show ?thesis .
  next
    case greater
    let ?d = n − m
    have d-pos: ?d > 0
      using greater by simp
    have n-eq: n = m + ?d
      using greater by simp
    have (Rz ^^ m) a = (Rz ^^ m) ((Rz ^^ ?d) a)
      using eq n-eq by (metis comp-apply funpow-add)
    hence a = (Rz ^^ ?d) a
    using SO3-inj[OF SO3-funpow[OF Rz-in-SO3, of m]] by (auto simp: inj-def)
    with no-Rz-power-fix[OF d-pos] show ?thesis
      by simp
  qed
qed

have T-pow-zero: ⋀n. (T ^^ n) 0 = ?c + (Rz ^^ n) (− ?c)
  unfolding T-def by (rule offcenter-rotation-funpow-zero)
have T-pow-zero-diff: ⋀n. (T ^^ n) 0 = (1 / 2) *R ((Rz ^^ n) a − a)
proof −
  fix n
  have lin: linear (Rz ^^ n)
    by (rule SO3-linear[OF SO3-funpow[OF Rz-in-SO3]])
  have (T ^^ n) 0 = ?c + (Rz ^^ n) (− ?c)
    by (rule T-pow-zero)
  also have ... = (− (1 / 2) *R a) + (1 / 2) *R ((Rz ^^ n) a)
    using linear-scale[OF lin, of 1 / 2 a] by simp
  also have ... = (1 / 2) *R ((Rz ^^ n) a − a)
    by (simp add: scaleR-right-diff-distrib)
  finally show (T ^^ n) 0 = (1 / 2) *R ((Rz ^^ n) a − a) .
qed

have T-orbit-inj: ⋀n m. (T ^^ n) 0 = (T ^^ m) 0 ⟹ n = m
proof −
  fix n m :: nat
  assume eq: (T ^^ n) 0 = (T ^^ m) 0
  have (Rz ^^ n) a = (Rz ^^ m) a
    using eq T-pow-zero-diff by simp
  thus n = m
    by (rule Rz-power-a-inj)
qed
have orbit-disj:
  ∀n m. n ≠ m ⟶ ((T^^n) ‘ {0}) ∩ ((T^^m) ‘ {0}) = {}
  using T-orbit-inj by auto

define E where E = (⋃n. (T^^n) ‘ {0::real^3})
```

have *T-E*: T ‘ $E = E - \{0\}$
unfolding *E-def* **by** (*rule absorbing-rotation-shift*[*OF orbit-disj*])
have *zero-subset-E*: $\{0::real\hat{}3\} \subseteq E$
proof
fix $x :: real\hat{}3$
assume $x \in \{0\}$
hence $x \in (T \frown\frown 0)$ ‘ $\{0\}$
by *simp*
thus $x \in E$
unfolding *E-def* **by** *blast*
qed
have *E-subset-ball3*: $E \subseteq ball3$
proof
fix x
assume $x \in E$
then obtain n **where** *x-eq*: $x = (T \frown\frown n)\ 0$
unfolding *E-def* **by** *blast*
have *Rza-sphere*: $(Rz \frown\frown n)\ a \in sphere2$
using *SO3-funpow-preserves-sphere2*[*OF Rz-in-SO3 a-sphere*] **.**
have $norm\ x = norm\ ((1\ /\ 2) *_R ((Rz \frown\frown n)\ a - a))$
using *x-eq T-pow-zero-diff* **by** *simp*
also have $\ldots = (1\ /\ 2) * norm\ ((Rz \frown\frown n)\ a - a)$
by *simp*
also have $\ldots \leq (1\ /\ 2) * (norm\ ((Rz \frown\frown n)\ a) + norm\ a)$
by (*intro mult-left-mono norm-triangle-ineq4*) *simp*
also have $\ldots = 1$
using *Rza-sphere a-sphere* **by** (*simp add*: *sphere2-def*)
finally show $x \in ball3$
by (*simp add*: *ball3-def*)
qed

let *?X* = $ball3 - \{0::real\hat{}3\}$
let *?Y* = *ball3*
let *?A* = $[E,\ ball3 - E]$
let *?B* = $[T$ ‘ $E,\ ball3 - E]$
let *?e* = $[T,\ id]$
let *?t* = $[U,\ id]$
have *A-cover*: $?Y = (\bigcup k{<}length\ ?A.\ ?A\ !\ k)$
using *E-subset-ball3* **by** (*auto simp*: *lessThan-Suc*)
have *A-disj*: *pairwise-disjoint ?A*
unfolding *pairwise-disjoint-def*
proof (*intro allI impI*)
fix *i j*
assume *i*: $i < length\ ?A$ **and** *j*: $j < length\ ?A$ **and** *ij*: $i \neq j$
have $(i = 0 \wedge j = 1) \vee (i = 1 \wedge j = 0)$
using *i j ij* **by** *auto*
thus $?A\ !\ i \cap ?A\ !\ j = \{\}$
by *auto*
qed

have *B-cover*: *?X* = (⋃ *k*<*length ?B*. *?B* ! *k*)
proof
  show *?X* ⊆ (⋃ *k*<*length ?B*. *?B* ! *k*)
  proof
    fix *x*
    assume *x*: *x* ∈ *?X*
