Verified Construction of Fair Voting Rules

Michael Kirsten

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
kirsten@kit.edu

November 27, 2024

Abstract

Voting rules aggregate multiple individual preferences in order to make a collective decision. Commonly, these mechanisms are expected to respect a multitude of different notions of fairness and reliability, which must be carefully balanced to avoid inconsistencies.

This article contains a formalisation of a framework for the construction of such fair voting rules using composable modules [1, 2]. The framework is a formal and systematic approach for the flexible and verified construction of voting rules from individual composable modules to respect such social-choice properties by construction. Formal composition rules guarantee resulting social-choice properties from properties of the individual components which are of generic nature to be reused for various voting rules. We provide proofs for a selected set of structures and composition rules. The approach can be readily extended in order to support more voting rules, e.g., from the literature by extending the sets of modules and composition rules.

Contents

1 Social-Choice Types **9**
1.1 Auxiliary Lemmas 9
1.2 Preference Relation 10
1.2.1 Definition 10
1.2.2 Ranking 11
1.2.3 Limited Preference 11
1.2.4 Auxiliary Lemmas 16
1.2.5 Lifting Property 26
1.3 Norm 35
1.3.1 Definition 35
1.3.2 Auxiliary Lemmas 35
1.3.3 Common Norms 37
1.3.4 Properties 37
1.3.5 Theorems 37
1.4 Electoral Result 38
1.4.1 Auxiliary Functions 38
1.4.2 Definition 38
1.5 Preference Profile 39
1.5.1 Definition 40
1.5.2 Vote Count 41
1.5.3 Voter Permutations 43
1.5.4 List Representation 47
1.5.5 Preference Counts 52
1.5.6 Condorcet Winner 55
1.5.7 Limited Profile 57
1.5.8 Lifting Property 57
1.6 Social Choice Result 60
1.6.1 Definition 60
1.6.2 Auxiliary Lemmas 60
1.7 Social Welfare Result 63
1.8 Electoral Result Types 64
1.9 Symmetry Properties of Functions 65
1.9.1 Functions 66

1.9.2 Relations for Symmetry Constructions 66
1.9.3 Invariance and Equivariance 66
1.9.4 Auxiliary Lemmas 67
1.9.5 Rewrite Rules 67
1.9.6 Group Actions 72
1.9.7 Invariance and Equivariance 74
1.9.8 Function Composition 79
1.10 Symmetry Properties of Voting Rules 82
1.10.1 Definitions 82
1.10.2 Auxiliary Lemmas 84
1.10.3 Anonymity Lemmas 93
1.10.4 Neutrality Lemmas 101
1.10.5 Homogeneity Lemmas 114
1.10.6 Reversal Symmetry Lemmas 115
1.11 Result-Dependent Voting Rule Properties 120
1.11.1 Property Definitions 120
1.11.2 Interpretations 121

2 Refined Types **122**
2.1 Preference List 122
2.1.1 Well-Formedness 122
2.1.2 Auxiliary Lemmas About Lists 122
2.1.3 Ranking 127
2.1.4 Definition 127
2.1.5 Limited Preference 136
2.1.6 Auxiliary Definitions 140
2.1.7 Auxiliary Lemmas 140
2.1.8 First Occurrence Indices 143
2.2 Preference (List) Profile 145
2.2.1 Definition 145
2.2.2 Refinement Proof 146
2.3 Ordered Relation Type 146
2.4 Alternative Election Type 149

3 Quotient Rules **151**
3.1 Quotients of Equivalence Relations 151
3.1.1 Definitions 151
3.1.2 Well-Definedness 151
3.1.3 Equivalence Relations 154
3.2 Quotients of Election Set Equivalences 156
3.2.1 Auxiliary Lemmas 156
3.2.2 Anonymity Quotient: Grid 159
3.2.3 Homogeneity Quotient: Simplex 168

4 Component Types 189
4.1 Distance . . . 189
4.1.1 Definition . . . 189
4.1.2 Conditions . . . 190
4.1.3 Standard-Distance Property . . . 190
4.1.4 Auxiliary Lemmas . . . 190
4.1.5 Swap Distance . . . 191
4.1.6 Spearman Distance . . . 192
4.1.7 Properties . . . 193
4.2 Votewise Distance . . . 198
4.2.1 Definition . . . 198
4.2.2 Inference Rules . . . 198
4.3 Consensus . . . 204
4.3.1 Definition . . . 204
4.3.2 Consensus Conditions . . . 204
4.3.3 Properties . . . 205
4.3.4 Auxiliary Lemmas . . . 205
4.3.5 Theorems . . . 208
4.4 Electoral Module . . . 212
4.4.1 Definition . . . 212
4.4.2 Auxiliary Definitions . . . 213
4.4.3 Properties . . . 213
4.4.4 Social-Welfare Properties . . . 215
4.4.5 Social-Choice Modules . . . 216
4.4.6 Equivalence Definitions . . . 217
4.4.7 Auxiliary Lemmas . . . 218
4.4.8 Non-Blocking . . . 227
4.4.9 Electing . . . 227
4.4.10 Properties . . . 229
4.4.11 Inference Rules . . . 233
4.4.12 Social-Choice Properties . . . 236
4.5 Electoral Module on Election Quotients . . . 237
4.6 Evaluation Function . . . 238
4.6.1 Definition . . . 238
4.6.2 Property . . . 239
4.6.3 Theorems . . . 239
4.7 Elimination Module . . . 241
4.7.1 General Definitions . . . 241
4.7.2 Social-Choice Definitions . . . 241
4.7.3 Social-Choice Eliminators . . . 241
4.7.4 Soundness . . . 242
4.7.5 Independence of Non-Voters . . . 243
4.7.6 Non-Blocking . . . 246
4.7.7 Non-Electing . . . 247

4.7.8 Inference Rules 248
4.8 Aggregator 251
4.8.1 Definition 251
4.8.2 Properties 251
4.9 Maximum Aggregator 252
4.9.1 Definition 252
4.9.2 Auxiliary Lemma 252
4.9.3 Soundness 253
4.9.4 Properties 253
4.10 Termination Condition 254
4.11 Defer Equal Condition 255

5 Basic Modules 256
5.1 Defer Module 256
5.1.1 Definition 256
5.1.2 Soundness 256
5.1.3 Properties 256
5.2 Elect-First Module 257
5.2.1 Definition 257
5.2.2 Soundness 257
5.3 Consensus Class 258
5.3.1 Definition 258
5.3.2 Consensus Choice 258
5.3.3 Auxiliary Lemmas 259
5.3.4 Consensus Rules 262
5.3.5 Properties 262
5.3.6 Inference Rules 263
5.3.7 Theorems 266
5.4 Distance Rationalization 272
5.4.1 Definitions 273
5.4.2 Standard Definitions 273
5.4.3 Auxiliary Lemmas 274
5.4.4 Soundness 281
5.4.5 Properties 281
5.4.6 Inference Rules 281
5.5 Votewise Distance Rationalization 289
5.5.1 Common Rationalizations 289
5.5.2 Theorems 289
5.5.3 Equivalence Lemmas 291
5.6 Symmetry in Distance-Rationalizable Rules 291
5.6.1 Minimizer Function 291
5.6.2 Minimizer Translation 296
5.6.3 Inference Rules 304
5.7 Distance Rationalization on Election Quotients 307

5.7.1 Distances . . . 307
5.7.2 Consensus and Results . . . 318
5.7.3 Distance Rationalization . . . 322
5.8 Code Generation Interpretations for Results and Properties . 329
5.8.1 Code Lemmas . . . 329
5.8.2 Interpretation Declarations and Constants . . . 331
5.9 Drop Module . . . 331
5.9.1 Definition . . . 331
5.9.2 Soundness . . . 332
5.9.3 Non-Electing . . . 332
5.9.4 Properties . . . 333
5.10 Pass Module . . . 333
5.10.1 Definition . . . 333
5.10.2 Soundness . . . 333
5.10.3 Non-Blocking . . . 334
5.10.4 Non-Electing . . . 335
5.10.5 Properties . . . 335
5.11 Elect Module . . . 341
5.11.1 Definition . . . 342
5.11.2 Soundness . . . 342
5.11.3 Electing . . . 342
5.12 Plurality Module . . . 342
5.12.1 Definition . . . 342
5.12.2 Soundness . . . 345
5.12.3 Non-Blocking . . . 346
5.12.4 Non-Electing . . . 346
5.12.5 Property . . . 347
5.13 Borda Module . . . 352
5.13.1 Definition . . . 352
5.13.2 Soundness . . . 352
5.13.3 Non-Blocking . . . 352
5.13.4 Non-Electing . . . 352
5.14 Condorcet Module . . . 353
5.14.1 Definition . . . 353
5.14.2 Soundness . . . 353
5.14.3 Property . . . 353
5.15 Copeland Module . . . 355
5.15.1 Definition . . . 355
5.15.2 Soundness . . . 355
5.15.3 Lemmas . . . 355
5.15.4 Property . . . 358
5.16 Minimax Module . . . 360
5.16.1 Definition . . . 361
5.16.2 Soundness . . . 361

5.16.3 Lemma . . . 361
5.16.4 Property . . . 362

6 Compositional Structures 365
6.1 Drop- and Pass-Compatibility . . . 365
6.2 Revision Composition . . . 370
6.2.1 Definition . . . 370
6.2.2 Soundness . . . 370
6.2.3 Composition Rules . . . 371
6.3 Sequential Composition . . . 374
6.3.1 Definition . . . 374
6.3.2 Soundness . . . 380
6.3.3 Lemmas . . . 381
6.3.4 Composition Rules . . . 385
6.4 Parallel Composition . . . 409
6.4.1 Definition . . . 409
6.4.2 Soundness . . . 409
6.4.3 Composition Rule . . . 410
6.5 Loop Composition . . . 412
6.5.1 Definition . . . 413
6.5.2 Soundness . . . 417
6.5.3 Lemmas . . . 418
6.5.4 Composition Rules . . . 430
6.6 Maximum Parallel Composition . . . 432
6.6.1 Definition . . . 433
6.6.2 Soundness . . . 433
6.6.3 Lemmas . . . 433
6.6.4 Composition Rules . . . 444
6.7 Elect Composition . . . 452
6.7.1 Definition . . . 453
6.7.2 Auxiliary Lemmas . . . 453
6.7.3 Soundness . . . 453
6.7.4 Electing . . . 453
6.7.5 Composition Rule . . . 455
6.8 Defer-One Loop Composition . . . 457
6.8.1 Soundness . . . 458

7 Voting Rules 459
7.1 Plurality Rule . . . 459
7.1.1 Definition . . . 459
7.1.2 Soundness . . . 462
7.1.3 Electing . . . 463
7.1.4 Properties . . . 466
7.2 Borda Rule . . . 468

7.2.1 Definition . . . 468
7.2.2 Soundness . . . 468
7.2.3 Anonymity . . . 468
7.3 Pairwise Majority Rule . . . 469
7.3.1 Definition . . . 469
7.3.2 Soundness . . . 469
7.3.3 Condorcet Consistency . . . 470
7.4 Copeland Rule . . . 470
7.4.1 Definition . . . 470
7.4.2 Soundness . . . 470
7.4.3 Condorcet Consistency . . . 470
7.5 Minimax Rule . . . 471
7.5.1 Definition . . . 471
7.5.2 Soundness . . . 471
7.5.3 Condorcet Consistency . . . 471
7.6 Black's Rule . . . 471
7.6.1 Definition . . . 472
7.6.2 Soundness . . . 472
7.6.3 Condorcet Consistency . . . 472
7.7 Nanson-Baldwin Rule . . . 472
7.7.1 Definition . . . 473
7.7.2 Soundness . . . 473
7.8 Classic Nanson Rule . . . 473
7.8.1 Definition . . . 473
7.8.2 Soundness . . . 473
7.9 Schwartz Rule . . . 474
7.9.1 Definition . . . 474
7.9.2 Soundness . . . 474
7.10 Sequential Majority Comparison . . . 474
7.10.1 Definition . . . 474
7.10.2 Soundness . . . 475
7.10.3 Electing . . . 475
7.10.4 (Weak) Monotonicity . . . 477
7.11 Kemeny Rule . . . 479
7.11.1 Definition . . . 480
7.11.2 Soundness . . . 480
7.11.3 Anonymity . . . 480
7.11.4 Neutrality . . . 480

Chapter 1

Social-Choice Types

1.1 Auxiliary Lemmas

```
theory Auxiliary-Lemmas
  imports Main
begin
```

Summation function is invariant under application of a (bijective) permutation on the elements.

```
lemma sum-comp:
  fixes
    f :: 'x ⇒ ('z :: comm-monoid-add) and
    g :: 'y ⇒ 'x and
    X :: 'x set and
    Y :: 'y set
  assumes bij-betw g Y X
  shows (∑ x ∈ X. f x) = (∑ x ∈ Y. (f ∘ g) x)
  using assms sum.reindex
  unfolding bij-betw-def
  by (metis (no-types))
```

The inversion of a composition of injective functions is equivalent to the composition of the two individual inverted functions.

```
lemma the-inv-comp:
  fixes
    X :: 'x set and
    Y :: 'y set and
    Z :: 'z set and
    f :: 'y ⇒ 'x and
    g :: 'z ⇒ 'y and
    x :: 'x
  assumes
    bij-betw f Y X and
    bij-betw g Z Y and
    x ∈ X
  shows the-inv-into Z (f ∘ g) x = ((the-inv-into Z g) ∘ (the-inv-into Y f)) x
  using assms the-inv-into-comp
  unfolding bij-betw-def
  by metis

end
```

1.2 Preference Relation

```
theory Preference-Relation
  imports Main
begin
```

The very core of the composable modules voting framework: types and functions, derivations, lemmas, operations on preference relations, etc.

