The Cardinality of the Continuum

Manuel Eberl

April 18, 2024

Abstract

This entry presents a short derivation of the cardinality of $\mathbb{R}$, namely that $|\mathbb{R}| = |2^{\mathbb{N}}| = 2^{\aleph_0}$. This is done by showing the injection $\mathbb{R} \to 2^{\mathbb{Q}},\ x \mapsto (-\infty, x) \cap \mathbb{Q}$ (i.e. Dedekind cuts) for one direction and the injection $2^{\mathbb{N}} \to \mathbb{Q},\ X \mapsto \sum_{n \in X} 3^{-n}$, i.e. ternary fractions, for the other direction.

Contents

1 Auxiliary material

```
theory Cardinality_Continuum_Library
  imports "HOL-Library.Equipollence" "HOL-Cardinals.Cardinals"
begin
```

1.1 Miscellaneous facts about cardinalities

```
lemma eqpoll_Pow [intro]:
  assumes "A ≈ B"
  shows   "Pow A ≈ Pow B"
proof -
  from assms obtain f where "bij_betw f A B"
    unfolding eqpoll_def by blast
  hence "bij_betw ((`) f) (Pow A) (Pow B)"
    by (rule bij_betw_Pow)
  thus ?thesis
    unfolding eqpoll_def by blast
qed

lemma lepoll_UNIV_nat_iff: "A ≲ (UNIV :: nat set) ⟷ countable A"
  unfolding countable_def lepoll_def by simp

lemma countable_eqpoll:
  assumes "countable A" "A ≈ B"
  shows   "countable B"
  using assms countable_iff_bij unfolding eqpoll_def by blast

lemma countable_eqpoll_cong: "A ≈ B ⟹ countable A ⟷ countable B"
  using countable_eqpoll[of A B] countable_eqpoll[of B A]
  by (auto simp: eqpoll_sym)

lemma eqpoll_UNIV_nat_iff: "A ≈ (UNIV :: nat set) ⟷ countable A ∧
infinite A"
proof
  assume *: "A ≈ (UNIV :: nat set)"
  show "countable A ∧ infinite A"
    using eqpoll_finite_iff[OF *] countable_eqpoll_cong[OF *] by simp
next
  assume *: "countable A ∧ infinite A"
  thus "A ≈ (UNIV :: nat set)"
    by (meson countableE_infinite eqpoll_def)
qed


lemma ordLeq_finite_infinite:
  "finite A ⟹ infinite B ⟹ (card_of A, card_of B) ∈ ordLeq"
  by (meson card_of_Well_order card_of_ordLeq_finite ordLeq_total)
```

```
lemma eqpoll_imp_card_of_ordIso: "A ≈ B ⟹ |A| =o |B|"
  by (simp add: eqpoll_iff_card_of_ordIso)

lemma card_of_Func: "|Func A B| =o |B| ^c |A|"
  by (simp add: cexp_def)

lemma card_of_leq_natLeq_iff_countable:
  "|X| ≤o natLeq ⟷ countable X"
proof -
  have "countable X  ⟷ |X| ≤o |UNIV :: nat set|"
    unfolding countable_def by (meson card_of_ordLeq top_greatest)
  with card_of_nat show ?thesis
    using ordIso_symmetric ordLeq_ordIso_trans by blast
qed

lemma card_of_Sigma_cong:
  assumes "⋀x. x ∈ A ⟹ |B x| =o |B' x|"
  shows   "|SIGMA x:A. B x| =o |SIGMA x:A. B' x|"
proof -
  have "∃f. bij_betw f (B x) (B' x)" if "x ∈ A" for x
    using assms that card_of_ordIso by blast
  then obtain f where f: "⋀x. x ∈ A ⟹ bij_betw (f x) (B x) (B' x)"
    by metis
  have "bij_betw (λ(x,y). (x, f x y)) (SIGMA x:A. B x) (SIGMA x:A. B'
x)"
    using f by (fastforce simp: bij_betw_def inj_on_def image_def)
  thus ?thesis
    by (rule card_of_ordIsoI)
qed

lemma Cfinite_cases:
  assumes "Cfinite c"
  obtains n :: nat where "(c, natLeq_on n) ∈ ordIso"
proof -
  from assms have "card_of (Field c) =o natLeq_on (card (Field c))"
    by (simp add: cfinite_def finite_imp_card_of_natLeq_on)
  with that[of "card (Field c)"] show ?thesis
    using assms card_of_unique ordIso_transitive by blast
qed

lemma empty_nat_ordIso_czero: "({} :: (nat × nat) set) =o czero"
proof -
  have "({} :: (nat × nat) set) =o |{} :: nat set|"
    using finite_imp_card_of_natLeq_on[of "{} :: nat set"] by (simp add:
ordIso_symmetric)
  moreover have "|{} :: nat set| =o czero"
    by (simp add: card_of_ordIso_czero_iff_empty)
  ultimately show "({} :: (nat × nat) set) =o czero"
    using ordIso_symmetric ordIso_transitive by blast
```

```
qed

lemma card_order_on_empty: "card_order_on {} {}"
  unfolding card_order_on_def well_order_on_def linear_order_on_def partial_order_on_def
            preorder_on_def antisym_def trans_def refl_on_def total_on_def
ordLeq_def embed_def
  by (auto intro!: ordLeq_refl)

lemma natLeq_on_plus_ordIso: "natLeq_on (m + n) =o natLeq_on m +c natLeq_on
n"
proof -
  have "{0..<m+n} = {0..<m} ∪ {m..<m+n}"
    by auto
  also have "card_of ({0..<m} ∪ {m..<m+n}) =o card_of ({0..<m} <+> {m..<m+n})"
    by (rule card_of_Un_Plus_ordIso) auto
  also have "card_of ({0..<m} <+> {m..<m+n}) =o card_of {0..<m} +c card_of
{m..<m+n}"
    by (rule Plus_csum)
  also have "card_of {0..<m} +c card_of {m..<m+n} =o natLeq_on m +c natLeq_on
n"
    using finite_imp_card_of_natLeq_on[of "{m..<m+n}"]
    by (intro csum_cong card_of_less) auto
  finally have "|{0..<m+n}| =o natLeq_on m +c natLeq_on n" .
  moreover have "card_of {0..<m+n} =o natLeq_on (m + n)"
    by (rule card_of_less)
  ultimately show ?thesis
    using ordIso_symmetric ordIso_transitive by blast