    show *x* ∈ (⋃ *k*<*length ?B*. *?B* ! *k*)
    proof (*cases x* ∈ *E*)
      case *True*
      with *x T-E* **show** *?thesis*
        by *force*
    next
      case *False*
      with *x* **show** *?thesis*
        by *force*
    qed
  qed
  show (⋃ *k*<*length ?B*. *?B* ! *k*) ⊆ *?X*
  proof
    fix *x*
    assume *x*: *x* ∈ (⋃ *k*<*length ?B*. *?B* ! *k*)
    then obtain *k* **where** *k*: *k* < *length ?B* **and** *xk*: *x* ∈ *?B* ! *k*
      by *blast*
    have *k* = *0* ∨ *k* = *1*
      using *k* **by** *auto*
    thus *x* ∈ *?X*
    proof
      assume *k* = *0*
      with *xk T-E E-subset-ball3* **show** *?thesis*
        by *auto*
    next
      assume *k* = *1*
      with *xk zero-subset-E* **show** *?thesis*
        by *auto*
    qed
  qed
qed
have *B-disj*: *pairwise-disjoint ?B*
  unfolding *pairwise-disjoint-def*
proof (*intro allI impI*)
  fix *i j*
  assume *i*: *i* < *length ?B* **and** *j*: *j* < *length ?B* **and** *ij*: *i* ≠ *j*
  have (*i* = *0* ∧ *j* = *1*) ∨ (*i* = *1* ∧ *j* = *0*)
    using *i j ij* **by** *auto*
  thus *?B* ! *i* ∩ *?B* ! *j* = {}
    using *T-E* **by** *auto*
qed
have *e-left*: ⋀*k x*. ⟦*k* < *length ?A*; *x* ∈ *?A* ! *k*⟧
  ⟹ (*?e* ! *k*) *x* ∈ *?B* ! *k* ∧ (*?t* ! *k*) ((*?e* ! *k*) *x*) = *x*

proof −
fix k x
assume k: $k < \mathit{length}\ \mathit{?A}$ **and** x: $x \in \mathit{?A}\ !\ k$
have $k = 0 \vee k = 1$
using k **by** *auto*
thus $(\mathit{?e}\ !\ k)\ x \in \mathit{?B}\ !\ k \wedge (\mathit{?t}\ !\ k)\ ((\mathit{?e}\ !\ k)\ x) = x$
proof
assume $k = 0$
with x *U-T* **show** *?thesis*
by *auto*
next
assume $k = 1$
with x **show** *?thesis*
by *auto*
qed
qed
have *t-left*: $\bigwedge k\ y.\ [\![k < \mathit{length}\ \mathit{?B};\ y \in \mathit{?B}\ !\ k]\!]$
$\Longrightarrow (\mathit{?t}\ !\ k)\ y \in \mathit{?A}\ !\ k \wedge (\mathit{?e}\ !\ k)\ ((\mathit{?t}\ !\ k)\ y) = y$
proof −
fix k y
assume k: $k < \mathit{length}\ \mathit{?B}$ **and** y: $y \in \mathit{?B}\ !\ k$
have $k = 0 \vee k = 1$
using k **by** *auto*
thus $(\mathit{?t}\ !\ k)\ y \in \mathit{?A}\ !\ k \wedge (\mathit{?e}\ !\ k)\ ((\mathit{?t}\ !\ k)\ y) = y$
proof
assume $k = 0$
then obtain x **where** x: $x \in E$ **and** *y-eq*: $y = T\ x$
using y **by** *auto*
hence $U\ y = x$
using *U-T* **by** *simp*
with x *y-eq* *T-U* ‹$k = 0$› **show** *?thesis*
by *simp*
next
assume $k = 1$
with y **show** *?thesis*
by *auto*
qed
qed

from *ball3-minus-origin-paradoxical-strong* **obtain** $P\ Q :: (\mathit{real}\hat{}3)\ \mathit{set}\ \mathit{list}$
and $gP\ gQ :: ((\mathit{real}\hat{}3) \Rightarrow (\mathit{real}\hat{}3))\ \mathit{list}$
where *lenP*: $\mathit{length}\ P = \mathit{length}\ gP$
and *lenQ*: $\mathit{length}\ Q = \mathit{length}\ gQ$
and *gP-SO3*: $\mathit{set}\ gP \subseteq \mathit{SO3}$
and *gQ-SO3*: $\mathit{set}\ gQ \subseteq \mathit{SO3}$
and *source-disj*: $\mathit{pairwise\text{-}disjoint}\ (P\ @\ Q)$
and *imageP-disj*: $\mathit{pairwise\text{-}disjoint}\ (\mathit{map2}\ (\lambda g\ A.\ g\ `\ A)\ gP\ P)$
and *imageQ-disj*: $\mathit{pairwise\text{-}disjoint}\ (\mathit{map2}\ (\lambda g\ A.\ g\ `\ A)\ gQ\ Q)$
and *P-sub*: $\forall\, i < \mathit{length}\ P.\ P\ !\ i \subseteq \mathit{?X}$

and *Q-sub*: $\forall\, i < \mathit{length}\ Q.\ Q\ !\ i \subseteq ?X$
and *source-cover*: $?X = (\bigcup i{<}\mathit{length}\ P.\ P\ !\ i) \cup (\bigcup i{<}\mathit{length}\ Q.\ Q\ !\ i)$