1.2.1 Definition

Each voter expresses pairwise relations between all alternatives, thereby inducing a linear order.

```
type-synonym 'a Preference-Relation = 'a rel

type-synonym 'a Vote = 'a set × 'a Preference-Relation

fun is-less-preferred-than :: 'a ⇒ 'a Preference-Relation ⇒ 'a ⇒ bool
        (- ≼- - [50, 1000, 51] 50) where
    a ≼r b = ((a, b) ∈ r)

fun alts-𝒱 :: 'a Vote ⇒ 'a set where
  alts-𝒱 V = fst V

fun pref-𝒱 :: 'a Vote ⇒ 'a Preference-Relation where
  pref-𝒱 V = snd V

lemma lin-imp-antisym:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation
  assumes linear-order-on A r
  shows antisym r
  using assms
  unfolding linear-order-on-def partial-order-on-def
  by simp
```

```
lemma lin-imp-trans:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation
  assumes linear-order-on A r
  shows trans r
  using assms order-on-defs
  by blast
```

1.2.2 Ranking

```
fun rank :: 'a Preference-Relation ⇒ 'a ⇒ nat where
  rank r a = card (above r a)

lemma rank-gt-zero:
  fixes
    r :: 'a Preference-Relation and
    a :: 'a
  assumes
    refl: a ≼_r a and
    fin: finite r
  shows rank r a ≥ 1
proof (unfold rank.simps above-def)
  have a ∈ {b ∈ Field r. (a, b) ∈ r}
    using FieldI2 refl
    by fastforce
  hence {b ∈ Field r. (a, b) ∈ r} ≠ {}
    by blast
  hence card {b ∈ Field r. (a, b) ∈ r} ≠ 0
    by (simp add: fin finite-Field)
  thus 1 ≤ card {b. (a, b) ∈ r}
    using Collect-cong FieldI2 less-one not-le-imp-less
    by (metis (no-types, lifting))
qed
```

1.2.3 Limited Preference

```
definition limited :: 'a set ⇒ 'a Preference-Relation ⇒ bool where
  limited A r ≡ r ⊆ A × A

lemma limited-dest:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation and
    a b :: 'a
  assumes
    a ≼_r b and
    limited A r
  shows a ∈ A ∧ b ∈ A
  using assms
  unfolding limited-def
  by auto

fun limit :: 'a set ⇒ 'a Preference-Relation ⇒ 'a Preference-Relation where
  limit A r = {(a, b) ∈ r. a ∈ A ∧ b ∈ A}

definition connex :: 'a set ⇒ 'a Preference-Relation ⇒ bool where
  connex A r ≡ limited A r ∧ (∀ a ∈ A. ∀ b ∈ A. a ≼_r b ∨ b ≼_r a)

lemma connex-imp-refl:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation
  assumes connex A r
  shows refl-on A r
  using assms
proof (unfold connex-def refl-on-def limited-def, elim conjE conjI, safe)
  fix a :: 'a
  assume a ∈ A
  hence a ≼_r a
    using assms
    unfolding connex-def
    by metis
  thus (a, a) ∈ r
    by simp
qed

lemma lin-ord-imp-connex:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation
  assumes linear-order-on A r
  shows connex A r
proof (unfold connex-def limited-def, safe)
  fix a b :: 'a
  assume (a, b) ∈ r
  moreover have refl-on A r
    using assms partial-order-onD
    unfolding linear-order-on-def
    by safe
  ultimately show
    a ∈ A and
    b ∈ A
    by (simp-all add: refl-on-domain)
next
  fix a b :: 'a
  assume
    a ∈ A and
    b ∈ A and
    ¬ b ≼_r a
  moreover from this
  have (b, a) ∉ r
    by simp
  moreover have refl-on A r
    using assms partial-order-onD
    unfolding linear-order-on-def
    by blast
  ultimately have (a, b) ∈ r
    using assms refl-onD
    unfolding linear-order-on-def total-on-def
    by metis
  thus a ≼_r b
    by simp
qed

lemma connex-antsym-and-trans-imp-lin-ord:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation
  assumes
    connex-r: connex A r and
    antisym-r: antisym r and
    trans-r: trans r
  shows linear-order-on A r
proof (unfold connex-def linear-order-on-def partial-order-on-def
              preorder-on-def refl-on-def total-on-def, safe)
  fix a b :: 'a
  assume (a, b) ∈ r
  thus
    a ∈ A and
    b ∈ A
    using connex-r refl-on-domain connex-imp-refl
    by (metis, metis)
next
  fix a :: 'a
  assume a ∈ A
  thus (a, a) ∈ r
    using connex-r connex-imp-refl refl-onD
    by metis
next
  show trans r
    using trans-r
    by simp
next
  show antisym r
    using antisym-r
    by simp
next
```

```
  fix a b :: 'a
  assume
    a ∈ A and
    b ∈ A and
    (b, a) ∉ r
  moreover with connex-r
  have a ≼_r b ∨ b ≼_r a
    unfolding connex-def
    by metis
  hence (a, b) ∈ r ∨ (b, a) ∈ r
    by simp
  ultimately show (a, b) ∈ r
    by metis
qed

lemma limit-to-limits:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation
  shows limited A (limit A r)
  unfolding limited-def
  by fastforce

lemma limit-presv-connex:
  fixes
    A B :: 'a set and
    r :: 'a Preference-Relation
  assumes
    connex: connex B r and
    subset: A ⊆ B
  shows connex A (limit A r)
proof (unfold connex-def limited-def limit.simps is-less-preferred-than.simps, safe)
  let ?s = {(a, b). (a, b) ∈ r ∧ a ∈ A ∧ b ∈ A}
  fix a b :: 'a
  assume
    a-in-A: a ∈ A and
    b-in-A: b ∈ A and
    not-b-pref-r-a: (b, a) ∉ r
  have b ≼_r a ∨ a ≼_r b
    using a-in-A b-in-A connex connex-def in-mono subset
    by metis
  hence a ≼_?s b ∨ b ≼_?s a
    using a-in-A b-in-A
    by auto
  thus (a, b) ∈ r
    using not-b-pref-r-a
    by simp
qed
```

```
lemma limit-presv-antisym:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation
  assumes antisym r
  shows antisym (limit A r)
  using assms
  unfolding antisym-def
  by simp

lemma limit-presv-trans:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation
  assumes trans r
  shows trans (limit A r)
  unfolding trans-def
  using transE assms
  by auto

lemma limit-presv-lin-ord:
  fixes
    A B :: 'a set and
    r :: 'a Preference-Relation
  assumes
    linear-order-on B r and
    A ⊆ B
  shows linear-order-on A (limit A r)
 using assms connex-antsym-and-trans-imp-lin-ord limit-presv-antisym limit-presv-connex
       limit-presv-trans lin-ord-imp-connex
  unfolding preorder-on-def partial-order-on-def linear-order-on-def
  by metis

lemma limit-presv-prefs:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation and
    a b :: 'a
  assumes
    a ≼_r b and
    a ∈ A and
    b ∈ A
  shows let s = limit A r in a ≼_s b
  using assms
  by simp

lemma limit-rel-presv-prefs:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation and
    a b :: 'a
  assumes (a, b) ∈ limit A r
  shows a ≼_r b
  using mem-Collect-eq assms
  by simp

lemma limit-trans:
  fixes
    A B :: 'a set and
    r :: 'a Preference-Relation
  assumes A ⊆ B
  shows limit A r = limit A (limit B r)
  using assms
  by auto

lemma lin-ord-not-empty:
  fixes r :: 'a Preference-Relation
  assumes r ≠ {}
  shows ¬ linear-order-on {} r
  using assms connex-imp-refl lin-ord-imp-connex refl-on-domain subrelI
  by fastforce

lemma lin-ord-singleton:
  fixes a :: 'a
  shows ∀ r. linear-order-on {a} r ⟶ r = {(a, a)}
proof (clarify)
  fix r :: 'a Preference-Relation
  assume lin-ord-r-a: linear-order-on {a} r
  hence a ≼_r a
    using lin-ord-imp-connex singletonI
    unfolding connex-def
    by metis
  moreover from lin-ord-r-a
  have ∀ (b, c) ∈ r. b = a ∧ c = a
    using connex-imp-refl lin-ord-imp-connex refl-on-domain split-beta
    by fastforce
  ultimately show r = {(a, a)}
    by auto
qed
```