qed

lemma natLeq_on_1_ord_iso: "natLeq_on 1 =o BNF_Cardinal_Arithmetic.cone"
proof -
  have "|{0..<1::nat}| =o natLeq_on 1"
    by (rule card_of_less)
  hence "|{0::nat}| =o natLeq_on 1"
    by simp
  moreover have "|{0::nat}| =o BNF_Cardinal_Arithmetic.cone"
    by (rule single_cone)
  ultimately show ?thesis
    using ordIso_symmetric ordIso_transitive by blast
qed

lemma cexp_infinite_finite_ordLeq:
  assumes "Cinfinite c" "Cfinite c'"
  shows   "c ^c c' ≤o c"
proof -
  have c: "Card_order c"
    using assms by auto
  from assms obtain n where n: "c' =o natLeq_on n"
    using Cfinite_cases by auto
```

```
  have "c ^c c' =o c ^c natLeq_on n"
    using assms(2) by (intro cexp_cong2 c n) auto
  also have "c ^c natLeq_on n ≤o c"
  proof (induction n)
    case 0
    have "c ^c natLeq_on 0 =o c ^c czero"
      by (intro cexp_cong2) (use assms in ‹auto simp: empty_nat_ordIso_czero
card_order_on_empty›)
    also have "c ^c czero =o BNF_Cardinal_Arithmetic.cone"
      by (rule cexp_czero)
    also have "BNF_Cardinal_Arithmetic.cone ≤o c"
      using assms by (simp add: Cfinite_cone Cfinite_ordLess_Cinfinite
ordLess_imp_ordLeq)
    finally show ?case .
  next
    case (Suc n)
    have "c ^c natLeq_on (Suc n) =o c ^c (natLeq_on n +c natLeq_on 1)"
      using assms natLeq_on_plus_ordIso[of n 1]
      by (intro cexp_cong2) (auto simp: natLeq_on_Card_order intro: ordIso_symmetric)
    also have "c ^c (natLeq_on n +c natLeq_on 1) =o c ^c natLeq_on n *c
c ^c natLeq_on 1"
      by (rule cexp_csum)
    also have "c ^c natLeq_on n *c c ^c natLeq_on 1 ≤o c *c c"
    proof (rule cprod_mono)
      show "c ^c natLeq_on n ≤o c"
        by (rule Suc.IH)
      have "c ^c natLeq_on 1 =o c ^c BNF_Cardinal_Arithmetic.cone"
        by (intro cexp_cong2 c natLeq_on_1_ord_iso natLeq_on_Card_order)
      also have "c ^c BNF_Cardinal_Arithmetic.cone =o c"
        by (intro cexp_cone c)
      finally show "c ^c natLeq_on 1 ≤o c"
        by (rule ordIso_imp_ordLeq)
    qed
    also have "c *c c =o c"
      using assms(1) by (rule cprod_infinite)
    finally show "c ^c natLeq_on (Suc n) ≤o c" .
  qed
  finally show ?thesis .
qed

lemma cexp_infinite_finite_ordIso:
  assumes "Cinfinite c" "Cfinite c'" "BNF_Cardinal_Arithmetic.cone ≤o
c'"
  shows   "c ^c c' =o c"
proof -
  have c: "Card_order c"
    using assms by auto
  have "c =o c ^c BNF_Cardinal_Arithmetic.cone"
    by (rule ordIso_symmetric, rule cexp_cone) fact
```

```
  also have "c ^c BNF_Cardinal_Arithmetic.cone ≤o c ^c c'"
    by (intro cexp_mono2 c assms Card_order_cone) (use cone_not_czero
in auto)
  finally have "c ≤o c ^c c'" .
  moreover have "c ^c c' ≤o c"
    by (rule cexp_infinite_finite_ordLeq) fact+
  ultimately show ?thesis
    by (simp add: ordIso_iff_ordLeq)
qed

lemma Cfinite_ordLeq_Cinfinite:
  assumes "Cfinite c" "Cinfinite c'"
  shows   "c ≤o c'"
  using assms Cfinite_ordLess_Cinfinite ordLess_imp_ordLeq by blast

lemma cfinite_card_of_iff [simp]: "BNF_Cardinal_Arithmetic.cfinite (card_of
X) ⟷ finite X"
  by (simp add: cfinite_def)

lemma cinfinite_card_of_iff [simp]: "BNF_Cardinal_Arithmetic.cinfinite
(card_of X) ⟷ infinite X"
  by (simp add: cinfinite_def)

lemma Func_conv_PiE: "Func A B = PiE A (λ_. B)"
  by (auto simp: Func_def PiE_def extensional_def)

lemma finite_Func [intro]:
  assumes "finite A" "finite B"
  shows   "finite (Func A B)"
  using assms unfolding Func_conv_PiE by (intro finite_PiE)

lemma ordLeq_antisym: "(c, c') ∈ ordLeq ⟹ (c', c) ∈ ordLeq ⟹ (c,
c') ∈ ordIso"
  using ordIso_iff_ordLeq by auto

lemma cmax_cong:
  assumes "(c1, c1') ∈ ordIso" "(c2, c2') ∈ ordIso" "Card_order c1" "Card_order
c2"
  shows   "cmax c1 c2 =o cmax c1' c2'"
proof -
  have [intro]: "Card_order c1'" "Card_order c2'"
    using assms Card_order_ordIso2 by auto
  have "c1 ≤o c2 ∨ c2 ≤o c1"
    by (intro ordLeq_total) (use assms in auto)
  thus ?thesis
  proof
    assume le: "c1 ≤o c2"
    with assms have le': "c1' ≤o c2'"
      by (meson ordIso_iff_ordLeq ordLeq_transitive)
```

```
    have "cmax c1 c2 =o c2"
      by (rule cmax2) (use le assms in auto)
    moreover have "cmax c1' c2' =o c2'"
      by (rule cmax2) (use le' assms in auto)
    ultimately show ?thesis
      using assms ordIso_symmetric ordIso_transitive by metis
  next
    assume le: "c2 ≤o c1"
    with assms have le': "c2' ≤o c1'"
      by (meson ordIso_iff_ordLeq ordLeq_transitive)
    have "cmax c1 c2 =o c1"
      by (rule cmax1) (use le assms in auto)
    moreover have "cmax c1' c2' =o c1'"
      by (rule cmax1) (use le' assms in auto)
    ultimately show ?thesis
      using assms ordIso_symmetric ordIso_transitive by metis
  qed
qed
```