and *imageP-cover*: $?X = (\bigcup i{<}\mathit{length}\ P.\ (gP\ !\ i)\ `\ (P\ !\ i))$
and *imageQ-cover*: $?X = (\bigcup i{<}\mathit{length}\ Q.\ (gQ\ !\ i)\ `\ (Q\ !\ i))$
by *auto*
have *gP-inj*: $\bigwedge i.\ i < \mathit{length}\ P \Longrightarrow \mathit{inj}\ (gP\ !\ i)$
using *SO3-inj gP-SO3 lenP nth-mem* **by** *auto*
have *gQ-inj*: $\bigwedge i.\ i < \mathit{length}\ Q \Longrightarrow \mathit{inj}\ (gQ\ !\ i)$
using *SO3-inj gQ-SO3 lenQ nth-mem* **by** *auto*
have *mapsP*: $\bigwedge k\ i\ l.\ [\![k < \mathit{length}\ ?A;\ i < \mathit{length}\ P;\ l < \mathit{length}\ ?B]\!]$
$\Longrightarrow \mathit{transfer\text{-}map}\ ?t\ gP\ ?e\ k\ i\ l \in \{f.\ \mathit{isometry}\ f\}$
proof −
fix *k i l*
assume *k*: $k < \mathit{length}\ ?A$ **and** *i*: $i < \mathit{length}\ P$ **and** *l*: $l < \mathit{length}\ ?B$
have *gi-SO3*: $gP\ !\ i \in \mathit{SO3}$
using *gP-SO3 i lenP* **by** *auto*
have *gi-iso*: $\mathit{isometry}\ (gP\ !\ i)$
by (*rule SO3-isometry*[*OF gi-SO3*])
have *ek-iso*: $\mathit{isometry}\ (?e\ !\ k)$
using *T-iso k less-Suc-eq* **by** *fastforce*
have *tl-iso*: $\mathit{isometry}\ (?t\ !\ l)$
using *U-iso l less-Suc-eq* **by** *fastforce*
show $\mathit{transfer\text{-}map}\ ?t\ gP\ ?e\ k\ i\ l \in \{f.\ \mathit{isometry}\ f\}$
unfolding *transfer-map-def*
by (*simp add*: *isometry-compose tl-iso gi-iso ek-iso*)
qed
have *mapsQ*: $\bigwedge k\ i\ l.\ [\![k < \mathit{length}\ ?A;\ i < \mathit{length}\ Q;\ l < \mathit{length}\ ?B]\!]$
$\Longrightarrow \mathit{transfer\text{-}map}\ ?t\ gQ\ ?e\ k\ i\ l \in \{f.\ \mathit{isometry}\ f\}$
proof −
fix *k i l*
assume *k*: $k < \mathit{length}\ ?A$ **and** *i*: $i < \mathit{length}\ Q$ **and** *l*: $l < \mathit{length}\ ?B$
have *gi-SO3*: $gQ\ !\ i \in \mathit{SO3}$
using *gQ-SO3 i lenQ* **by** *auto*
have *gi-iso*: $\mathit{isometry}\ (gQ\ !\ i)$
by (*rule SO3-isometry*[*OF gi-SO3*])
have *ek-iso*: $\mathit{isometry}\ (?e\ !\ k)$
using *T-iso k less-Suc-eq* **by** *fastforce*
have *tl-iso*: $\mathit{isometry}\ (?t\ !\ l)$
using *U-iso l less-Suc-eq* **by** *fastforce*
show $\mathit{transfer\text{-}map}\ ?t\ gQ\ ?e\ k\ i\ l \in \{f.\ \mathit{isometry}\ f\}$
unfolding *transfer-map-def*
by (*simp add*: *isometry-compose tl-iso gi-iso ek-iso*)
qed
have *lenAB*: $\mathit{length}\ ?A = \mathit{length}\ ?B$
by *simp*
from *transfer-partitioned-paradox*[
OF lenAB A-cover A-disj B-cover B-disj e-left t-left lenP lenQ source-disj source-cover imageP-disj imageQ-disj imageP-cover imageQ-cover gP-inj gQ-inj

```
    mapsP mapsQ]
  obtain P' Q' :: (real^3) set list and hP hQ :: ((real^3) ⇒ (real^3)) list
    where lenP': length P' = length hP
      and lenQ': length Q' = length hQ
      and hP-iso: set hP ⊆ {f. isometry f}
      and hQ-iso: set hQ ⊆ {f. isometry f}
      and source-disj': pairwise-disjoint (P' @ Q')
      and imageP-disj': pairwise-disjoint (map2 (λg A. g ‘ A) hP P')
      and imageQ-disj': pairwise-disjoint (map2 (λg A. g ‘ A) hQ Q')
      and source-cover': ?Y = (⋃i<length P'. P' ! i) ∪ (⋃i<length Q'. Q' ! i)
      and imageP-cover': ?Y = (⋃i<length P'. (hP ! i) ‘ (P' ! i))
      and imageQ-cover': ?Y = (⋃i<length Q'. (hQ ! i) ‘ (Q' ! i))
    by auto
  have P'-sub: ∀ i < length P'. P' ! i ⊆ ball3
    using source-cover' by blast