1.2.4 Auxiliary Lemmas

```
lemma above-trans:
  fixes
    r :: 'a Preference-Relation and
    a b :: 'a
  assumes
    trans r and
    (a, b) ∈ r
  shows above r b ⊆ above r a
  using Collect-mono assms transE
  unfolding above-def
  by metis

lemma above-refl:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation and
    a :: 'a
  assumes
    refl-on A r and
     a ∈ A
  shows a ∈ above r a
  using assms refl-onD
  unfolding above-def
  by simp

lemma above-subset-geq-one:
  fixes
    A :: 'a set and
    r r' :: 'a Preference-Relation and
    a :: 'a
  assumes
    linear-order-on A r and
    linear-order-on A r' and
    above r a ⊆ above r' a and
    above r' a = {a}
  shows above r a = {a}
 using assms connex-imp-refl above-refl insert-absorb lin-ord-imp-connex mem-Collect-eq
      refl-on-domain singletonI subset-singletonD
  unfolding above-def
  by metis

lemma above-connex:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation and
    a :: 'a
  assumes
    connex A r and
    a ∈ A
  shows a ∈ above r a
  using assms connex-imp-refl above-refl
  by metis

lemma pref-imp-in-above:
  fixes
```

```
    r :: 'a Preference-Relation and
    a b :: 'a
  shows (a ≼r b) = (b ∈ above r a)
  unfolding above-def
  by simp

lemma limit-presv-above:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation and
    a b :: 'a
  assumes
    b ∈ above r a and
    a ∈ A and
    b ∈ A
  shows b ∈ above (limit A r) a
  using assms pref-imp-in-above limit-presv-prefs
  by metis

lemma limit-rel-presv-above:
  fixes
    A B :: 'a set and
    r :: 'a Preference-Relation and
    a b :: 'a
  assumes b ∈ above (limit B r) a
  shows b ∈ above r a
  using assms limit-rel-presv-prefs mem-Collect-eq pref-imp-in-above
  unfolding above-def
  by metis

lemma above-one:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation
  assumes
    lin-ord-r: linear-order-on A r and
    fin-A: finite A and
    non-empty-A: A ≠ {}
  shows ∃ a ∈ A. above r a = {a} ∧ (∀ a' ∈ A. above r a' = {a'} ⟶ a' = a)
proof −
  obtain n :: nat where
    len-n-plus-one: n + 1 = card A
    using Suc-eq-plus1 antisym-conv2 fin-A non-empty-A card-eq-0-iff
        gr0-implies-Suc le0
    by metis
  have linear-order-on A r ∧ finite A ∧ A ≠ {} ∧ n + 1 = card A
        ⟶ (∃ a ∈ A. above r a = {a})
  proof (induction n arbitrary: A r; clarify)
    case 0
  fix
    A' :: 'a set and
    r' :: 'a Preference-Relation
  assume
    lin-ord-r: linear-order-on A' r' and
    len-A-is-one: 0 + 1 = card A'
  then obtain a :: 'a where
    A' = {a}
    using card-1-singletonE add.left-neutral
    by metis
  hence
    a ∈ A' and
    above r' a = {a}
    using lin-ord-r connex-imp-refl above-refl lin-ord-imp-connex refl-on-domain
    unfolding above-def
    by (blast, fast)
  thus ∃ a' ∈ A'. above r' a' = {a'}
    by metis
next
  case (Suc n)
  fix
    A' :: 'a set and
    r' :: 'a Preference-Relation
  assume
    lin-ord-r: linear-order-on A' r' and
    fin-A: finite A' and
    A-not-empty: A' ≠ {} and
    len-A-n-plus-one: Suc n + 1 = card A'
  then obtain B :: 'a set where
    subset-B-card: card B = n + 1 ∧ B ⊆ A'
    using Suc-inject add-Suc card.insert-remove finite.cases insert-Diff-single
        subset-insertI
    by (metis (mono-tags, lifting))
  then obtain a :: 'a where
    a: A' − B = {a}
   using Suc-eq-plus1 add-diff-cancel-left' fin-A len-A-n-plus-one card-1-singletonE
        card-Diff-subset finite-subset
    by metis
  have ∃ a' ∈ B. above (limit B r') a' = {a'}
   using subset-B-card Suc.IH add-diff-cancel-left' lin-ord-r card-eq-0-iff diff-le-self
        leD lessI limit-presv-lin-ord
    unfolding One-nat-def
    by metis
  then obtain b :: 'a where
    alt-b: above (limit B r') b = {b}
    by blast
  hence b-above: {a'. (b, a') ∈ limit B r'} = {b}
    unfolding above-def
    by metis
```

```
hence b-pref-b: b ≼_r' b
  using CollectD limit-rel-presv-prefs singletonI
  by (metis (lifting))
show ∃ a' ∈ A'. above r' a' = {a'}
proof (cases)
  assume a-pref-r-b: a ≼_r' b
  have refl-A:
    ∀ A'' r'' a' a''.
        refl-on A'' r'' ∧ (a' :: 'a, a'') ∈ r'' ⟶ a' ∈ A'' ∧ a'' ∈ A''
    using refl-on-domain
    by metis
  have ∀ A'' r''. linear-order-on (A'' :: 'a set) r'' ⟶ connex A'' r''
    by (simp add: lin-ord-imp-connex)
  hence refl-A': refl-on A' r'
    using connex-imp-refl lin-ord-r
    by metis
  hence a ∈ A' ∧ b ∈ A'
    using refl-on-domain a-pref-r-b
    by simp
  hence b-in-r: ∀ a'. a' ∈ A' ⟶ b = a' ∨ (b, a') ∈ r' ∨ (a', b) ∈ r'
    using lin-ord-r
    unfolding linear-order-on-def total-on-def
    by metis
  have b-in-lim-B-r: (b, b) ∈ limit B r'
    using alt-b mem-Collect-eq singletonI
    unfolding above-def
    by metis
  have b-wins: {a'. (b, a') ∈ limit B r'} = {b}
    using alt-b
    unfolding above-def
    by (metis (no-types))
  have b-refl: (b, b) ∈ {(a', a''). (a', a'') ∈ r' ∧ a' ∈ B ∧ a'' ∈ B}
    using b-in-lim-B-r
    by simp
  moreover have b-wins-B: ∀ b' ∈ B. b ∈ above r' b'
  using subset-B-card b-in-r b-wins b-refl CollectI Product-Type.Collect-case-prodD
    unfolding above-def
    by fastforce
  moreover have b ∈ above r' a
    using a-pref-r-b pref-imp-in-above
    by metis
  ultimately have b-wins: ∀ a' ∈ A'. b ∈ above r' a'
    using Diff-iff a empty-iff insert-iff
    by (metis (no-types))
  hence ∀ a' ∈ A'. a' ∈ above r' b ⟶ a' = b
    using CollectD lin-ord-r lin-imp-antisym
    unfolding above-def antisym-def
    by metis
  hence ∀ a' ∈ A'. (a' ∈ above r' b) = (a' = b)
```

```
    using b-wins
    by blast
  moreover have above-b-in-A: above r′ b ⊆ A′
    unfolding above-def
    using refl-A′ refl-A
    by auto
  ultimately have above r′ b = {b}
    using alt-b
    unfolding above-def
    by fastforce
  thus ?thesis
    using above-b-in-A
    by blast
next
  assume ¬ a ≼_r′ b
  hence b ≼_r′ a
    using subset-B-card DiffE a lin-ord-r alt-b limit-to-limits limited-dest
          singletonI subset-iff lin-ord-imp-connex pref-imp-in-above
    unfolding connex-def
    by metis
  hence b-smaller-a: (b, a) ∈ r′
    by simp
  have lin-ord-subset-A:
    ∀ B′ B″ r″.
      linear-order-on (B″ :: ′a set) r″ ∧ B′ ⊆ B″
        ⟶ linear-order-on B′ (limit B′ r″)
    using limit-presv-lin-ord
    by metis
  have {a′. (b, a′) ∈ limit B r′} = {b}
    using alt-b
    unfolding above-def
    by metis
  hence b-in-B: b ∈ B
    by auto
  have limit-B: partial-order-on B (limit B r′) ∧ total-on B (limit B r′)
    using lin-ord-subset-A subset-B-card lin-ord-r
    unfolding linear-order-on-def
    by metis
  have
    ∀ A″ r″.
      total-on A″ r″ =
        (∀ a′. (a′ :: ′a) ∉ A″
          ∨ (∀ a″. a″ ∉ A″ ∨ a′ = a″ ∨ (a′, a″) ∈ r″ ∨ (a″, a′) ∈ r″))
    unfolding total-on-def
    by metis
  hence
    ∀ a′ a″.
      a′ ∈ B ⟶ a″ ∈ B
        ⟶ a′ = a″ ∨ (a′, a″) ∈ limit B r′ ∨ (a″, a′) ∈ limit B r′
        using limit-B
        by simp
      hence ∀ a' ∈ B. b ∈ above r' a'
        using limit-rel-presv-prefs pref-imp-in-above singletonD mem-Collect-eq
              lin-ord-r alt-b b-above b-pref-b subset-B-card b-in-B
        by (metis (lifting))
      hence ∀ a' ∈ B. a' ≼r' b
        unfolding above-def
        by simp
      hence b-wins: ∀ a' ∈ B. (a', b) ∈ r'
        by simp
      have trans r'
        using lin-ord-r lin-imp-trans
        by metis
      hence ∀ a' ∈ B. (a', a) ∈ r'
        using transE b-smaller-a b-wins
        by metis
      hence ∀ a' ∈ B. a' ≼r' a
        by simp
      hence nothing-above-a: ∀ a' ∈ A'. a' ≼r' a
        using a lin-ord-r lin-ord-imp-connex above-connex Diff-iff empty-iff insert-iff
              pref-imp-in-above
        by metis
      have ∀ a' ∈ A'. (a' ∈ above r' a) = (a' = a)
        using lin-ord-r lin-imp-antisym nothing-above-a pref-imp-in-above CollectD
        unfolding antisym-def above-def
        by metis
      moreover have above-a-in-A: above r' a ⊆ A'
        using lin-ord-r connex-imp-refl lin-ord-imp-connex mem-Collect-eq refl-on-domain
        unfolding above-def
        by fastforce
      ultimately have above r' a = {a}
        using a
        unfolding above-def
        by blast
      thus ?thesis
        using above-a-in-A
        by blast
    qed
  qed
  hence ∃ a ∈ A. above r a = {a}
    using fin-A non-empty-A lin-ord-r len-n-plus-one
    by blast
  thus ?thesis
    using assms lin-ord-imp-connex pref-imp-in-above singletonD
    unfolding connex-def
    by metis
qed
```

```
lemma above-one-eq:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation and
    a b :: 'a
  assumes
    lin-ord: linear-order-on A r and
    fin-A: finite A and
    not-empty-A: A ≠ {} and
    above-a: above r a = {a} and
    above-b: above r b = {b}
  shows a = b
proof -
  have
    a ≼_r a and
    b ≼_r b
    using above-a above-b singletonI pref-imp-in-above
    by (metis, metis)
  moreover have
    ∃ a' ∈ A. above r a' = {a'} ∧ (∀ a'' ∈ A. above r a'' = {a''} ⟶ a'' = a')
    using lin-ord fin-A not-empty-A
    by (simp add: above-one)
  moreover have connex A r
    using lin-ord
    by (simp add: lin-ord-imp-connex)
  ultimately show a = b
    using above-a above-b limited-dest
    unfolding connex-def
    by metis
qed

lemma above-one-imp-rank-one:
  fixes
    r :: 'a Preference-Relation and
    a :: 'a
  assumes above r a = {a}
  shows rank r a = 1
  using assms
  by simp

lemma rank-one-imp-above-one:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation and
    a :: 'a
  assumes
    lin-ord: linear-order-on A r and
    rank-one: rank r a = 1
  shows above r a = {a}
```

```
proof –
  from lin-ord
  have refl-on A r
    using linear-order-on-def partial-order-onD
    by blast
  moreover from assms
  have a ∈ A
    unfolding rank.simps above-def linear-order-on-def partial-order-on-def
              preorder-on-def total-on-def
    using card-1-singletonE insertI1 mem-Collect-eq refl-onD1
    by metis
  ultimately have a ∈ above r a
    using above-refl
    by fastforce
  with rank-one
  show above r a = {a}
    using card-1-singletonE rank.simps singletonD
    by metis
qed

theorem above-rank:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation and
    a :: 'a
  assumes linear-order-on A r
  shows (above r a = {a}) = (rank r a = 1)
  using assms above-one-imp-rank-one rank-one-imp-above-one
  by metis

lemma rank-unique:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation and
    a b :: 'a
  assumes
    lin-ord: linear-order-on A r and
    fin-A: finite A and
    a-in-A: a ∈ A and
    b-in-A: b ∈ A and
    a-neq-b: a ≠ b
  shows rank r a ≠ rank r b
proof (unfold rank.simps above-def, clarify)
  assume card-eq: card {a'. (a, a') ∈ r} = card {a'. (b, a') ∈ r}
  have refl-r: refl-on A r
    using lin-ord
    by (simp add: lin-ord-imp-connex connex-imp-refl)
  hence rel-refl-b: (b, b) ∈ r
    using b-in-A
    unfolding refl-on-def
    by (metis (no-types))
  have rel-refl-a: (a, a) ∈ r
    using a-in-A refl-r refl-onD
    by (metis (full-types))
  obtain p :: 'a ⇒ bool where
    rel-b: ∀ y. p y = ((b, y) ∈ r)
    using is-less-preferred-than.simps
    by metis
  hence finite (Collect p)
    using refl-r refl-on-domain fin-A rev-finite-subset mem-Collect-eq subsetI
    by metis
  hence finite {a'. (b, a') ∈ r}
    using rel-b
    by (simp add: Collect-mono rev-finite-subset)
  moreover from this
  have finite {a'. (a, a') ∈ r}
    using card-eq card-gt-0-iff rel-refl-b
    by force
  moreover have trans r
    using lin-ord lin-imp-trans
    by metis
  moreover have (a, b) ∈ r ∨ (b, a) ∈ r
    using lin-ord a-in-A b-in-A a-neq-b
    unfolding linear-order-on-def total-on-def
    by metis
  ultimately have sets-eq: {a'. (a, a') ∈ r} = {a'. (b, a') ∈ r}
    using card-eq above-trans card-seteq order-refl
    unfolding above-def
    by metis
  hence (b, a) ∈ r
    using rel-refl-a sets-eq
    by blast
  hence (a, b) ∉ r
    using lin-ord lin-imp-antisym a-neq-b antisymD
    by metis
  thus False
    using lin-ord partial-order-onD sets-eq b-in-A
    unfolding linear-order-on-def refl-on-def
    by blast
qed

lemma above-presv-limit:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation and
    a :: 'a
  shows above (limit A r) a ⊆ A
  unfolding above-def
  by auto
```