1.2 The set of finite subsets

We define an operator *FinPow(X)* that, given a set *X*, returns the set of all finite subsets of that set. For finite *X*, this is boring since it is obviously just the power set. For infinite *X*, it is however a useful concept to have.

We will show that if *X* is infinite then the cardinality of *FinPow(X)* is exactly the same as that of *X*.

```
definition FinPow :: "'a set ⇒ 'a set set" where
  "FinPow X = {Y. Y ⊆ X ∧ finite Y}"

lemma finite_FinPow [intro]: "finite A ⟹ finite (FinPow A)"
  by (auto simp: FinPow_def)

lemma in_FinPow_iff: "Y ∈ FinPow X ⟷ Y ⊆ X ∧ finite Y"
  by (auto simp: FinPow_def)

lemma FinPow_subseteq_Pow: "FinPow X ⊆ Pow X"
  unfolding FinPow_def by blast

lemma FinPow_eq_Pow: "finite X ⟹ FinPow X = Pow X"
  unfolding FinPow_def using finite_subset by blast

theorem card_of_FinPow_infinite:
  assumes "infinite A"
  shows   "|FinPow A| =o |A|"
proof -
  have "set ` lists A = FinPow A"
    using finite_list[where ?'a = 'a] by (force simp: FinPow_def)
  hence "|FinPow A| ≤o |lists A|"
```

```
    by (metis card_of_image)
  also have "|lists A| =o |A|"
    using assms by (rule card_of_lists_infinite)
  finally have "|FinPow A| ≤o |A|" .
  moreover have "inj_on (λx. {x}) A" "(λx. {x}) ` A ⊆ FinPow A"
    by (auto simp: inj_on_def FinPow_def)
  hence "|A| ≤o |FinPow A|"
    using card_of_ordLeq by blast
  ultimately show ?thesis
    by (simp add: ordIso_iff_ordLeq)
qed
```

1.3 The set of functions with finite support

Next, we define an operator $\text{Func\_finsupp}_z(A, B)$ that, given sets A and B and an element z ∈ B, returns the set of functions f : A → B that have *finite support*, i.e. that map all but a finite subset of A to z.

```
definition Func_finsupp :: "'b ⇒ 'a set ⇒ 'b set ⇒ ('a ⇒ 'b) set" where
  "Func_finsupp z A B = {f∈A → B. (∀x. x ∉ A ⟶ f x = z) ∧ finite {x.
f x ≠ z}}"

lemma bij_betw_Func_finsup_Func_finite:
  assumes "finite A"
  shows   "bij_betw (λf. restrict f A) (Func_finsupp z A B) (Func A B)"
  by (rule bij_betwI[of _ _ _ "λf x. if x ∈ A then f x else z"])
     (use assms in ‹auto simp: Func_finsupp_def Func_def›)

lemma eqpoll_Func_finsup_Func_finite: "finite A ⟹ Func_finsupp z A
B ≈ Func A B"
  by (meson bij_betw_Func_finsup_Func_finite eqpoll_def)

lemma card_of_Func_finsup_finite: "finite A ⟹ |Func_finsupp z A B|
=o |B| ^c |A|"
  using eqpoll_Func_finsup_Func_finite
  by (metis Field_card_of cexp_def eqpoll_imp_card_of_ordIso)
```

The cases where A and B are both finite or $B = \{0\}$ or $A = \emptyset$ are of course trivial.