  have Q'-sub: ∀ i < length Q'. Q' ! i ⊆ ball3
    using source-cover' by blast
  show ?thesis
  proof (intro exI conjI)
    show length P' = length hP
      by (rule lenP')
    show length Q' = length hQ
      by (rule lenQ')
    show ∀ g∈set hP. isometry g
      using hP-iso by blast
    show ∀ g∈set hQ. isometry g
      using hQ-iso by blast
    show pairwise-disjoint (P' @ Q')
      by (rule source-disj')
    show pairwise-disjoint (map2 (λg A. g ‘ A) hP P')
      by (rule imageP-disj')
    show pairwise-disjoint (map2 (λg A. g ‘ A) hQ Q')
      by (rule imageQ-disj')
    show ∀ i<length P'. P' ! i ⊆ ball3
      by (rule P'-sub)
    show ∀ i<length Q'. Q' ! i ⊆ ball3
      by (rule Q'-sub)
    show ball3 = (⋃i<length P'. P' ! i) ∪ (⋃i<length Q'. Q' ! i)
      by (rule source-cover')
    show ball3 = (⋃i<length P'. (hP ! i) ‘ (P' ! i))
      by (rule imageP-cover')
    show ball3 = (⋃i<length Q'. (hQ ! i) ‘ (Q' ! i))
      by (rule imageQ-cover')
  qed
qed
```

An auxiliary finite-cover theorem for the closed unit ball.

```
theorem banach-tarski-finite-cover:
  ∃ P :: (real^3) set list. ∃ gs :: ((real^3) ⇒ (real^3)) list.
```

$\mathit{length}\ P = \mathit{length}\ gs \wedge$
$(\forall\, g \in \mathit{set}\ gs.\ \mathit{isometry}\ g) \wedge$
$(\forall\, i < \mathit{length}\ P.\ P\ !\ i \subseteq \mathit{ball3}) \wedge$
$\mathit{ball3} = (\bigcup i{<}\mathit{length}\ P.\ P\ !\ i) \wedge$
$(\mathit{ball3\text{-}copy\text{-}left} \cup \mathit{ball3\text{-}copy\text{-}right}) =$
$(\bigcup i{<}\mathit{length}\ P.\ (gs\ !\ i)\ `\ (P\ !\ i)) \wedge$
$\mathit{ball3\text{-}copy\text{-}left} \cap \mathit{ball3\text{-}copy\text{-}right} = \{\}$

proof −
from *ball3-paradoxical* **obtain** $P\ Q :: (\mathit{real\hat{}3})\ \mathit{set}\ \mathit{list}$
and $gP\ gQ :: ((\mathit{real\hat{}3} \Rightarrow \mathit{real\hat{}3}))\ \mathit{list}$
where *decomp*:
$\mathit{length}\ P = \mathit{length}\ gP \wedge \mathit{length}\ Q = \mathit{length}\ gQ \wedge$
$\mathit{set}\ gP \subseteq \mathit{SO3} \wedge \mathit{set}\ gQ \subseteq \mathit{SO3} \wedge$
$(\forall\, i < \mathit{length}\ P.\ P\ !\ i \subseteq \mathit{ball3}) \wedge$
$(\forall\, i < \mathit{length}\ Q.\ Q\ !\ i \subseteq \mathit{ball3}) \wedge$
$\mathit{ball3} = (\bigcup i{<}\mathit{length}\ P.\ P\ !\ i) \wedge$
$\mathit{ball3} = (\bigcup i{<}\mathit{length}\ Q.\ Q\ !\ i) \wedge$
$\mathit{ball3} = (\bigcup i{<}\mathit{length}\ P.\ (gP\ !\ i)\ `\ (P\ !\ i)) \wedge$
$\mathit{ball3} = (\bigcup i{<}\mathit{length}\ Q.\ (gQ\ !\ i)\ `\ (Q\ !\ i))$
by (*elim exE*)
from *decomp* **have** *lenP*: $\mathit{length}\ P = \mathit{length}\ gP$
by (*elim conjE*)
from *decomp* **have** *lenQ*: $\mathit{length}\ Q = \mathit{length}\ gQ$
by (*elim conjE*)
from *decomp* **have** *gP-SO3*: $\mathit{set}\ gP \subseteq \mathit{SO3}$
by (*elim conjE*)
from *decomp* **have** *gQ-SO3*: $\mathit{set}\ gQ \subseteq \mathit{SO3}$
by (*elim conjE*)
from *decomp* **have** *P-sub*: $\forall\, i < \mathit{length}\ P.\ P\ !\ i \subseteq \mathit{ball3}$
by (*elim conjE*)
from *decomp* **have** *Q-sub*: $\forall\, i < \mathit{length}\ Q.\ Q\ !\ i \subseteq \mathit{ball3}$
by (*elim conjE*)
from *decomp* **have** *P-src*: $\mathit{ball3} = (\bigcup i{<}\mathit{length}\ P.\ P\ !\ i)$
by (*elim conjE*)