1.2.5 Lifting Property

```
definition equiv-rel-except-a :: 'a set ⇒ 'a Preference-Relation ⇒
      'a Preference-Relation ⇒ 'a ⇒ bool where
  equiv-rel-except-a A r r' a ≡
    linear-order-on A r ∧ linear-order-on A r' ∧ a ∈ A ∧
    (∀ a' ∈ A − {a}. ∀ b' ∈ A − {a}. (a' ≼_r b') = (a' ≼_r' b'))

definition lifted :: 'a set ⇒ 'a Preference-Relation ⇒ 'a Preference-Relation ⇒
      'a ⇒ bool where
  lifted A r r' a ≡
    equiv-rel-except-a A r r' a ∧ (∃ a' ∈ A − {a}. a ≼_r a' ∧ a' ≼_r' a)

lemma trivial-equiv-rel:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation
  assumes linear-order-on A r
  shows ∀ a ∈ A. equiv-rel-except-a A r r a
  unfolding equiv-rel-except-a-def
  using assms
  by simp

lemma lifted-imp-equiv-rel-except-a:
  fixes
    A :: 'a set and
    r r' :: 'a Preference-Relation and
    a :: 'a
  assumes lifted A r r' a
  shows equiv-rel-except-a A r r' a
  using assms
  unfolding lifted-def equiv-rel-except-a-def
  by simp

lemma lifted-imp-switched:
  fixes
    A :: 'a set and
    r r' :: 'a Preference-Relation and
    a :: 'a
  assumes lifted A r r' a
  shows ∀ a' ∈ A − {a}. ¬ (a' ≼_r a ∧ a ≼_r' a')
proof (safe)
  fix b :: 'a
  assume
    b-in-A: b ∈ A and
    b-neq-a: b ≠ a and
    b-pref-a: b ≼_r a and
    a-pref-b: a ≼_r' b
  hence
    a-pref-b-rel: (a, b) ∈ r' and
    b-pref-a-rel: (b, a) ∈ r
    by simp-all
  have antisym r
    using assms lifted-imp-equiv-rel-except-a lin-imp-antisym
    unfolding equiv-rel-except-a-def
    by metis
  hence imp-b-eq-a: (b, a) ∈ r ⟶ (a, b) ∈ r ⟶ b = a
    unfolding antisym-def
    by simp
  have ∃ a' ∈ A − {a}. a ≼_r a' ∧ a' ≼_r' a
    using assms
    unfolding lifted-def
    by metis
  then obtain c :: 'a where
    c ∈ A − {a} ∧ a ≼_r c ∧ c ≼_r' a
    by metis
  hence c-eq-r-s-exc-a: c ∈ A − {a} ∧ (a, c) ∈ r ∧ (c, a) ∈ r'
    by simp
  have equiv-r-s-exc-a: equiv-rel-except-a A r r' a
    using assms
    unfolding lifted-def
    by metis
  hence ∀ a' ∈ A − {a}. ∀ b' ∈ A − {a}. ((a', b') ∈ r) = ((a', b') ∈ r')
    unfolding equiv-rel-except-a-def
    by simp
  moreover have ∀ a' b' c'. (a', b') ∈ r ⟶ (b', c') ∈ r ⟶ (a', c') ∈ r
    using equiv-r-s-exc-a
    unfolding equiv-rel-except-a-def linear-order-on-def partial-order-on-def
              preorder-on-def trans-def
    by metis
  ultimately have (b, c) ∈ r'
    using b-in-A b-neq-a b-pref-a-rel c-eq-r-s-exc-a equiv-r-s-exc-a
          insertE insert-Diff
    unfolding equiv-rel-except-a-def
    by metis
  hence (a, c) ∈ r'
    using a-pref-b-rel b-pref-a-rel imp-b-eq-a b-neq-a equiv-r-s-exc-a
          lin-imp-trans transE
    unfolding equiv-rel-except-a-def
    by metis
  thus False
   using c-eq-r-s-exc-a equiv-r-s-exc-a antisymD DiffD2 lin-imp-antisym singletonI
    unfolding equiv-rel-except-a-def
    by metis
qed
```

```
lemma lifted-mono:
  fixes
    A :: 'a set and
    r r' :: 'a Preference-Relation and
    a a' :: 'a
  assumes
    lifted: lifted A r r' a and
    a'-pref-a: a' ≼r a
  shows a' ≼r' a
proof (unfold is-less-preferred-than.simps)
  have a'-pref-a-rel: (a', a) ∈ r
    using a'-pref-a
    by simp
  hence a'-in-A: a' ∈ A
    using lifted connex-imp-refl lin-ord-imp-connex refl-on-domain
    unfolding equiv-rel-except-a-def lifted-def
    by metis
  have rest-eq: ∀ b ∈ A − {a}. ∀ b' ∈ A − {a}. ((b, b') ∈ r) = ((b, b') ∈ r')
    using lifted
    unfolding lifted-def equiv-rel-except-a-def
    by simp
  have ex-lifted: ∃ b ∈ A − {a}. (a, b) ∈ r ∧ (b, a) ∈ r'
    using lifted
    unfolding lifted-def
    by simp
  show (a', a) ∈ r'
  proof (cases a' = a)
    case True
    thus ?thesis
      using connex-imp-refl refl-onD lifted lin-ord-imp-connex
      unfolding equiv-rel-except-a-def lifted-def
      by metis
  next
    case False
    thus ?thesis
      using a'-pref-a-rel a'-in-A rest-eq ex-lifted insertE insert-Diff
            lifted lin-imp-trans lifted-imp-equiv-rel-except-a
      unfolding equiv-rel-except-a-def trans-def
      by metis
  qed
qed

lemma lifted-above-subset:
  fixes
    A :: 'a set and
    r r' :: 'a Preference-Relation and
    a :: 'a
  assumes lifted A r r' a
  shows above r' a ⊆ above r a
```

```
proof (unfold above-def, safe)
  fix a' :: 'a
  assume a-pref-x: (a, a') ∈ r'
  from assms
  have lifted-r: ∃ b ∈ A − {a}. (a, b) ∈ r ∧ (b, a) ∈ r'
    unfolding lifted-def
    by simp
  from assms
  have rest-eq: ∀ b ∈ A − {a}. ∀ b' ∈ A − {a}. ((b, b') ∈ r) = ((b, b') ∈ r')
    unfolding lifted-def equiv-rel-except-a-def
    by simp
  from assms
  have trans-r: ∀ b c d. (b, c) ∈ r ⟶ (c, d) ∈ r ⟶ (b, d) ∈ r
    using lin-imp-trans
    unfolding trans-def lifted-def equiv-rel-except-a-def
    by metis
  from assms
  have trans-s: ∀ b c d. (b, c) ∈ r' ⟶ (c, d) ∈ r' ⟶ (b, d) ∈ r'
    using lin-imp-trans
    unfolding trans-def lifted-def equiv-rel-except-a-def
    by metis
  from assms
  have refl-r: (a, a) ∈ r
    using connex-imp-refl lin-ord-imp-connex refl-onD
    unfolding equiv-rel-except-a-def lifted-def
    by metis
  from a-pref-x assms
  have a' ∈ A
    using connex-imp-refl lin-ord-imp-connex refl-onD2
    unfolding equiv-rel-except-a-def lifted-def
    by metis
  with a-pref-x lifted-r rest-eq trans-r trans-s refl-r
  show (a, a') ∈ r
    using Diff-iff singletonD
    by (metis (full-types))
qed

lemma lifted-above-mono:
  fixes
    A :: 'a set and
    r r' :: 'a Preference-Relation and
    a a' :: 'a
  assumes
    lifted-a: lifted A r r' a and
    a'-in-A-sub-a: a' ∈ A − {a}
  shows above r a' ⊆ above r' a' ∪ {a}
proof (safe)
  fix b :: 'a
  assume
```

```
    b-in-above-r: b ∈ above r a' and
    b-not-in-above-s: b ∉ above r' a'
  have ∀ b' ∈ A − {a}. (b' ∈ above r a') = (b' ∈ above r' a')
    using a'-in-A-sub-a lifted-a
    unfolding lifted-def equiv-rel-except-a-def above-def
    by simp
  thus b = a
    using lifted-a b-not-in-above-s limited-dest lin-ord-imp-connex
          member-remove pref-imp-in-above b-in-above-r
    unfolding lifted-def equiv-rel-except-a-def remove-def connex-def
    by metis
qed

lemma limit-lifted-imp-eq-or-lifted:
  fixes
    A A' :: 'a set and
    r r' :: 'a Preference-Relation and
    a :: 'a
  assumes
    lifted: lifted A' r r' a and
    subset: A ⊆ A'
  shows limit A r = limit A r' ∨ lifted A (limit A r) (limit A r') a
proof −
  have ∀ a' ∈ A − {a}. ∀ b' ∈ A − {a}. (a' ⪯_r b') = (a' ⪯_r' b')
    using lifted subset
    unfolding lifted-def equiv-rel-except-a-def
    by auto
  hence eql-rs:
    ∀ a' ∈ A − {a}. ∀ b' ∈ A − {a}.
        ((a', b') ∈ (limit A r)) = ((a', b') ∈ (limit A r'))
    using DiffD1 limit-presv-prefs limit-rel-presv-prefs
    by simp
  have lin-ord-r-s: linear-order-on A (limit A r) ∧ linear-order-on A (limit A r')
    using lifted subset lifted-def equiv-rel-except-a-def limit-presv-lin-ord
    by metis
  show ?thesis
  proof (cases)
    assume a-in-A: a ∈ A
    thus ?thesis
    proof (cases)
      assume ∃ a' ∈ A − {a}. a ⪯_r a' ∧ a' ⪯_r' a
      thus ?thesis
        using DiffD1 limit-presv-prefs a-in-A eql-rs lin-ord-r-s
        unfolding lifted-def equiv-rel-except-a-def
        by simp
    next
      assume ¬ (∃ a' ∈ A − {a}. a ⪯_r a' ∧ a' ⪯_r' a)
      hence strict-pref-to-a: ∀ a' ∈ A − {a}. ¬ (a ⪯_r a' ∧ a' ⪯_r' a)
        by simp
```

moreover have *not-worse*: $\forall\ a' \in A - \{a\}.\ \neg\ (a' \preceq_r a \wedge a \preceq_{r'} a')$
  using *lifted subset lifted-imp-switched*
  by *fastforce*
moreover have *connex*: *connex A* (*limit A r*) $\wedge$ *connex A* (*limit A r'*)
  using *lifted subset limit-presv-lin-ord lin-ord-imp-connex*
  unfolding *lifted-def equiv-rel-except-a-def*
  by *metis*
moreover have
  $\forall\ A''\ r''.$ *connex* $A''\ r'' =$
    (*limited* $A''\ r''$
      $\wedge\ (\forall\ b\ b'.\ (b :: 'a) \in A'' \longrightarrow b' \in A'' \longrightarrow (b \preceq_{r''} b' \vee b' \preceq_{r''} b)))$
  unfolding *connex-def*
  by (*simp add*: *Ball-def-raw*)
hence *limit-rel-r*:
  limited A (*limit A r*)
    $\wedge\ (\forall\ b\ b'.\ b \in A \wedge b' \in A \longrightarrow (b, b') \in$ *limit A r* $\vee\ (b', b) \in$ *limit A r*)
  using *connex*
  by *simp*
have *limit-imp-rel*: $\forall\ b\ b'\ A''\ r''.\ (b :: 'a, b') \in$ *limit* $A''\ r'' \longrightarrow b \preceq_{r''} b'$
  using *limit-rel-presv-prefs*
  by *metis*
have *limit-rel-s*:
  limited A (*limit A r'*)
    $\wedge\ (\forall\ b\ b'.\ b \in A \wedge b' \in A \longrightarrow (b, b') \in$ *limit A r'* $\vee\ (b', b) \in$ *limit A r'*)
  using *connex*
  unfolding *connex-def*
  by *simp*
ultimately have
  $\forall\ a' \in A - \{a\}.\ a \preceq_r a' \wedge a \preceq_{r'} a' \vee a' \preceq_r a \wedge a' \preceq_{r'} a$
  using *DiffD1 limit-rel-r limit-rel-presv-prefs a-in-A*
  by *metis*
have $\forall\ a' \in A - \{a\}.\ ((a, a') \in$ (*limit A r*)) = ((*a*, *a'*) $\in$ (*limit A r'*))
  using *DiffD1 limit-imp-rel limit-rel-r limit-rel-s a-in-A*
    strict-pref-to-a not-worse
  by *metis*
hence
  $\forall\ a' \in A - \{a\}.$
    (*let q* = *limit A r in* $a \preceq_q a'$) = (*let q* = *limit A r' in* $a \preceq_q a'$)
  by *simp*
moreover have
  $\forall\ a' \in A - \{a\}.\ ((a', a) \in$ (*limit A r*)) = ((*a'*, *a*) $\in$ (*limit A r'*))
  using *a-in-A strict-pref-to-a not-worse DiffD1 limit-rel-presv-prefs*
    limit-rel-s limit-rel-r
  by *metis*
moreover have $(a, a) \in$ (*limit A r*) $\wedge\ (a, a) \in$ (*limit A r'*)
  using *a-in-A connex connex-imp-refl refl-onD*
  by *metis*
ultimately show *?thesis*
  using *eql-rs*

```
        by auto
    qed
  next
    assume a ∉ A
    thus ?thesis
      using limit-to-limits limited-dest subrelI subset-antisym eql-rs
      by auto
  qed
qed

lemma negl-diff-imp-eq-limit:
  fixes
    A A' :: 'a set and
    r r' :: 'a Preference-Relation and
    a :: 'a
  assumes
    change: equiv-rel-except-a A' r r' a and
    subset: A ⊆ A' and
    not-in-A: a ∉ A
  shows limit A r = limit A r'
proof –
  have A ⊆ A' − {a}
    unfolding subset-Diff-insert
    using not-in-A subset
    by simp
  hence ∀ b ∈ A. ∀ b' ∈ A. (b ≼_r b') = (b ≼_r' b')
    using change in-mono
    unfolding equiv-rel-except-a-def
    by metis
  thus ?thesis
    by auto
qed

theorem lifted-above-winner-alts:
  fixes
    A :: 'a set and
    r r' :: 'a Preference-Relation and
    a a' :: 'a
  assumes
    lifted-a: lifted A r r' a and
    a'-above-a': above r a' = {a'} and
    fin-A: finite A
  shows above r' a' = {a'} ∨ above r' a = {a}
proof (cases)
  assume a = a'
  thus ?thesis
    using above-subset-geq-one lifted-a a'-above-a' lifted-above-subset
    unfolding lifted-def equiv-rel-except-a-def
    by metis
```

```
next
  assume a-neq-a': a ≠ a'
  thus ?thesis
  proof (cases)
    assume above r' a' = {a'}
    thus ?thesis
      by simp
  next
    assume a'-not-above-a': above r' a' ≠ {a'}
    have ∀ a'' ∈ A. a'' ≼_r a'
    proof (safe)
      fix b :: 'a
      assume y-in-A: b ∈ A
      hence A ≠ {}
        by blast
      moreover have linear-order-on A r
        using lifted-a
        unfolding equiv-rel-except-a-def lifted-def
        by simp
      ultimately show b ≼_r a'
        using y-in-A a'-above-a' lin-ord-imp-connex pref-imp-in-above
              singletonD limited-dest singletonI
        unfolding connex-def
        by (metis (no-types))
    qed
    moreover have equiv-rel-except-a A r r' a
      using lifted-a
      unfolding lifted-def
      by metis
    moreover have a' ∈ A − {a}
      using a-neq-a' calculation member-remove
            limited-dest lin-ord-imp-connex
      using equiv-rel-except-a-def remove-def connex-def
      by metis
    ultimately have ∀ a'' ∈ A − {a}. a'' ≼_r' a'
      using DiffD1 lifted-a
      unfolding equiv-rel-except-a-def
      by metis
    hence ∀ a'' ∈ A − {a}. above r' a'' ≠ {a''}
      using a'-not-above-a' empty-iff insert-iff pref-imp-in-above
      by metis
    hence above r' a = {a}
      using Diff-iff all-not-in-conv lifted-a above-one singleton-iff fin-A
      unfolding lifted-def equiv-rel-except-a-def
      by metis
    thus above r' a' = {a'} ∨ above r' a = {a}
      by simp
  qed
qed
```

```
theorem lifted-above-winner-single:
  fixes
    A :: 'a set and
    r r' :: 'a Preference-Relation and
    a :: 'a
  assumes
    lifted A r r' a and
    above r a = {a} and
    finite A
  shows above r' a = {a}
  using assms lifted-above-winner-alts
  by metis

theorem lifted-above-winner-other:
  fixes
    A :: 'a set and
    r r' :: 'a Preference-Relation and
    a a' :: 'a
  assumes
    lifted-a: lifted A r r' a and
    a'-above-a': above r' a' = {a'} and
    fin-A: finite A and
    a-not-a': a ≠ a'
  shows above r a' = {a'}
proof (rule ccontr)
  assume not-above-x: above r a' ≠ {a'}
  then obtain b :: 'a where
    b-above-b: above r b = {b}
    using lifted-a fin-A insert-Diff insert-not-empty above-one
    unfolding lifted-def equiv-rel-except-a-def
    by metis
  hence above r' b = {b} ∨ above r' a = {a}
    using lifted-a fin-A lifted-above-winner-alts
    by metis
  moreover have ∀ a''. above r' a'' = {a''} ⟶ a'' = a'
    using all-not-in-conv lifted-a a'-above-a' fin-A above-one-eq
    unfolding lifted-def equiv-rel-except-a-def
    by metis
  ultimately have b = a'
    using a-not-a'
    by presburger
  moreover have b ≠ a'
    using not-above-x b-above-b
    by blast
  ultimately show False
    by simp
qed
```

```
end
```