Perhaps not completely obviously, it turns out that in all other cases, the cardinality of $\text{Func\_finsupp}_z(A, B)$ is exactly $\max(|A|, |B|)$.

```
theorem card_of_Func_finsupp_infinite:
  assumes "z ∈ B" and "B - {z} ≠ {}" and "A ≠ {}"
  assumes "infinite A ∨ infinite B"
  shows   "|Func_finsupp z A B| =o cmax |A| |B|"
proof -
  have inf_cmax: "Cinfinite (cmax |A| |B| )"
    using assms by (simp add: Card_order_cmax cinfinite_def finite_cmax)
```

```
  have "bij_betw (λf. ({x. f x ≠ z}, restrict f {x. f x ≠ z}))
          (Func_finsupp z A B) (SIGMA X:FinPow A. Func X (B - {z}))"
    by (rule bij_betwI[of _ _ _ "λ(X,f) x. if x ∈ -X ∪ -A then z else
f x"])
       (fastforce simp: Func_def Func_finsupp_def FinPow_def fun_eq_iff
‹z ∈ B› split: if_splits)+
  hence "|Func_finsupp z A B| =o |SIGMA X:FinPow A. Func X (B - {z})|"
    by (rule card_of_ordIsoI)

  also have "|SIGMA X:FinPow A. Func X (B - {z})| =o cmax |A| |B|"
  proof (rule ordLeq_antisym)
    show "|SIGMA X:FinPow A. Func X (B - {z})| ≤o cmax |A| |B|"
    proof (intro card_of_Sigma_ordLeq_infinite_Field ballI)
      show "infinite (Field (cmax |A| |B| ))"
        using assms by (simp add: finite_cmax)
    next
      show "Card_order (cmax |A| |B| )"
        by (intro Card_order_cmax) auto
    next
      show "|FinPow A| ≤o cmax |A| |B|"
      proof (cases "finite A")
        assume "infinite A"
        hence "|FinPow A| =o |A|"
          by (rule card_of_FinPow_infinite)
        also have "|A| ≤o cmax |A| |B|"
          by (simp add: ordLeq_cmax)
        finally show ?thesis .
      next
        assume A: "finite A"
        have "finite (FinPow A)"
          using A by auto
        thus "|FinPow A| ≤o cmax |A| |B|"
          using A by (intro Cfinite_ordLeq_Cinfinite inf_cmax) auto
      qed
    next
      show "|Func X (B - {z})| ≤o cmax |A| |B|" if "X ∈ FinPow A" for
X
      proof (cases "finite B")
        case True
        have "finite X"
          using that by (auto simp: FinPow_def)
        hence "finite (Func X (B - {z}))"
          using True by blast
        with inf_cmax show ?thesis
          by (intro Cfinite_ordLeq_Cinfinite) auto
      next
        case False
        have "|Func X (B - {z})| =o |B - {z}| ^c |X|"
          by (rule card_of_Func)
```

```
        also have "|B - {z}| ^c |X| ≤o |B - {z}|"
          by (rule cexp_infinite_finite_ordLeq) (use False that in ‹auto
simp: FinPow_def›)
        also have "|B - {z}| =o |B|"
          using False by (simp add: infinite_card_of_diff_singl)
        also have "|B| ≤o cmax |A| |B|"
          by (simp add: ordLeq_cmax)
        finally show ?thesis .
      qed
    qed
  next
    have "cmax |A| |B| =o |A| *c |B - {z}|"
    proof (cases "|A| ≤o |B|")
      case False
      have "¬|B - {z}| =o czero"
        using ‹B - {z} ≠ {}› by (subst card_of_ordIso_czero_iff_empty)
auto
      from False and assms have "infinite A"
        using ordLeq_finite_infinite by blast
      from False have "|B| ≤o |A|"
        by (simp add: ordLess_imp_ordLeq)
      have "|B - {z}| ≤o |B|"
        by (rule card_of_mono1) auto
      also note ‹|B| ≤o |A|›
      finally have "|A| *c |B - {z}| =o |A|"
        using ‹infinite A› ‹¬|B - {z}| =o czero› by (intro cprod_infinite1')
auto
      moreover have "cmax |A| |B| =o |A|"
        using ‹|B| ≤o |A|› by (simp add: cmax1)
      ultimately show ?thesis
        using ordIso_symmetric ordIso_transitive by blast
    next
      case True
      from True and assms have "infinite B"
        using card_of_ordLeq_finite by blast
      have "|A| *c |B - {z}| =o |A| *c |B|"
        using ‹infinite B›  by (intro cprod_cong2) (simp add: infinite_card_of_diff_singl)
      also have "|A| *c |B| =o |B|"
        using True ‹infinite B› assms(3)
        by (simp add: card_of_ordIso_czero_iff_empty cprod_infinite2')
      also have "|B| =o cmax |A| |B|"
        using True by (meson card_of_Card_order cmax2 ordIso_symmetric)
      finally show ?thesis
        using ordIso_symmetric ordIso_transitive by blast
    qed
    also have "|A| *c |B - {z}| =o |A × (B - {z})|"
      by (metis Field_card_of card_of_refl cprod_def)
    also have "|A × (B - {z})| ≤o |SIGMA X:(λx. {x})`A. B - {z}|"
      by (intro card_of_Sigma_mono[of "λx. {x}"]) auto
```

```
    also have "|SIGMA X:(λx. {x})`A. B - {z}| =o |SIGMA X:(λx. {x})`A.
Func X (B - {z})|"
    proof (rule card_of_Sigma_cong; safe)
      fix x assume x: "x ∈ A"
      have "|Func {x} (B - {z})| =o |B - {z}| ^c |{x}|"
        by (simp add: card_of_Func)
      also have "|B - {z}| ^c |{x}| =o |B - {z}| ^c BNF_Cardinal_Arithmetic.cone"
        by (intro cexp_cong2) (auto simp: single_cone)
      also have "|B - {z}| ^c Wellorder_Constructions.cone =o |B - {z}|"
        using card_of_Card_order cexp_cone by blast
      finally show "|B - {z}| =o |Func {x} (B - {z})|"
        using ordIso_symmetric by blast
    qed
    also have "|SIGMA X:(λx. {x})`A. Func X (B - {z})| ≤o |SIGMA X:FinPow
A. Func X (B - {z})|"
      by (rule card_of_Sigma_mono) (auto simp: FinPow_def)
    finally show "cmax |A| |B| ≤o |SIGMA X:FinPow A. Func X (B - {z})|"
.
  qed
  finally show ?thesis .
qed

end
```