from *decomp* **have** *Q-src*: $\mathit{ball3} = (\bigcup i{<}\mathit{length}\ Q.\ Q\ !\ i)$
by (*elim conjE*)
from *decomp* **have** *P-cov*: $\mathit{ball3} = (\bigcup i{<}\mathit{length}\ P.\ (gP\ !\ i)\ `\ (P\ !\ i))$
by (*elim conjE*)
from *decomp* **have** *Q-cov*: $\mathit{ball3} = (\bigcup i{<}\mathit{length}\ Q.\ (gQ\ !\ i)\ `\ (Q\ !\ i))$
by (*elim conjE*)

define *pieces* **where** $\mathit{pieces} = P\ @\ Q$
define *hQ* **where** $hQ = \mathit{map}\ (\lambda g.\ (+)\ \mathit{shift3} \circ g)\ gQ$
define *gs* **where** $gs = gP\ @\ hQ$
have *len-hQ*: $\mathit{length}\ Q = \mathit{length}\ hQ$
by (*simp add*: *hQ-def lenQ*)
have *len*: $\mathit{length}\ \mathit{pieces} = \mathit{length}\ gs$
by (*simp add*: *pieces-def gs-def lenP len-hQ*)
have *iso*: $\forall\, g \in \mathit{set}\ gs.\ \mathit{isometry}\ g$

```
  using gP-SO3 gQ-SO3
  by (auto simp add: gs-def hQ-def SO3-isometry)
have sub: ∀ i < length pieces. pieces ! i ⊆ ball3
  using P-sub Q-sub by (auto simp add: pieces-def nth-append)
have src: ball3 = (⋃ i<length pieces. pieces ! i)
proof −
  have append-union:
    (⋃ i<length (P @ Q). (P @ Q) ! i) =
      (⋃ i<length P. P ! i) ∪ (⋃ i<length Q. Q ! i)
    using indexed-union-append[of P Q] .
  have (⋃ i<length pieces. pieces ! i) =
      (⋃ i<length P. P ! i) ∪ (⋃ i<length Q. Q ! i)
    unfolding pieces-def
    using append-union .
  also have ... = ball3
    using P-src Q-src by simp
  finally show ?thesis by simp
qed
have right-cov:
  ball3-copy-right = (⋃ i<length Q. (hQ ! i) ‘ (Q ! i))
proof −
  have right-union:
    (⋃ i<length Q. (hQ ! i) ‘ (Q ! i)) =
      (+) shift3 ‘ (⋃ i<length Q. (gQ ! i) ‘ (Q ! i))
  proof (rule equalityI)
    show (⋃ i<length Q. (hQ ! i) ‘ (Q ! i)) ⊆
        (+) shift3 ‘ (⋃ i<length Q. (gQ ! i) ‘ (Q ! i))
    proof
      fix x assume x ∈ (⋃ i<length Q. (hQ ! i) ‘ (Q ! i))
      then obtain i y where i-lt: i < length Q and y-in: y ∈ Q ! i
        and x-eq: x = (hQ ! i) y
        by blast
      have h-eq: hQ ! i = (+) shift3 ∘ (gQ ! i)
        using i-lt lenQ by (simp add: hQ-def)
      have (gQ ! i) y ∈ (⋃ i<length Q. (gQ ! i) ‘ (Q ! i))
        using i-lt y-in by blast
      moreover have x = shift3 + (gQ ! i) y
        using x-eq h-eq by simp
      ultimately show x ∈ (+) shift3 ‘ (⋃ i<length Q. (gQ ! i) ‘ (Q ! i))
        by blast
    qed
    show (+) shift3 ‘ (⋃ i<length Q. (gQ ! i) ‘ (Q ! i)) ⊆
        (⋃ i<length Q. (hQ ! i) ‘ (Q ! i))
    proof
      fix x assume x ∈ (+) shift3 ‘ (⋃ i<length Q. (gQ ! i) ‘ (Q ! i))
      then obtain z i y where x-eq: x = shift3 + z
        and i-lt: i < length Q and y-in: y ∈ Q ! i and z-eq: z = (gQ ! i) y
        by blast
      have h-eq: hQ ! i = (+) shift3 ∘ (gQ ! i)
```

```
        using i-lt lenQ by (simp add: hQ-def)
      have x = (hQ ! i) y
        using x-eq z-eq h-eq by simp
      thus x ∈ (⋃i<length Q. (hQ ! i) ‘ (Q ! i))
        using i-lt y-in by blast
    qed
  qed
  show ?thesis
    unfolding ball3-copy-right-def
    using right-union Q-cov by simp
qed
have images:
  ball3-copy-left ∪ ball3-copy-right =
    (⋃i<length pieces. (gs ! i) ‘ (pieces ! i))
proof −
  have append-images:
    (⋃i<length (P @ Q). (gP @ hQ) ! i ‘ ((P @ Q) ! i)) =
      (⋃i<length P. (gP ! i) ‘ (P ! i)) ∪
      (⋃i<length Q. (hQ ! i) ‘ (Q ! i))
    using indexed-image-union-append[OF lenP len-hQ] .