1.3 Norm

```
theory Norm
  imports HOL−Library.Extended-Real
          HOL−Combinatorics.List-Permutation
          Auxiliary-Lemmas
begin
```

A norm on R to n is a mapping N: $R \mapsto n$ on R that has the following properties for all mappings u (and v) in R to n:

- positive scalability: $N(a * u) = |a| * N(u)$ for all a in R.
- positive semidefiniteness: $N(u) \geq 0$ with $N(u) = 0$ if and only if $u = (0, 0, \ldots, 0)$.
- triangle inequality: $N(u + v) \leq N(u) + N(v)$.

1.3.1 Definition

```
type-synonym Norm = ereal list ⇒ ereal

definition norm :: Norm ⇒ bool where
  norm n ≡ ∀ x :: ereal list. n x ≥ 0 ∧ (∀ i < length x. x!i = 0 ⟶ n x = 0)
```

1.3.2 Auxiliary Lemmas

```
lemma sum-over-image-of-bijection:
  fixes
    A :: 'a set and
    A' :: 'b set and
    f :: 'a ⇒ 'b and
    g :: 'a ⇒ ereal
  assumes bij-betw f A A'
  shows (∑ a ∈ A. g a) = (∑ a' ∈ A'. g (the-inv-into A f a'))
  using assms
proof (induction card A arbitrary: A A')
  case 0
  thus ?case
    using bij-betw-same-card card-0-eq sum.empty sum.infinite
    by metis
next
  case (Suc x)
  fix
    A :: 'a set and
    A′ :: 'b set and
    x :: nat
  assume
    suc-x: Suc x = card A and
    bij-A-A′: bij-betw f A A′
  hence card-A′-from-x: card A′ = Suc x
    using bij-betw-same-card
    by metis
  have x-lt-card-A: x < card A
    using suc-x
    by presburger
  obtain a :: 'a where
    a-in-A: a ∈ A
    using suc-x card-eq-SucD insertI1
    by metis
  hence a-compl-A: insert a (A − {a}) = A
    using insert-absorb
    by simp
  hence
    inj-on-A: inj-on f A and
    img-of-A: A′ = f ‘ A
    using bij-A-A′
    unfolding bij-betw-def
    by (simp, simp)
  hence inj-on f (insert a A)
    using a-compl-A
    by simp
  hence A′-sub-fa: A′ − {f a} = f ‘ (A − {a})
    using img-of-A
    by blast
  hence bij-without-a: bij-betw f (A − {a}) (A′ − {f a})
    using inj-on-A a-compl-A inj-on-insert
    unfolding bij-betw-def
    by (metis (no-types))
  moreover have card-without-a: card (A − {a}) = x
    using suc-x a-in-A
    by simp
  ultimately have card-A′-sub-f-eq-x: card (A′ − {f a}) = x
    using bij-betw-same-card
    by metis
  have (∑ a ∈ A. g a) = (∑ a ∈ A − {a}. g a) + g a
    using x-lt-card-A add.commute card-Diff1-less-iff card-without-a
        insert-Diff insert-Diff-single sum.insert-remove
    by (metis (no-types))
  also have ... = (∑ a′ ∈ A′ − {f a}. g (the-inv-into A f a′))
                  + g (the-inv-into A f (f a))
    using bij-without-a a-in-A bij-A-A′ bij-betw-imp-inj-on the-inv-into-f-f
        A′-sub-fa DiffD1 sum.reindex-cong
```

```
    by (metis (mono-tags, lifting))
  finally show (∑ a ∈ A. g a) = (∑ a′ ∈ A′. g (the-inv-into A f a′))
    using add.commute card-Diff1-less-iff insert-Diff insert-Diff-single lessI
          sum.insert-remove card-A′-from-x card-A′-sub-f-eq-x
    by metis
qed
```

1.3.3 Common Norms

```
fun l-one :: Norm where
  l-one x = (∑ i < length x. |x!i|)
```

1.3.4 Properties

```
definition symmetry :: Norm ⇒ bool where
  symmetry n ≡ ∀ x y. x <~~> y ⟶ n x = n y
```

1.3.5 Theorems

```
theorem l-one-is-sym: symmetry l-one
proof (unfold symmetry-def, safe)
  fix l l′ :: ereal list
  assume perm: l <~~> l′
  then obtain π :: nat ⇒ nat
    where
      perm_π: π permutes {..< length l} and
      l_π: permute-list π l = l′
    using mset-eq-permutation
    by metis
  hence (∑ i < length l. |l′!i|) = (∑ i < length l. |l!(π i)|)
    using permute-list-nth
    by fastforce
  also have ... = (∑ i = 0 ..< length l. |l!(π i)|)
    using lessThan-atLeast0
    by presburger
  also have (λ i. |l!(π i)|) = ((λ i. |l!i|) ∘ π)
    by fastforce
  also have (∑ y = 0 ..< length l. ((λ i. |l!i|) ∘ π) y) =
              (∑ i = 0 ..< length l. |l!i|)
    using perm_π atLeast-upt set-upt sum.permute
    by metis
  also have ... = (∑ i < length l. |l!i|)
    using atLeast0LessThan
    by presburger
  finally have (∑ i < length l. |l′!i|) = (∑ i < length l. |l!i|)
    by metis
  moreover have length l = length l′
    using perm perm-length
    by metis
  ultimately show l-one l = l-one l′
  using l-one.elims
  by metis
qed

end
```

1.4 Electoral Result

```
theory Result
  imports Main
begin
```

An electoral result is the principal result type of the composable modules voting framework, as it is a generalization of the set of winning alternatives from social choice functions. Electoral results are selections of the received (possibly empty) set of alternatives into the three disjoint groups of elected, rejected and deferred alternatives. Any of those sets, e.g., the set of winning (elected) alternatives, may also be left empty, as long as they collectively still hold all the received alternatives.

1.4.1 Auxiliary Functions

```
type-synonym 'r Result = 'r set * 'r set * 'r set
```

A partition of a set A are pairwise disjoint sets that "set equals partition" A. For this specific predicate, we have three disjoint sets in a three-tuple.

```
fun disjoint3 :: 'r Result ⇒ bool where
  disjoint3 (e, r, d) =
    ((e ∩ r = {}) ∧
      (e ∩ d = {}) ∧
      (r ∩ d = {}))

fun set-equals-partition :: 'r set ⇒'r Result ⇒ bool where
  set-equals-partition X (e, r, d) = (e ∪ r ∪ d = X)
```

1.4.2 Definition

A result generally is related to the alternative set A (of type 'a). A result should be well-formed on the alternatives. Also it should be possible to limit a well-formed result to a subset of the alternatives.

Specific result types like social choice results (sets of alternatives) can be realized via sublocales of the result locale.

```
locale result =
  fixes
    well-formed :: 'a set ⇒ ('r Result) ⇒ bool and
    limit :: 'a set ⇒ 'r set ⇒ 'r set
  assumes ∀ (A :: 'a set) (r :: 'r Result).
    (set-equals-partition (limit A UNIV) r ∧ disjoint3 r) ⟶ well-formed A r
```

These three functions return the elect, reject, or defer set of a result.

```
fun (in result) limit𝓡 :: 'a set ⇒ 'r Result ⇒ 'r Result where
  limit𝓡 A (e, r, d) = (limit A e, limit A r, limit A d)

abbreviation elect-r :: 'r Result ⇒ 'r set where
  elect-r r ≡ fst r

abbreviation reject-r :: 'r Result ⇒ 'r set where
  reject-r r ≡ fst (snd r)

abbreviation defer-r :: 'r Result ⇒ 'r set where
  defer-r r ≡ snd (snd r)

end
```

1.5 Preference Profile

```
theory Profile
  imports Preference-Relation
          Auxiliary-Lemmas
          HOL−Library.Extended-Nat
          HOL−Combinatorics.Permutations
begin
```

Preference profiles denote the decisions made by the individual voters on the eligible alternatives. They are represented in the form of one preference relation (e.g., selected on a ballot) per voter, collectively captured in a mapping of voters onto their respective preference relations. If there are finitely many voters, they can be enumerated and the mapping can be interpreted as a list of preference relations. Unlike the common preference profiles in the social-choice sense, the profiles described here consider only the (sub-)set of alternatives that are received.