2 The Cardinality of the Continuum

```
theory Cardinality_Continuum
  imports Complex_Main Cardinality_Continuum_Library
begin
```

2.1 $|\mathbb{R}| \leq |2^{\mathbb{Q}}|$ via Dedekind cuts

```
lemma le_cSup_iff:
  fixes A :: "'a :: conditionally_complete_linorder set"
  assumes "A ≠ {}" "bdd_above A"
  shows   "Sup A ≥ c ⟷ (∀x<c. ∃y∈A. y > x)"
  using assms by (meson less_cSup_iff not_le_imp_less order_less_irrefl
order_less_le_trans)
```

We show that the function mapping a real number to all the rational numbers below it is an injective map from the reals to $2^{\mathbb{Q}}$. This is the same idea that is used in the Dedekind cut definition of the reals.

```
lemma inj_Dedekind_cut:
  fixes f :: "real ⇒ rat set"
  defines "f ≡ (λx::real. {r::rat. of_rat r < x})"
  shows   "inj f"
proof
  fix x y :: real
```

```
  assume "f x = f y"

  have *: "Sup (real_of_rat ` {r. real_of_rat r < z}) = z" for z :: real
  proof -
    have "real_of_rat ` {r. real_of_rat r < z} = {r∈ℚ. r < z}"
      by (auto elim!: Rats_cases)
    also have "Sup ... = z"
    proof (rule antisym)
      have "{r ∈ ℚ. r < z} ≠ {}"
        using Rats_no_bot_less less_eq_real_def by blast
      hence "Sup {r∈ℚ. r < z} ≤ Sup {..z}"
        by (rule cSup_subset_mono) auto
      also have "... = z"
        by simp
      finally show "Sup {r∈ℚ. r < z} ≤ z" .
    next
      show "Sup {r∈ℚ. r < z} ≥ z"
      proof (subst le_cSup_iff)
        show "{r∈ℚ. r < z} ≠ {}"
          using Rats_no_bot_less less_eq_real_def by blast
        show "∀y<z. ∃r∈{r∈ℚ. r < z}. y < r"
          using Rats_dense_in_real by fastforce
        show "bdd_above {r ∈ ℚ. r < z}"
          by (rule bdd_aboveI[of _ z]) auto
      qed
    qed
    finally show ?thesis .
  qed

  from ‹f x = f y› have "Sup (real_of_rat ` f x) = Sup (real_of_rat `
f y)"
    by simp
  thus "x = y"
    by (simp only: * f_def)
qed
```

2.2 $2^{|\mathbb{N}|} \leq |\mathbb{R}|$ via ternary fractions

For the other direction, we construct an injective function that maps a set of natural numbers *A* to a real number by constructing a ternary decimal number of the form $d_0.d_1d_2d_3\ldots$ where d_m is 1 if $m \in A$ and 0 otherwise.

We will first show a few more general results about such *n*-ary fraction expansions.

```
lemma geometric_sums':
  fixes c :: "'a :: real_normed_field"
  assumes "norm c < 1"
  shows   "(λn. c ^ (n + m)) sums (c ^ m / (1 - c))"
proof -
```

```
  have "(λn. c ^ m * c ^ n) sums (c ^ m * (1 / (1 - c)))"
    by (intro sums_mult geometric_sums assms)
  thus ?thesis
    by (simp add: power_add field_simps)
qed

lemma summable_nary_fraction:
  fixes d :: real and f :: "nat ⇒ real"
  assumes "⋀n. norm (f n) ≤ c" "d > 1"
  shows   "summable (λn. f n / d ^ n)"
proof (rule summable_comparison_test)
  show "∃N. ∀n≥N. norm (f n / d ^ n :: real) ≤ c * (1 / d) ^ n"
    using assms by (intro exI[of _ 0]) (auto simp: field_simps)
  show "summable (λn. c * (1 / d) ^ n :: real)"
    using assms by (intro summable_mult summable_geometric) auto
qed
```

Consider two n-ary fraction expansions $u = u_1.u_2u_3\ldots$ and $v = v_1.v_2v_3\ldots$ with $n \geq 2$. Suppose that all the u_i and v_i are between 0 and $n - 2$ (i.e. the highest digit does not occur). Then u and v are equal if and only if all $u_i = v_i$ for all i.

Note that without the additional restriction the result does not hold, as e.g. the decimal numbers 0.2 and $0.1\overline{9}$ are equal.