  have (⋃i<length pieces. (gs ! i) ‘ (pieces ! i)) =
      (⋃i<length P. (gP ! i) ‘ (P ! i)) ∪
      (⋃i<length Q. (hQ ! i) ‘ (Q ! i))
    unfolding pieces-def gs-def
    using append-images .
  also have ... = ball3-copy-left ∪ ball3-copy-right
    using P-cov right-cov by (simp add: ball3-copy-left-def)
  finally show ?thesis by simp
qed
show ?thesis
  apply (rule exI[of - pieces])
  apply (rule exI[of - gs])
  using len iso sub src images ball3-copies-disjoint
  by simp
qed
```

The Banach–Tarski theorem for the closed unit ball, as a partition theorem.

```
theorem banach-tarski:
  ∃ P :: (real^3) set list. ∃ gs :: ((real^3) ⇒ (real^3)) list.
      length P = length gs ∧
      (∀ g ∈ set gs. isometry g) ∧
      pairwise-disjoint P ∧
      pairwise-disjoint (map2 (λg A. g ‘ A) gs P) ∧
      (∀ i < length P. P ! i ⊆ ball3) ∧
      ball3 = (⋃i<length P. P ! i) ∧
      (ball3-copy-left ∪ ball3-copy-right) =
        (⋃i<length P. (gs ! i) ‘ (P ! i)) ∧
      ball3-copy-left ∩ ball3-copy-right = {}
proof −
```

from *ball3-paradoxical-strong* **obtain** $P\ Q :: (real\hat{}3)\ set\ list$
  and $gP\ gQ :: ((real\hat{}3 \Rightarrow real\hat{}3))\ list$
  where *lenP*: $length\ P = length\ gP$
    and *lenQ*: $length\ Q = length\ gQ$
    and *gP-iso*: $\forall g \in set\ gP.\ isometry\ g$
    and *gQ-iso*: $\forall g \in set\ gQ.\ isometry\ g$
    and *source-disj*: $pairwise\text{-}disjoint\ (P\ @\ Q)$
    and *imageP-disj*: $pairwise\text{-}disjoint\ (map2\ (\lambda g\ A.\ g\ `\ A)\ gP\ P)$
    and *imageQ-disj*: $pairwise\text{-}disjoint\ (map2\ (\lambda g\ A.\ g\ `\ A)\ gQ\ Q)$
    and *P-sub*: $\forall i < length\ P.\ P\ !\ i \subseteq ball3$
    and *Q-sub*: $\forall i < length\ Q.\ Q\ !\ i \subseteq ball3$
    and *source-cover*: $ball3 = (\bigcup i{<}length\ P.\ P\ !\ i) \cup (\bigcup i{<}length\ Q.\ Q\ !\ i)$
    and *P-cov*: $ball3 = (\bigcup i{<}length\ P.\ (gP\ !\ i)\ `\ (P\ !\ i))$
    and *Q-cov*: $ball3 = (\bigcup i{<}length\ Q.\ (gQ\ !\ i)\ `\ (Q\ !\ i))$
  by *auto*
define *pieces* **where** $pieces = P\ @\ Q$
define *hQ* **where** $hQ = map\ (\lambda g.\ (+)\ shift3 \circ g)\ gQ$
define *gs* **where** $gs = gP\ @\ hQ$
have *len-hQ*: $length\ Q = length\ hQ$
  by (*simp add*: *hQ-def lenQ*)
have *len*: $length\ pieces = length\ gs$
  by (*simp add*: *pieces-def gs-def lenP len-hQ*)
have *iso*: $\forall g \in set\ gs.\ isometry\ g$
  using *gP-iso gQ-iso*
  by (*auto simp add*: *gs-def hQ-def isometry-compose*)
have *sub*: $\forall i < length\ pieces.\ pieces\ !\ i \subseteq ball3$
  using *P-sub Q-sub* **by** (*auto simp add*: *pieces-def nth-append*)
have *src*: $ball3 = (\bigcup i{<}length\ pieces.\ pieces\ !\ i)$
  by (*metis indexed-union-append pieces-def source-cover*)
have *source-disj-pieces*: $pairwise\text{-}disjoint\ pieces$
  using *source-disj* **by** (*simp add*: *pieces-def*)

have *right-cov*:
  $ball3\text{-}copy\text{-}right = (\bigcup i{<}length\ Q.\ (hQ\ !\ i)\ `\ (Q\ !\ i))$
proof −
  have *right-union*:
    $(\bigcup i{<}length\ Q.\ (hQ\ !\ i)\ `\ (Q\ !\ i)) =$
      $(+)\ shift3\ `\ (\bigcup i{<}length\ Q.\ (gQ\ !\ i)\ `\ (Q\ !\ i))$