1.5.1 Definition

A profile contains one ballot for each voter. An election consists of a set of participating voters, a set of eligible alternatives, and a corresponding profile.

```
type-synonym ('a, 'v) Profile = 'v ⇒ ('a Preference-Relation)

type-synonym ('a, 'v) Election = 'a set × 'v set × ('a, 'v) Profile

fun alternatives-ℰ :: ('a, 'v) Election ⇒ 'a set where
  alternatives-ℰ E = fst E

fun voters-ℰ :: ('a, 'v) Election ⇒ 'v set where
  voters-ℰ E = fst (snd E)

fun profile-ℰ :: ('a, 'v) Election ⇒ ('a, 'v) Profile where
  profile-ℰ E = snd (snd E)

fun election-equality :: ('a, 'v) Election ⇒ ('a, 'v) Election ⇒ bool where
  election-equality (A, V, p) (A', V', p') =
        (A = A' ∧ V = V' ∧ (∀ v ∈ V. p v = p' v))
```

A profile on a set of alternatives A and a voter set V consists of ballots that are linear orders on A for all voters in V. A finite profile is one with finitely many alternatives and voters.

```
definition profile :: 'v set ⇒ 'a set ⇒ ('a, 'v) Profile ⇒ bool where
  profile V A p ≡ ∀ v ∈ V. linear-order-on A (p v)

abbreviation finite-profile :: 'v set ⇒ 'a set ⇒ ('a, 'v) Profile ⇒ bool where
  finite-profile V A p ≡ finite A ∧ finite V ∧ profile V A p

abbreviation finite-election :: ('a, 'v) Election ⇒ bool where
  finite-election E ≡ finite-profile (voters-ℰ E) (alternatives-ℰ E) (profile-ℰ E)

definition finite-elections-𝒱 :: ('a, 'v) Election set where
  finite-elections-𝒱 ≡ {E :: ('a, 'v) Election. finite (voters-ℰ E)}

definition finite-elections :: ('a, 'v) Election set where
  finite-elections ≡ {E :: ('a, 'v) Election. finite-election E}

definition well-formed-elections :: ('a, 'v) Election set where
  well-formed-elections ≡ {E. profile (voters-ℰ E) (alternatives-ℰ E) (profile-ℰ E)}
```

— This function subsumes elections with fixed alternatives, finite voters, and a default value for the profile value on non-voters.

```
fun elections-𝒜 :: 'a set ⇒ ('a, 'v) Election set where
  elections-𝒜 A =
        well-formed-elections
```

```
    ∩ {E. alternatives-ℰ E = A ∧ finite (voters-ℰ E)
          ∧ (∀ v. v ∉ voters-ℰ E ⟶ profile-ℰ E v = {})}
```

— Here, we count the occurrences of a ballot in an election, i.e., how many voters specifically chose that exact ballot.

```
fun vote-count :: 'a Preference-Relation ⇒ ('a, 'v) Election ⇒ nat where
  vote-count p E = card {v ∈ (voters-ℰ E). (profile-ℰ E) v = p}
```

1.5.2 Vote Count

```
lemma vote-count-sum:
  fixes E :: ('a, 'v) Election
  assumes
    finite (voters-ℰ E) and
    finite (UNIV :: ('a × 'a) set)
  shows (∑ p ∈ UNIV. vote-count p E) = card (voters-ℰ E)
proof (unfold vote-count.simps)
  have ∀ p. finite {v ∈ voters-ℰ E. profile-ℰ E v = p}
    using assms
    by force
  moreover have disjoint {{v ∈ voters-ℰ E. profile-ℰ E v = p} | p. p ∈ UNIV}
    unfolding disjoint-def
    by blast
  moreover have partition:
    voters-ℰ E = ⋃ {{v ∈ voters-ℰ E. profile-ℰ E v = p} | p. p ∈ UNIV}
    using Union-eq
    by blast
  ultimately have card-eq-sum':
    card (voters-ℰ E) =
        sum card {{v ∈ voters-ℰ E. profile-ℰ E v = p} | p. p ∈ UNIV}
    using card-Union-disjoint[of
            {{v ∈ voters-ℰ E. profile-ℰ E v = p} | p. p ∈ UNIV}]
    by auto
  have finite {{v ∈ voters-ℰ E. profile-ℰ E v = p} | p. p ∈ UNIV}
    using partition assms
    by (simp add: finite-UnionD)
  moreover have
    {{v ∈ voters-ℰ E. profile-ℰ E v = p} | p. p ∈ UNIV} =
        {{v ∈ voters-ℰ E. profile-ℰ E v = p}
            | p. p ∈ UNIV ∧ {v ∈ voters-ℰ E. profile-ℰ E v = p} ≠ {}}
      ∪ {{v ∈ voters-ℰ E. profile-ℰ E v = p}
            | p. p ∈ UNIV ∧ {v ∈ voters-ℰ E. profile-ℰ E v = p} = {}}
    by blast
  moreover have
    {} =
        {{v ∈ voters-ℰ E. profile-ℰ E v = p}
            | p. p ∈ UNIV ∧ {v ∈ voters-ℰ E. profile-ℰ E v = p} ≠ {}}
      ∩ {{v ∈ voters-ℰ E. profile-ℰ E v = p}
            | p. p ∈ UNIV ∧ {v ∈ voters-ℰ E. profile-ℰ E v = p} = {}}
    by blast
  ultimately have
    sum card {{v ∈ voters-ℰ E. profile-ℰ E v = p} | p. p ∈ UNIV} =
        sum card {{v ∈ voters-ℰ E. profile-ℰ E v = p}
            | p. p ∈ UNIV ∧ {v ∈ voters-ℰ E. profile-ℰ E v = p} ≠ {}}
      + sum card {{v ∈ voters-ℰ E. profile-ℰ E v = p}
            | p. p ∈ UNIV ∧ {v ∈ voters-ℰ E. profile-ℰ E v = p} = {}}
    using sum.union-disjoint[of
            {{v ∈ voters-ℰ E. profile-ℰ E v = p}
                | p. p ∈ UNIV ∧ {v ∈ voters-ℰ E. profile-ℰ E v = p} ≠ {}}
            {{v ∈ voters-ℰ E. profile-ℰ E v = p}
                | p. p ∈ UNIV ∧ {v ∈ voters-ℰ E. profile-ℰ E v = p} = {}}]
    by simp
  moreover have
    ∀ X ∈ {{v ∈ voters-ℰ E. profile-ℰ E v = p}
            | p. p ∈ UNIV ∧ {v ∈ voters-ℰ E. profile-ℰ E v = p} = {}}.
        card X = 0
    using card-eq-0-iff
    by fastforce
  ultimately have card-eq-sum:
    card (voters-ℰ E) =
        sum card {{v ∈ voters-ℰ E. profile-ℰ E v = p}
            | p. p ∈ UNIV ∧ {v ∈ voters-ℰ E. profile-ℰ E v = p} ≠ {}}
    using card-eq-sum′
    by simp
  have
    inj-on (λ p. {v ∈ voters-ℰ E. profile-ℰ E v = p})
        {p. {v ∈ voters-ℰ E. profile-ℰ E v = p} ≠ {}}
    unfolding inj-on-def
    by blast
  moreover have
    (λ p. {v ∈ voters-ℰ E. profile-ℰ E v = p})
            ‘ {p. {v ∈ voters-ℰ E. profile-ℰ E v = p} ≠ {}}
        ⊆ {{v ∈ voters-ℰ E. profile-ℰ E v = p}
            | p. p ∈ UNIV ∧ {v ∈ voters-ℰ E. profile-ℰ E v = p} ≠ {}}
    by blast
  moreover have
    (λ p. {v ∈ voters-ℰ E. profile-ℰ E v = p})
            ‘ {p. {v ∈ voters-ℰ E. profile-ℰ E v = p} ≠ {}}
        ⊇ {{v ∈ voters-ℰ E. profile-ℰ E v = p}
            | p. p ∈ UNIV ∧ {v ∈ voters-ℰ E. profile-ℰ E v = p} ≠ {}}
    by blast
  ultimately have
    bij-betw (λ p. {v ∈ voters-ℰ E. profile-ℰ E v = p})
            {p. {v ∈ voters-ℰ E. profile-ℰ E v = p} ≠ {}}
        {{v ∈ voters-ℰ E. profile-ℰ E v = p}
          | p. p ∈ UNIV ∧ {v ∈ voters-ℰ E. profile-ℰ E v = p} ≠ {}}
    unfolding bij-betw-def
    by simp
  hence sum-rewrite:
    (∑ x ∈ {p. {v ∈ voters-ℰ E. profile-ℰ E v = p} ≠ {}}.
          card {v ∈ voters-ℰ E. profile-ℰ E v = x}) =
        sum card {{v ∈ voters-ℰ E. profile-ℰ E v = p}
            | p. p ∈ UNIV ∧ {v ∈ voters-ℰ E. profile-ℰ E v = p} ≠ {}}
    using sum-comp[of
          λ p. {v ∈ voters-ℰ E. profile-ℰ E v = p} - - card]
    unfolding comp-def
    by simp
  have {p. {v ∈ voters-ℰ E. profile-ℰ E v = p} = {}}
        ∩ {p. {v ∈ voters-ℰ E. profile-ℰ E v = p} ≠ {}} = {}
    by blast
  moreover have
    {p. {v ∈ voters-ℰ E. profile-ℰ E v = p} = {}}
      ∪ {p. {v ∈ voters-ℰ E. profile-ℰ E v = p} ≠ {}} = UNIV
    by blast
  ultimately have
    (∑ p ∈ UNIV. card {v ∈ voters-ℰ E. profile-ℰ E v = p}) =
      (∑ x ∈ {p. {v ∈ voters-ℰ E. profile-ℰ E v = p} ≠ {}}.
        card {v ∈ voters-ℰ E. profile-ℰ E v = x})
     + (∑ x ∈ {p. {v ∈ voters-ℰ E. profile-ℰ E v = p} = {}}.
        card {v ∈ voters-ℰ E. profile-ℰ E v = x})
    using assms
        sum.union-disjoint[of
          {p. {v ∈ voters-ℰ E. profile-ℰ E v = p} = {}}
          {p. {v ∈ voters-ℰ E. profile-ℰ E v = p} ≠ {}}]
    using Finite-Set.finite-set add.commute finite-Un
    by (metis (mono-tags, lifting))
  moreover have
    ∀ x ∈ {p. {v ∈ voters-ℰ E. profile-ℰ E v = p} = {}}.
      card {v ∈ voters-ℰ E. profile-ℰ E v = x} = 0
    using card-eq-0-iff
    by fastforce
  ultimately show
    (∑ p ∈ UNIV. card {v ∈ voters-ℰ E. profile-ℰ E v = p}) =
      card (voters-ℰ E)
    using card-eq-sum sum-rewrite
    by simp
qed
```

1.5.3 Voter Permutations

A common action of interest on elections is renaming the voters, e.g., when talking about anonymity.

```
fun rename :: ('v ⇒ 'v) ⇒ ('a, 'v) Election ⇒ ('a, 'v) Election where
  rename π (A, V, p) = (A, π ‘ V, p ∘ (the-inv π))

lemma rename-sound:
  fixes
```

```
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    π :: 'v ⇒ 'v
  assumes
    prof: profile V A p and
    renamed: (A, V', q) = rename π (A, V, p) and
    bij-perm: bij π
  shows profile V' A q
proof (unfold profile-def, safe)
  fix v' :: 'v
  assume v' ∈ V'
  moreover have V' = π ` V
    using renamed
    by simp
  ultimately have ((the-inv π) v') ∈ V
    using UNIV-I bij-perm bij-is-inj bij-is-surj
          f-the-inv-into-f inj-image-mem-iff
    by metis
  thus linear-order-on A (q v')
    using renamed bij-perm prof
    unfolding profile-def
    by simp
qed

lemma rename-prof:
  fixes
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    π :: 'v ⇒ 'v
  assumes
    profile V A p and
    (A, V', q) = rename π (A, V, p) and
    bij π
  shows profile V' A q
  using assms rename-sound
  by metis

lemma rename-finite:
  fixes
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    π :: 'v ⇒ 'v
  assumes
    finite V and
    (A, V', q) = rename π (A, V, p) and
    bij π
  shows finite V′
  using assms
  by simp

lemma rename-inv:
  fixes
    π :: ′v ⇒ ′v and
    A :: ′a set and
    V :: ′v set and
    p :: (′a, ′v) Profile
  assumes bij π
  shows rename π (rename (the-inv π) (A, V, p)) = (A, V, p)
proof −
  have rename π (rename (the-inv π) (A, V, p)) =
        (A, π ‘ (the-inv π) ‘ V, p ∘ (the-inv (the-inv π)) ∘ (the-inv π))
    by simp
  moreover have π ‘ (the-inv π) ‘ V = V
    using assms
    by (simp add: f-the-inv-into-f-bij-betw image-comp)
  moreover have (the-inv (the-inv π)) = π
    using assms surj-def inj-on-the-inv-into surj-imp-inv-eq the-inv-f-f
    unfolding bij-betw-def
    by (metis (mono-tags, opaque-lifting))
  moreover have π ∘ (the-inv π) = id
    using assms f-the-inv-into-f-bij-betw
    by fastforce
  ultimately show rename π (rename (the-inv π) (A, V, p)) = (A, V, p)
    by (simp add: rewriteR-comp-comp)
qed

lemma rename-inj:
  fixes π :: ′v ⇒ ′v
  assumes bij π
  shows inj (rename π)
proof (unfold inj-def split-paired-All rename.simps, safe)
  fix
    A A′ :: ′a set and
    V V′ :: ′v set and
    p p′ :: (′a, ′v) Profile and
    v :: ′v
  assume
    p ∘ the-inv π = p′ ∘ the-inv π and
    π ‘ V = π ‘ V′
  thus
    v ∈ V ⟹ v ∈ V′ and
    v ∈ V′ ⟹ v ∈ V and
    p = p′
    using assms
    by (metis bij-betw-imp-inj-on inj-image-eq-iff,
      metis bij-betw-imp-inj-on inj-image-eq-iff,
      metis bij-betw-the-inv-into bij-is-surj surj-fun-eq)
qed

lemma rename-surj:
  fixes π :: 'v ⇒ 'v
  assumes bij π
  shows
    rename π ‘ well-formed-elections = well-formed-elections and
    rename π ‘ finite-elections = finite-elections
proof (safe)
  fix
    A A' :: 'a set and
    V V' :: 'v set and
    p p' :: ('a, 'v) Profile
  assume wf: (A, V, p) ∈ well-formed-elections
  hence rename (the-inv π) (A, V, p) ∈ well-formed-elections
    using assms bij-betw-the-inv-into rename-sound
    unfolding well-formed-elections-def
    by fastforce
  thus (A, V, p) ∈ rename π ‘ well-formed-elections
    using assms image-eqI rename-inv
    by metis
  assume (A', V', p') = rename π (A, V, p)
  thus (A', V', p') ∈ well-formed-elections
    using rename-sound wf assms
    unfolding well-formed-elections-def
    by fastforce
next
  fix
    A A' :: 'b set and
    V V' :: 'v set and
    p p' :: ('b, 'v) Profile
  assume finite: (A, V, p) ∈ finite-elections
  hence rename (the-inv π) (A, V, p) ∈ finite-elections
    using assms bij-betw-the-inv-into rename-prof rename-finite
    unfolding finite-elections-def
    by fastforce
  thus (A, V, p) ∈ rename π ‘ finite-elections
    using assms image-eqI rename-inv
    by metis
  assume (A', V', p') = rename π (A, V, p)
  thus (A', V', p') ∈ finite-elections
    using rename-sound finite assms
    unfolding finite-elections-def
    by fastforce
qed
```