The reasoning boils down to showing that if m is the smallest index where the two sequences differ, then $|u - v| \geq \frac{1}{d-1} > 0$.

```
lemma nary_fraction_unique:
  fixes u v :: "nat ⇒ nat"
  assumes f_eq: "(∑n. real (u n) / real d ^ n) = (∑n. real (v n) /
real d ^ n)"
  assumes uv: "⋀n. u n ≤ d - 2" "⋀n. v n ≤ d - 2" and d: "d ≥ 2"
  shows   "u = v"
proof -
  define f :: "(nat ⇒ nat) ⇒ real" where
    "f = (λu. ∑n. real (u n) / real d ^ n)"

  have "u m = v m" for m
  proof (induction m rule: less_induct)
    case (less m)
    show "u m = v m"
    proof (rule ccontr)
      assume "u m ≠ v m"

      show False
        using ‹u m ≠ v m› uv less.IH f_eq
      proof (induction "u m" "v m" arbitrary: u v rule: linorder_wlog)
        case (sym u v)
        from sym(1)[of v u] sym(2-) show ?case
          by (simp add: eq_commute)
```

```
      next
        case (le u v)
        have uv': "real (u n) ≤ real d - 2" "real (v n) ≤ real d - 2"
for n
          by (metis d of_nat_diff of_nat_le_iff of_nat_numeral le(3,4))+
        have "f u - f v - (real (u m) - real (v m)) / real d ^ m ≤
                (real d - 2) * ((1 / real d) ^ m / (real d - 1))"
        proof (rule sums_le)
          have "(λn. (real (u n) - real (v n)) / real d ^ n) sums (f u
- f v)"
            unfolding diff_divide_distrib f_def using le d uv'
            by (intro sums_diff summable_sums summable_nary_fraction[where
c = "real d - 2"]) auto
          hence "(λn. (real (u (n + m)) - real (v (n + m))) / real d ^
(n + m)) sums
                  (f u - f v - (∑n<m. (real (u n) - real (v n)) / real
d ^ n))"
            by (rule sums_split_initial_segment)
          also have "(∑n<m. (real (u n) - real (v n)) / real d ^ n) =
0"
            by (intro sum.neutral) (use le in auto)
          finally have "(λn. (real (u (n + m)) - real (v (n + m))) / real
d ^ (n + m)) sums (f u - f v)"
            by simp
          thus "(λn. (real (u (Suc n + m)) - real (v (Suc n + m))) / real
d ^ (Suc n + m)) sums
                  (f u - f v - (real (u m) - real (v m)) / real d ^ m)"
            by (subst sums_Suc_iff) auto
        next
          have "(λn. (real d - 2) * ((1 / real d) ^ (n + Suc m))) sums
                 ((real d - 2) * ((1 / real d) ^ Suc m / (1 - 1 / real
d)))"
            using d by (intro sums_mult geometric_sums') auto
          thus "(λn. (real d - 2) * ((1 / real d) ^ (n + Suc m))) sums
                 ((real d - 2) * ((1 / real d) ^ m / (real d - 1)) ::
real)"
            using d by (simp add: sums_iff field_simps)
        next
          fix n :: nat
          have "(real (u (Suc n + m)) - real (v (Suc n + m))) / real d
^ (Suc n + m) ≤
                 ((real d - 2) - 0) / real d ^ (Suc n + m)"
            using uv' by (intro divide_right_mono diff_mono) auto
          thus "(real (u (Suc n + m)) - real (v (Suc n + m))) / real d
^ (Suc n + m) ≤
                 (real d - 2) * (1 / real d) ^ (n + Suc m)"
            by (simp add: field_simps)
        qed
        hence "f u - f v ≤
```

```
              (real d - 2) / (real d - 1) / real d ^ m + (real (u m)
- real (v m)) / real d ^ m"
          by (simp add: field_simps)
        also have "... = ((real d - 2) / (real d - 1) + real (u m) - real
(v m)) / real d ^ m"
          by (simp add: add_divide_distrib diff_divide_distrib)
        also have "... = ((real d - 2) / (real d - 1) + real_of_int (int
(u m) - int (v m))) / real d ^ m"
          using ‹u m ≤ v m› by simp
        also have "... ≤ ((real d - 2) / (real d - 1) + -1) / real d ^
m"
          using le d by (intro divide_right_mono add_mono) auto
        also have "(real d - 2) / (real d - 1) + -1 = -1 / (real d - 1)"
          using d by (simp add: field_simps)
        also have "... < 0"
          using d by (simp add: field_simps)
        finally have "f u - f v < 0"
          using d by (simp add: field_simps)
        with le show False
          by (simp add: f_def)
      qed
    qed
  qed
  thus ?thesis
    by blast
qed
```