  proof (*rule equalityI*)
    show $(\bigcup i{<}length\ Q.\ (hQ\ !\ i)\ `\ (Q\ !\ i)) \subseteq$
        $(+)\ shift3\ `\ (\bigcup i{<}length\ Q.\ (gQ\ !\ i)\ `\ (Q\ !\ i))$
    proof
      fix *x* **assume** $x \in (\bigcup i{<}length\ Q.\ (hQ\ !\ i)\ `\ (Q\ !\ i))$
      then obtain *i y* **where** *i-lt*: $i < length\ Q$ **and** *y-in*: $y \in Q\ !\ i$
        and *x-eq*: $x = (hQ\ !\ i)\ y$
        by *blast*
      have *h-eq*: $hQ\ !\ i = (+)\ shift3 \circ (gQ\ !\ i)$
        using *i-lt lenQ* **by** (*simp add*: *hQ-def*)
      have $(gQ\ !\ i)\ y \in (\bigcup i{<}length\ Q.\ (gQ\ !\ i)\ `\ (Q\ !\ i))$

```
      using i-lt y-in by blast
    moreover have x = shift3 + (gQ ! i) y
      using x-eq h-eq by simp
    ultimately show x ∈ (+) shift3 ‘ (⋃i<length Q. (gQ ! i) ‘ (Q ! i))
      by blast
  qed
  show (+) shift3 ‘ (⋃i<length Q. (gQ ! i) ‘ (Q ! i)) ⊆
      (⋃i<length Q. (hQ ! i) ‘ (Q ! i))
  proof
    fix x assume x ∈ (+) shift3 ‘ (⋃i<length Q. (gQ ! i) ‘ (Q ! i))
    then obtain z i y where x-eq: x = shift3 + z
      and i-lt: i < length Q and y-in: y ∈ Q ! i and z-eq: z = (gQ ! i) y
      by blast
    have h-eq: hQ ! i = (+) shift3 ∘ (gQ ! i)
      using i-lt lenQ by (simp add: hQ-def)
    have x = (hQ ! i) y
      using x-eq z-eq h-eq by simp
    thus x ∈ (⋃i<length Q. (hQ ! i) ‘ (Q ! i))
      using i-lt y-in by blast
  qed
 qed
 show ?thesis
  unfolding ball3-copy-right-def
  using right-union Q-cov by simp
qed
have images:
 ball3-copy-left ∪ ball3-copy-right =
  (⋃i<length pieces. (gs ! i) ‘ (pieces ! i))
proof −
 have (⋃i<length pieces. (gs ! i) ‘ (pieces ! i)) =
    (⋃i<length P. (gP ! i) ‘ (P ! i)) ∪
    (⋃i<length Q. (hQ ! i) ‘ (Q ! i))
  unfolding pieces-def gs-def
  by (rule indexed-image-union-append[OF lenP len-hQ])
 also have ... = ball3-copy-left ∪ ball3-copy-right
  using P-cov right-cov by (simp add: ball3-copy-left-def)
 finally show ?thesis by simp
qed

have shift-inj: inj ((+) shift3)
 by (simp add: inj-def)
have right-image-disj: pairwise-disjoint (map2 (λg A. g ‘ A) hQ Q)
 unfolding pairwise-disjoint-def
proof (intro allI impI)
 fix i j
 assume i: i < length (map2 (λg A. g ‘ A) hQ Q)
  and j: j < length (map2 (λg A. g ‘ A) hQ Q)
  and ij: i ≠ j
 have iQ: i < length Q and jQ: j < length Q
```

```
    using i j len-hQ by auto
  have base-disj: (gQ ! i ‘ (Q ! i)) ∩ (gQ ! j ‘ (Q ! j)) = {}
    using imageQ-disj lenQ iQ jQ ij by (simp add: pairwise-disjoint-def)
  have hi: hQ ! i = (+) shift3 ∘ (gQ ! i)
    using iQ lenQ by (simp add: hQ-def)
  have hj: hQ ! j = (+) shift3 ∘ (gQ ! j)
    using jQ lenQ by (simp add: hQ-def)
  show map2 (λg A. g ‘ A) hQ Q ! i ∩ map2 (λg A. g ‘ A) hQ Q ! j = {}
    using base-disj shift-inj iQ jQ len-hQ
    by (auto simp: hi hj inj-def)
qed
have left-sets-sub: ⋀A. A ∈ set (map2 (λg A. g ‘ A) gP P) ⟹ A ⊆ ball3-copy-left
proof −
  fix A
  assume A: A ∈ set (map2 (λg A. g ‘ A) gP P)
  then obtain i where i: i < length P and A-eq: A = (gP ! i) ‘ (P ! i)
    using lenP by (auto simp: in-set-conv-nth)
  have A ⊆ (⋃i<length P. (gP ! i) ‘ (P ! i))
    using i A-eq by blast