1.5.4 List Representation

A profile on a voter set that has a natural order can be viewed as a list of ballots.

```
fun to-list :: 'v :: linorder set ⇒ ('a, 'v) Profile ⇒
        ('a Preference-Relation) list where
  to-list V p = (if (finite V)
                  then (map p (sorted-list-of-set V))
                  else [])

lemma map-helper:
  fixes
    f :: 'x ⇒ 'y ⇒ 'z and
    g :: 'x ⇒ 'x and
    h :: 'y ⇒ 'y and
    l :: 'x list and
    l' :: 'y list
  shows map2 f (map g l) (map h l') = map2 (λ x y. f (g x) (h y)) l l'
proof −
  have map2 f (map g l) (map h l') =
          map (λ (x, y). f x y) (map (λ (x, y). (g x, h y)) (zip l l'))
    using zip-map-map
    by metis
  thus ?thesis
    by force
qed

lemma to-list-simp:
  fixes
    i :: nat and
    V :: 'v :: linorder set and
    p :: ('a, 'v) Profile
  assumes i < card V
  shows (to-list V p)!i = p ((sorted-list-of-set V)!i)
  using assms
  by force

lemma to-list-comp:
  fixes
    V :: 'v :: linorder set and
    p :: ('a, 'v) Profile and
    f :: 'a rel ⇒ 'a rel
  shows to-list V (f ∘ p) = map f (to-list V p)
  by simp

lemma set-card-upper-bound:
  fixes
    i :: nat and
    V :: nat set
  assumes
    fin-V: finite V and
    bound-v: ∀ v ∈ V. v < i
  shows card V ≤ i
proof (cases V = {})
  case True
  thus ?thesis
    by simp
next
  case False
  hence Max V ∈ V
    using fin-V
    by simp
  thus ?thesis
    using assms Suc-leI card-le-Suc-Max order-trans
    by metis
qed

lemma sorted-list-of-set-nth-equals-card:
  fixes
    V :: 'v :: linorder set and
    x :: 'v
  assumes
    fin-V: finite V and
    x-V: x ∈ V
  shows sorted-list-of-set V!(card {v ∈ V. v < x}) = x
proof −
  let ?c = card {v ∈ V. v < x} and
      ?set = {v ∈ V. v < x}
  have ∀ v ∈ V. ∃ n. n < card V ∧ (sorted-list-of-set V!n) = v
    using length-sorted-list-of-set sorted-list-of-set-unique in-set-conv-nth fin-V
    by metis
  then obtain φ :: 'v ⇒ nat where
    index-φ: ∀ v ∈ V. φ v < card V ∧ (sorted-list-of-set V!(φ v)) = v
    by metis
  — φ x = ?c, i.e., φ x ≥ ?c and φ x ≤ ?c
  let ?i = φ x
  have inj-φ: inj-on φ V
    using inj-onI index-φ
    by metis
  have ∀ v ∈ V. ∀ v' ∈ V. v < v' ⟶ φ v < φ v'
    using leD linorder-le-less-linear sorted-list-of-set-unique
          sorted-sorted-list-of-set sorted-nth-mono fin-V index-φ
    by metis
  hence ∀ j ∈ {φ v | v. v ∈ ?set}. j < ?i
    using x-V
    by blast
  moreover have fin-img: finite ?set
    using fin-V
    by simp
  ultimately have ?i ≥ card {φ v | v. v ∈ ?set}
    using set-card-upper-bound
    by simp
  hence geq: ?c ≤ ?i
    using inj-φ
    by (simp add: card-image inj-on-subset setcompr-eq-image)
  have sorted-φ:
    ∀ i < card V. ∀ j < card V. i < j
        ⟶ (sorted-list-of-set V!i) < (sorted-list-of-set V!j)
    by (simp add: sorted-wrt-nth-less)
  have leq: ?i ≤ ?c
  proof (rule ccontr, cases ?c < card V)
    case True
    let ?A = λ j. {sorted-list-of-set V!j}
    assume ¬ ?i ≤ ?c
    hence ?c < ?i
      by simp
    hence ∀ j ≤ ?c. sorted-list-of-set V!j ∈ V ∧ sorted-list-of-set V!j < x
      using sorted-φ geq index-φ x-V fin-V set-sorted-list-of-set
            length-sorted-list-of-set nth-mem order.strict-trans1
      by (metis (mono-tags, lifting))
    hence {sorted-list-of-set V!j | j. j ≤ ?c} ⊆ {v ∈ V. v < x}
      by blast
    also have {sorted-list-of-set V!j | j. j ≤ ?c} =
                {sorted-list-of-set V!j | j. j ∈ {0 ..< (?c + 1)}}
      using add.commute
      by auto
    also have ... = (⋃ j ∈ {0 ..< (?c + 1)}. {sorted-list-of-set V!j})
      by blast
    finally have subset: (⋃ j ∈ {0 ..< (?c + 1)}. ?A j) ⊆ {v ∈ V. v < x}
      by simp
    have ∀ i ≤ ?c. ∀ j ≤ ?c.
            i ≠ j ⟶ sorted-list-of-set V!i ≠ sorted-list-of-set V!j
      using True
      by (simp add: nth-eq-iff-index-eq)
    hence ∀ i ∈ {0 ..< (?c + 1)}. ∀ j ∈ {0 ..< (?c + 1)}.
            (i ≠ j ⟶ {sorted-list-of-set V!i} ∩ {sorted-list-of-set V!j} = {})
      by fastforce
    hence disjoint-family-on ?A {0 ..< (?c + 1)}
      unfolding disjoint-family-on-def
      by simp
    moreover have ∀ j ∈ {0 ..< (?c + 1)}. card (?A j) = 1
      by simp
    ultimately have
      card (⋃ j ∈ {0 ..< (?c + 1)}. ?A j) = (∑ j ∈ {0 ..< (?c + 1)}. 1)
      using card-UN-disjoint'
      by fastforce
    hence card (⋃ j ∈ {0 ..< (?c + 1)}. ?A j) = ?c + 1
```

```
      by simp
    hence ?c + 1 ≤ ?c
      using subset card-mono fin-img
      by (metis (no-types, lifting))
    thus False
      by simp
  next
    case False
    thus False
      using x-V index-φ geq order-le-less-trans
      by blast
  qed
  thus ?thesis
    using geq leq x-V index-φ
    by simp
qed

lemma to-list-permutes-under-bij:
  fixes
    π :: 'v :: linorder ⇒ 'v and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assumes bij π
  shows
    let φ = λ i. card {v ∈ π ‘ V. v < π ((sorted-list-of-set V)!i)}
      in (to-list V p) = permute-list φ (to-list (π ‘ V) (λ x. p (the-inv π x)))
proof (cases finite V)
  case False
  — If V is infinite, both lists are empty.
  hence to-list V p = []
    by simp
  moreover have infinite (π ‘ V)
    using False assms bij-betw-finite bij-betw-subset top-greatest
    by metis
  hence to-list (π ‘ V) (λ x. p (the-inv π x)) = []
    by simp
  ultimately show ?thesis
    by simp
next
  case True
  let
    ?q = λ x. p (the-inv π x) and
    ?img = π ‘ V and
    ?n = length (to-list V p) and
    ?perm = λ i. card {v ∈ π ‘ V. v < π ((sorted-list-of-set V)!i)}
    — These are auxiliary statements equating everything with ?n.
  have card-eq: card ?img = card V
    using assms bij-betw-same-card bij-betw-subset top-greatest
    by metis
```

```
also have card-length-V: ?n = ...
  by simp
also have card-length-img: length (to-list ?img ?q) = card ?img
  using True
  by simp
finally have eq-length: length (to-list ?img ?q) = ?n
  by simp
show ?thesis
proof (unfold Let-def permute-list-def, rule nth-equalityI)
  — The lists have equal lengths.
  show
    length (to-list V p) =
        length (map
          (λ i. to-list ?img ?q!(card {v ∈ ?img.
              v < π (sorted-list-of-set V!i)}))
            [0 ..< length (to-list ?img ?q)])
    using eq-length
    by simp
next
  — The ith entries of the lists coincide.
  fix i :: nat
  assume in-bnds: i < ?n
  let ?c = card {v ∈ ?img. v < π (sorted-list-of-set V!i)}
  have map (λ i. (to-list ?img ?q)!?c) [0 ..< ?n]!i =
          p ((sorted-list-of-set V)!i)
  proof -
    have ∀ v. v ∈ ?img ⟶ {v' ∈ ?img. v' < v} ⊆ ?img − {v}
      by blast
    moreover have elem-of-img: π (sorted-list-of-set V!i) ∈ ?img
      using True in-bnds image-eqI nth-mem card-length-V
            length-sorted-list-of-set set-sorted-list-of-set
      by metis
    ultimately have
      {v ∈ ?img. v < π (sorted-list-of-set V!i)}
    ⊆ ?img − {π (sorted-list-of-set V!i)}
      by simp
    hence {v ∈ ?img. v < π (sorted-list-of-set V!i)} ⊂ ?img
      using elem-of-img
      by blast
    moreover have img-card-eq-V-length: card ?img = ?n
      using card-eq card-length-V
      by presburger
    ultimately have card-in-bnds: ?c < ?n
      using True finite-imageI psubset-card-mono
      by (metis (mono-tags, lifting))
    moreover have img-list-map:
      map (λ i. to-list ?img ?q!?c) [0 ..< ?n]!i = to-list ?img ?q!?c
      using in-bnds
      by simp
      also have img-list-card-eq-inv-img-list:
        . . . = ?q ((sorted-list-of-set ?img)!?c)
        using in-bnds to-list-simp in-bnds img-card-eq-V-length card-in-bnds
        by (metis (no-types, lifting))
      also have img-card-eq-img-list-i:
        . . . = ?q (π (sorted-list-of-set V!i))
        using True elem-of-img
        by (simp add: sorted-list-of-set-nth-equals-card)
      finally show ?thesis
        using assms bij-betw-imp-inj-on the-inv-f-f
              img-list-map img-card-eq-img-list-i
              img-list-card-eq-inv-img-list
        by metis
    qed
    also have to-list V p!i = p ((sorted-list-of-set V)!i)
      using True in-bnds
      by simp
    finally show to-list V p!i =
        map (λ i. (to-list ?img ?q)!(card {v ∈ ?img. v < π (sorted-list-of-set V!i)}))
          [0 ..< length (to-list ?img ?q)]!i
      using in-bnds eq-length Collect-cong card-eq
      by simp
  qed
qed
```