It now follows straightforwardly that mapping sets of natural numbers to ternary fraction expansions is indeed injective. For binary fractions, this would not work due to the aforementioned issue.

```
lemma inj_nat_set_to_ternary:
  fixes f :: "nat set ⇒ real"
  defines "f ≡ (λA. ∑n. (if n ∈ A then 1 else 0) / 3 ^ n)"
  shows   "inj f"
proof
  fix A B :: "nat set"
  assume "f A = f B"
  have "(λn. if n ∈ A then 1 else 0 :: nat) = (λn. if n ∈ B then 1 else
0 :: nat)"
  proof (rule nary_fraction_unique)
    have *: "(∑n. (if n ∈ A then 1 else 0) / 3 ^ n) =
             (∑n. real (if n ∈ A then 1 else 0) / real 3 ^ n)"
      for A by (intro suminf_cong) auto
    show "(∑n. real (if n ∈ A then 1 else 0) / real 3 ^ n) =
          (∑n. real (if n ∈ B then 1 else 0) / real 3 ^ n)"
      using ‹f A = f B› by (simp add: f_def *)
  qed auto
  thus "A = B"
    by (metis equalityI subsetI zero_neq_one)
```

qed

2.3 Equipollence proof

```
theorem eqpoll_UNIV_real: "(UNIV :: real set) ≈ (UNIV :: nat set set)"
proof (rule lepoll_antisym)
  show "(UNIV :: nat set set) ≲ (UNIV :: real set)"
    unfolding lepoll_def using inj_nat_set_to_ternary by blast
next
  have "(UNIV :: real set) ≲ (UNIV :: rat set set)"
    unfolding lepoll_def using inj_Dedekind_cut by blast
  also have "... = Pow (UNIV :: rat set)"
    by simp
  also have "... ≈ Pow (UNIV :: nat set)"
    by (rule eqpoll_Pow) (auto simp: infinite_UNIV_char_0 eqpoll_UNIV_nat_iff)
  also have "... = (UNIV :: nat set set)"
    by simp
  finally show "(UNIV :: real set) ≲ (UNIV :: nat set set)" .
qed
```

We can also write the language in the language of cardinal numbers as $|\mathbb{R}| = 2^{\aleph_0}$ using Isabelle's cardinal number library:

```
corollary card_of_UNIV_real: "|UNIV :: real set| =o ctwo ^c natLeq"
proof -
  have "|UNIV :: real set| =o |UNIV :: nat set set|"
    using eqpoll_UNIV_real by (simp add: eqpoll_iff_card_of_ordIso)
  also have "|UNIV :: nat set set| =o cpow |UNIV :: nat set|"
    by (simp add: cpow_def)
  also have "cpow |UNIV :: nat set| =o ctwo ^c |UNIV :: nat set|"
    by (rule cpow_cexp_ctwo)
  also have "ctwo ^c |UNIV :: nat set| =o ctwo ^c natLeq"
    by (intro cexp_cong2) (simp_all add: card_of_nat Card_order_ctwo)
  finally show ?thesis .
qed
```

2.4 Corollaries for real intervals

It is easy to show that any real interval (whether open, closed, or infinite) is equipollent to the full set of real numbers.

```
lemma eqpoll_Ioo_real:
  fixes a b :: real
  assumes "a < b"
  shows   "{a<..<b} ≈ (UNIV :: real set)"
proof -
  have Ioo: "{a<..<b} ≈ {0::real<..<1}" if "a < b" for a b :: real
  proof -
    have "bij_betw (λx. x * (b - a) + a) {0<..<1} {a<..<b}"
    proof (rule bij_betwI[of _ _ _ "λy. (y - a) / (b - a)"], goal_cases)
```

```
      case 1
      show ?case
      proof
        fix x :: real assume x: "x ∈ {0<..<1}"
        have "x * (b - a) + a > 0 + a"
          using x ‹a < b› by (intro add_strict_right_mono mult_pos_pos)
auto
        moreover have "x * (b - a) + a < 1 * (b - a) + a"
          using x ‹a < b› by (intro add_strict_right_mono mult_strict_right_mono)
auto
        ultimately show "x * (b - a) + a ∈ {a<..<b}"
          by simp
      qed
    qed (use ‹a < b› in ‹auto simp: field_simps›)
    thus ?thesis
      using eqpoll_def eqpoll_sym by blast
  qed

  have "{a<..<b} ≈ {-pi/2<..<pi/2}"
    using eqpoll_trans[OF Ioo[of a b] eqpoll_sym[OF Ioo[of "-pi/2" "pi/2"]]]
assms
    by simp
  also have "bij_betw tan {-pi/2<..<pi/2} (UNIV :: real set)"
    by (rule bij_betwI[of _ _ _ arctan])
       (use arctan_lbound arctan_ubound in ‹auto simp: arctan_tan tan_arctan›)
  hence "{-pi/2<..<pi/2} ≈ (UNIV :: real set)"
    using eqpoll_def by blast
  finally show ?thesis .
qed

lemma eqpoll_real:
  assumes "{a::real<..<b} ⊆ X" "a < b"
  shows   "X ≈ (UNIV :: real set)"
  using eqpoll_Ioo_real[OF assms(2)] assms(1)
  by (meson eqpoll_sym lepoll_antisym lepoll_trans1 subset_UNIV subset_imp_lepoll)

lemma eqpoll_Icc_real: "(a::real) < b ⟹ {a..b} ≈ (UNIV :: real set)"
  and eqpoll_Ioc_real: "(a::real) < b ⟹ {a<..b} ≈ (UNIV :: real set)"
  and eqpoll_Ico_real: "(a::real) < b ⟹ {a..<b} ≈ (UNIV :: real set)"
  by (rule eqpoll_real[of a b]; force)+

lemma eqpoll_Ici_real: "{a::real..} ≈ (UNIV :: real set)"
  and eqpoll_Ioi_real: "{a::real<..} ≈ (UNIV :: real set)"
  by (rule eqpoll_real[of a "a + 1"]; force)+

lemma eqpoll_Iic_real: "{..a::real} ≈ (UNIV :: real set)"
  and eqpoll_Iio_real: "{..<a::real} ≈ (UNIV :: real set)"
  by (rule eqpoll_real[of "a - 1" a]; force)+
```

```
lemmas eqpoll_real_ivl =
  eqpoll_Ioo_real eqpoll_Ioc_real eqpoll_Ico_real eqpoll_Icc_real
  eqpoll_Iio_real eqpoll_Iic_real eqpoll_Ici_real eqpoll_Ioi_real

lemmas card_of_ivl_real =
  eqpoll_real_ivl[THEN eqpoll_imp_card_of_ordIso, THEN ordIso_transitive[OF
_ card_of_UNIV_real]]
```