  thus A ⊆ ball3-copy-left
    using P-cov by (simp add: ball3-copy-left-def)
qed
have right-sets-sub: ⋀A. A ∈ set (map2 (λg A. g ‘ A) hQ Q) ⟹ A ⊆ ball3-copy-right
proof −
  fix A
  assume A: A ∈ set (map2 (λg A. g ‘ A) hQ Q)
  then obtain i where i: i < length Q and A-eq: A = (hQ ! i) ‘ (Q ! i)
    using len-hQ by (auto simp: in-set-conv-nth)
  have A ⊆ (⋃i<length Q. (hQ ! i) ‘ (Q ! i))
    using i A-eq by blast
  thus A ⊆ ball3-copy-right
    using right-cov by simp
qed
have image-cross-disj:
  ⋀A B. ⟦A ∈ set (map2 (λg A. g ‘ A) gP P);
    B ∈ set (map2 (λg A. g ‘ A) hQ Q)⟧ ⟹ A ∩ B = {}
proof −
  fix A B
  assume A: A ∈ set (map2 (λg A. g ‘ A) gP P)
    and B: B ∈ set (map2 (λg A. g ‘ A) hQ Q)
  have A ∩ B ⊆ ball3-copy-left ∩ ball3-copy-right
    using left-sets-sub[OF A] right-sets-sub[OF B] by blast
  thus A ∩ B = {}
    using ball3-copies-disjoint by blast
qed
have image-disj: pairwise-disjoint (map2 (λg A. g ‘ A) gs pieces)
  unfolding gs-def pieces-def
 using pairwise-disjoint-appendI[OF imageP-disj right-image-disj image-cross-disj]
  using lenP by fastforce
```

```
  show ?thesis
  proof (intro exI conjI)
    show length pieces = length gs
      by (rule len)
    show ∀ g∈set gs. isometry g
      by (rule iso)
    show pairwise-disjoint pieces
      by (rule source-disj-pieces)
    show pairwise-disjoint (map2 (λg A. g ‘ A) gs pieces)
      by (rule image-disj)
    show ∀ i<length pieces. pieces ! i ⊆ ball3
      by (rule sub)
    show ball3 = (⋃ i<length pieces. pieces ! i)
      by (rule src)
    show ball3-copy-left ∪ ball3-copy-right =
        (⋃ i<length pieces. (gs ! i) ‘ (pieces ! i))
      by (rule images)
    show ball3-copy-left ∩ ball3-copy-right = {}
      by (rule ball3-copies-disjoint)
  qed
qed

end
```

References

[1] S. Banach and A. Tarski. Sur la décomposition des ensembles de points en parties respectivement congruentes. *Fundamenta Mathematicae*, 6(1):244–277, 1924. DOI: https://doi.org/10.4064/fm-6-1-244-277. Also available at https://eudml.org/doc/214280.

[2] J. Breitner. Free groups. *Archive of Formal Proofs*, June 2010. https://isa-afp.org/entries/Free-Groups.html, Formal proof development.

[3] D. de Rauglaudre. Formal proof of Banach–Tarski paradox. *Journal of Formalized Reasoning*, 10(1):37–49, 2017. DOI: https://doi.org/10.6092/issn.1972-5787/6927.

[4] F. Hausdorff. Bemerkung über den Inhalt von Punktmengen. *Mathematische Annalen*, 75:428–433, 1914. DOI: https://doi.org/10.1007/BF01563735. Also available at https://eudml.org/doc/158671.

[5] T. Tao. *An Introduction to Measure Theory*, volume 126 of *Graduate Studies in Mathematics*. American Mathematical Society, 2011. DOI: https://doi.org/10.1090/gsm/126.

[6] G. Tomkowicz and S. Wagon. *The Banach–Tarski Paradox*, volume 163 of *Encyclopedia of Mathematics and its Applications*. Cambridge

University Press, 2nd edition, 2016. DOI: https://doi.org/10.1017/CBO9781107337145.

[7] S. Wagon. *The Banach–Tarski Paradox*, volume 24 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1993. First paperback edition. DOI: https://doi.org/10.1017/CBO9780511609596.