2.5 Corollaries for vector spaces

We will now also show some results about the cardinality of vector spaces. To do this, we use the obvious isomorphism between a vector space *V* with a basis *B* and the set of finite-support functions *B* → *V*.

```
lemma (in vector_space) card_of_span:
  assumes "independent B"
  shows   "|span B| =o |Func_finsupp 0 B (UNIV :: 'a set)|"
proof -
  define f :: "('b ⇒ 'a) ⇒ 'b" where "f = (λg. ∑b | g b ≠ 0. scale
(g b) b)"
  define g :: "'b ⇒ 'b ⇒ 'a" where "g = representation B"
  have "bij_betw g (span B) (Func_finsupp 0 B UNIV)"
  proof (rule bij_betwI[of _ _ _ f], goal_cases)
    case 1
    thus ?case
      by (auto simp: g_def Func_finsupp_def finite_representation intro:
representation_ne_zero)
  next
    case 2
    thus ?case
      by (auto simp: f_def Func_finsupp_def intro!: span_sum span_scale
intro: span_base)
  next
    case (3 x)
    show "f (g x) = x" unfolding g_def f_def
      by (intro sum_nonzero_representation_eq) (use 3 assms in auto)
  next
    case (4 v)
    show "g (f v) = v" unfolding g_def using 4
      by (intro representation_eqI)
         (auto simp: assms f_def Func_finsupp_def intro: span_base
               intro!: sum.cong span_sum span_scale split: if_splits)
  qed
  thus "|span B| =o |Func_finsupp 0 B (UNIV :: 'a set)|"
    by (simp add: card_of_ordIsoI)
qed
```

We can now easily show the following: Let *K* be an infinite field and *V* a non-trivial finite-dimensional *K*-vector space. Then |*V*| = |*K*|.

```
lemma (in vector_space) card_of_span_finite_dim_infinite_field:
  assumes "independent B" and "finite B" and "B ≠ {}" and "infinite
(UNIV :: 'a set)"
  shows   "|span B| =o |UNIV :: 'a set|"
proof -
  have "|span B| =o |Func_finsupp 0 B (UNIV :: 'a set)|"
    by (rule card_of_span) fact
  also have "|Func_finsupp 0 B (UNIV :: 'a set)| =o cmax |B| |UNIV ::
'a set|"
  proof (rule card_of_Func_finsupp_infinite)
    show "UNIV - {0 :: 'a} ≠ {}"
      using assms by (metis finite.emptyI infinite_remove)
  qed (use assms in auto)
  also have "cmax |B| |UNIV :: 'a set| =o |UNIV :: 'a set|"
    using assms by (intro cmax2 ordLeq3_finite_infinite) auto
  finally show ?thesis .
qed
```

Similarly, we can show the following: Let *V* be an infinite-dimensional vector space *V* over some (not necessarily infinite) field *K*. Then $|V| = \max(\dim_K(V), |K|)$.

```
lemma (in vector_space) card_of_span_infinite_dim_infinite_field:
  assumes "independent B" "infinite B"
  shows   "|span B| =o cmax |B| |UNIV :: 'a set|"
proof -
  have "|span B| =o |Func_finsupp 0 B (UNIV :: 'a set)|"
    by (rule card_of_span) fact
  also have "|Func_finsupp 0 B (UNIV :: 'a set)| =o cmax |B| |UNIV ::
'a set|"
  proof (rule card_of_Func_finsupp_infinite)
    have "(1 :: 'a) ∈ UNIV" "(1 :: 'a) ≠ 0"
      by auto
    thus "UNIV - {0 :: 'a} ≠ {}"
      by blast
  qed (use assms in auto)
  finally show "|span B| =o cmax |B| |UNIV :: 'a set|" .
qed

end

theory Cardinality_Euclidean_Space
  imports "HOL-Analysis.Analysis" Cardinality_Continuum
begin
```

With these results, it is now easy to see that any Euclidean space (i.e. finite-dimensional real vector space) has the same cardinality as ℝ:

```
corollary card_of_UNIV_euclidean_space:
  "|UNIV :: 'a :: euclidean_space set| =o ctwo ^c natLeq"
proof -
  have "|span Basis :: 'a set| =o |UNIV :: real set|"
```

```
    by (rule card_of_span_finite_dim_infinite_field)
       (simp_all add: independent_Basis infinite_UNIV_char_0)
  also have "|UNIV :: real set| =o ctwo ^c natLeq"
    by (rule card_of_UNIV_real)
  finally show ?thesis
    by simp
qed
```

In particular, this applies to ℂ and $\mathbb{R}^n$:

```
corollary card_of_complex: "|UNIV :: complex set| =o ctwo ^c natLeq"
  by (rule card_of_UNIV_euclidean_space)

corollary card_of_real_vec: "|UNIV :: (real ^ 'n :: finite) set| =o ctwo
^c natLeq"
  by (rule card_of_UNIV_euclidean_space)

end
```