1.5.5 Preference Counts

The win count for an alternative a with respect to a finite voter set V in a profile p is the amount of ballots from V in p that rank alternative a in first position. If the voter set is infinite, counting is not generally possible.

```
fun win-count :: 'v set ⇒ ('a, 'v) Profile ⇒ 'a ⇒ enat where
  win-count V p a = (if (finite V)
    then card {v ∈ V. above (p v) a = {a}} else ∞)

fun prefer-count :: 'v set ⇒ ('a, 'v) Profile ⇒ 'a ⇒ 'a ⇒ enat where
  prefer-count V p x y = (if (finite V)
      then card {v ∈ V. (let r = (p v) in (y ≼_r x))} else ∞)

lemma pref-count-voter-set-card:
  fixes
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a b :: 'a
  assumes finite V
  shows prefer-count V p a b ≤ card V
  using assms
  by (simp add: card-mono)

lemma set-compr:
  fixes
    A :: 'a set and
    f :: 'a ⇒ 'a set
  shows {f x | x. x ∈ A} = f ` A
  by blast

lemma pref-count-set-compr:
  fixes
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  shows {prefer-count V p a a' | a'. a' ∈ A − {a}} =
          (prefer-count V p a) ` (A − {a})
  by blast

lemma pref-count:
  fixes
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a b :: 'a
  assumes
    prof: profile V A p and
    fin: finite V and
    a-in-A: a ∈ A and
    b-in-A: b ∈ A and
    neq: a ≠ b
  shows prefer-count V p a b = card V − (prefer-count V p b a)
proof −
  have ∀ v ∈ V. let r = (p v) in (¬ b ≼_r a ⟶ a ≼_r b)
    using a-in-A b-in-A prof lin-ord-imp-connex
    unfolding profile-def connex-def
    by metis
  moreover have ∀ v ∈ V. (b, a) ∈ (p v) ⟶ (a, b) ∉ (p v)
    using antisymD neq lin-imp-antisym prof
    unfolding profile-def
    by metis
  ultimately have
    {v ∈ V. (let r = (p v) in (b ≼_r a))} =
        V − {v ∈ V. (let r = (p v) in (a ≼_r b))}
    by auto
  thus ?thesis
    by (simp add: card-Diff-subset Collect-mono fin)
qed

lemma pref-count-sym:
  fixes
    p :: ('a, 'v) Profile and
    V :: 'v set and
    a b c :: 'a
  assumes
    pref-count-ineq: prefer-count V p a c ≥ prefer-count V p c b and
    prof: profile V A p and
    a-in-A: a ∈ A and
    b-in-A: b ∈ A and
    c-in-A: c ∈ A and
    a-neq-c: a ≠ c and
    c-neq-b: c ≠ b
  shows prefer-count V p b c ≥ prefer-count V p c a
proof (cases finite V)
  case True
  moreover have
    prefer-count V p c a ∈ ℕ and
    prefer-count V p b c ∈ ℕ
    unfolding Nats-def
    using True of-nat-eq-enat
    by (simp, simp)
  moreover have prefer-count V p c a ≤ card V
    using True prof pref-count-voter-set-card
    by metis
  moreover have
    prefer-count V p a c = card V − (prefer-count V p c a) and
    prefer-count V p c b = card V − (prefer-count V p b c)
    using True pref-count prof c-in-A
    by (metis (no-types, opaque-lifting) a-in-A a-neq-c,
        metis (no-types, opaque-lifting) b-in-A c-neq-b)
  hence card V − (prefer-count V p b c) + (prefer-count V p c a)
      ≤ card V − (prefer-count V p c a) + (prefer-count V p c a)
    using pref-count-ineq
    by simp
  ultimately show ?thesis
    by simp
next
  case False
  thus ?thesis
    by simp
qed

lemma empty-prof-imp-zero-pref-count:
  fixes
    p :: ('a, 'v) Profile and
    V :: 'v set and
    a b :: 'a
  assumes V = {}
  shows prefer-count V p a b = 0
  unfolding zero-enat-def
  using assms
  by simp

fun wins :: 'v set ⇒ 'a ⇒ ('a, 'v) Profile ⇒ 'a ⇒ bool where
  wins V a p b =
    (prefer-count V p a b > prefer-count V p b a)

lemma wins-inf-voters:
  fixes
    p :: ('a, 'v) Profile and
    a b :: 'a and
    V :: 'v set
  assumes infinite V
  shows ¬ wins V b p a
  using assms
  by simp
```

Having alternative a win against b implies that b does not win against a.

```
lemma wins-antisym:
  fixes
    p :: ('a, 'v) Profile and
    a b :: 'a and
    V :: 'v set
  assumes wins V a p b — This already implies that V is finite.
  shows ¬ wins V b p a
  using assms
  by simp

lemma wins-irreflex:
  fixes
    p :: ('a, 'v) Profile and
    a :: 'a and
    V :: 'v set
  shows ¬ wins V a p a
  using wins-antisym
  by metis
```

1.5.6 Condorcet Winner

```
fun condorcet-winner :: 'v set ⇒ 'a set ⇒ ('a, 'v) Profile ⇒ 'a ⇒ bool where
  condorcet-winner V A p a =
     (finite-profile V A p ∧ a ∈ A ∧ (∀ x ∈ A − {a}. wins V a p x))


lemma cond-winner-unique-eq:
  fixes
    V :: 'v set and
    A :: 'a set and
    p :: ('a, 'v) Profile and
    a b :: 'a
  assumes
    condorcet-winner V A p a and
    condorcet-winner V A p b
  shows b = a
proof (rule ccontr)
  assume b-neq-a: b ≠ a
  hence wins V b p a
    using insert-Diff insert-iff assms
    by simp
  hence ¬ wins V a p b
    by (simp add: wins-antisym)
  moreover have wins V a p b
    using Diff-iff b-neq-a singletonD assms
    by auto
  ultimately show False
    by simp
qed

lemma cond-winner-unique:
  fixes
    A :: 'a set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assumes condorcet-winner V A p a
  shows {a' ∈ A. condorcet-winner V A p a'} = {a}
proof (safe)
  fix a' :: 'a
  assume condorcet-winner V A p a'
  thus a' = a
    using assms cond-winner-unique-eq
    by metis
next
  show a ∈ A
    using assms
    unfolding condorcet-winner.simps
    by (metis (no-types))
next
  show condorcet-winner V A p a
    using assms
    by presburger
qed

lemma cond-winner-unique':
  fixes
    V :: 'v set and
    A :: 'a set and
    p :: ('a, 'v) Profile and
    a b :: 'a
  assumes
```

```
    condorcet-winner V A p a and
    b ≠ a
  shows ¬ condorcet-winner V A p b
  using cond-winner-unique-eq assms
  by metis
```

1.5.7 Limited Profile

This function restricts a profile p to a set A of alternatives and a set V of voters s.t. voters outside of V do not have any preferences or do not cast a vote. This keeps all of A's preferences.

```
fun limit-profile :: 'a set ⇒ ('a, 'v) Profile ⇒ ('a, 'v) Profile where
  limit-profile A p = (λ v. limit A (p v))

lemma limit-prof-trans:
  fixes
    A B C :: 'a set and
    p :: ('a, 'v) Profile
  assumes
    B ⊆ A and
    C ⊆ B
  shows limit-profile C p = limit-profile C (limit-profile B p)
  using assms
  by auto

lemma limit-profile-sound:
  fixes
    A B :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assumes
    profile V B p and
    A ⊆ B
  shows profile V A (limit-profile A p)
proof (unfold profile-def)
  have ∀ v ∈ V. linear-order-on A (limit A (p v))
    using assms limit-presv-lin-ord
    unfolding profile-def
    by metis
  thus ∀ v ∈ V. linear-order-on A ((limit-profile A p) v)
    by simp
qed
```

1.5.8 Lifting Property

```
definition equiv-prof-except-a :: 'v set ⇒ 'a set ⇒ ('a, 'v) Profile ⇒
        ('a, 'v) Profile ⇒ 'a ⇒ bool where
  equiv-prof-except-a V A p p' a ≡
    profile V A p ∧ profile V A p' ∧ a ∈ A ∧
    (∀ v ∈ V. equiv-rel-except-a A (p v) (p′ v) a)
```

An alternative gets lifted from one profile to another iff its ranking increases in at least one ballot, and nothing else changes.

```
definition lifted :: ′v set ⇒ ′a set ⇒ (′a, ′v) Profile ⇒
        (′a, ′v) Profile ⇒ ′a ⇒ bool where
  lifted V A p p′ a ≡
    finite-profile V A p ∧ finite-profile V A p′ ∧ a ∈ A
      ∧ (∀ v ∈ V. ¬ Preference-Relation.lifted A (p v) (p′ v) a ⟶ (p v) = (p′ v))
      ∧ (∃ v ∈ V. Preference-Relation.lifted A (p v) (p′ v) a)

lemma lifted-imp-equiv-prof-except-a:
  fixes
    A :: ′a set and
    V :: ′v set and
    p p′ :: (′a, ′v) Profile and
    a :: ′a
  assumes lifted V A p p′ a
  shows equiv-prof-except-a V A p p′ a
proof (unfold equiv-prof-except-a-def, safe)
  show
    profile V A p and
    profile V A p′ and
    a ∈ A
    using assms
    unfolding lifted-def
    by (metis, metis, metis)
next
  fix v :: ′v
  assume v ∈ V
  thus equiv-rel-except-a A (p v) (p′ v) a
    using assms lifted-imp-equiv-rel-except-a trivial-equiv-rel
    unfolding lifted-def profile-def
    by (metis (no-types))
qed

lemma negl-diff-imp-eq-limit-prof:
  fixes
    A A′ :: ′a set and
    V :: ′v set and
    p p′ :: (′a, ′v) Profile and
    a :: ′a
  assumes
    change: equiv-prof-except-a V A′ p q a and
    subset: A ⊆ A′ and
    not-in-A: a ∉ A
  shows ∀ v ∈ V. (limit-profile A p) v = (limit-profile A q) v
```

— With the current definitions of *equiv-prof-except-a* and *limit-prof*, we can only conclude that the limited profiles coincide on the given voter set, since *limit-prof* may change the profiles everywhere, while *equiv-prof-except-a* only makes statements about the voter set.

```
proof (clarify)
  fix v :: 'v
  assume v ∈ V
  hence equiv-rel-except-a A' (p v) (q v) a
    using change equiv-prof-except-a-def
    by metis
  thus limit-profile A p v = limit-profile A q v
    using subset not-in-A negl-diff-imp-eq-limit
    by simp
qed

lemma limit-prof-eq-or-lifted:
  fixes
    A A' :: 'a set and
    V :: 'v set and
    p p' :: ('a, 'v) Profile and
    a :: 'a
  assumes
    lifted-a: lifted V A' p p' a and
    subset: A ⊆ A'
  shows (∀ v ∈ V. limit-profile A p v = limit-profile A p' v)
        ∨ lifted V A (limit-profile A p) (limit-profile A p') a
proof (cases a ∈ A)
  case True
  have ∀ v ∈ V. Preference-Relation.lifted A' (p v) (p' v) a ∨ (p v) = (p' v)
    using lifted-a
    unfolding lifted-def
    by metis
  hence one:
    ∀ v ∈ V.
         Preference-Relation.lifted A (limit A (p v)) (limit A (p' v)) a ∨
           (limit A (p v)) = (limit A (p' v))
    using limit-lifted-imp-eq-or-lifted subset
    by metis
  thus ?thesis
  proof (cases ∀ v ∈ V. limit A (p v) = limit A (p' v))
    case True
    thus ?thesis
      by simp
  next
    case False
    let ?p = limit-profile A p
    let ?q = limit-profile A p'
    have
      profile V A ?p and
      profile V A ?q
      using lifted-a subset limit-profile-sound
      unfolding lifted-def
      by (safe, safe)
    moreover have
      ∃ v ∈ V. Preference-Relation.lifted A (?p v) (?q v) a
      using False one
      unfolding limit-profile.simps
      by (metis (no-types, lifting))
    ultimately have lifted V A ?p ?q a
      using True lifted-a one rev-finite-subset subset
      unfolding lifted-def limit-profile.simps
      by (metis (no-types, lifting))
    thus ?thesis
      by simp
  qed
next
  case False
  thus ?thesis
    using lifted-a negl-diff-imp-eq-limit-prof subset lifted-imp-equiv-prof-except-a
    by metis
qed

end
```

1.6 Social Choice Result

```
theory Social-Choice-Result
  imports Result
begin
```

1.6.1 Definition

A social choice result contains three sets of alternatives: elected, rejected, and deferred alternatives.

```
fun well-formed-𝒮𝒞ℱ :: 'a set ⇒ 'a Result ⇒ bool where
  well-formed-𝒮𝒞ℱ A res = (disjoint3 res ∧ set-equals-partition A res)

fun limit-𝒮𝒞ℱ :: 'a set ⇒ 'a set ⇒ 'a set where
  limit-𝒮𝒞ℱ A r = A ∩ r
```

1.6.2 Auxiliary Lemmas

```
lemma result-imp-rej:
  fixes A e r d :: 'a set
  assumes well-formed-𝒮𝒞ℱ A (e, r, d)
  shows A − (e ∪ d) = r
proof (safe)
  fix a :: 'a
  assume
    a ∈ A and
    a ∉ r and
    a ∉ d
  moreover have
    (e ∩ r = {}) ∧ (e ∩ d = {}) ∧ (r ∩ d = {}) ∧ (e ∪ r ∪ d = A)
    using assms
    by simp
  ultimately show a ∈ e
    by blast
next
  fix a :: 'a
  assume a ∈ r
  moreover have
    (e ∩ r = {}) ∧ (e ∩ d = {}) ∧ (r ∩ d = {}) ∧ (e ∪ r ∪ d = A)
    using assms
    by simp
  ultimately show a ∈ A
    by blast
next
  fix a :: 'a
  assume
    a ∈ r and
    a ∈ e
  moreover have
    (e ∩ r = {}) ∧ (e ∩ d = {}) ∧ (r ∩ d = {}) ∧ (e ∪ r ∪ d = A)
    using assms
    by simp
  ultimately show False
    by auto
next
  fix a :: 'a
  assume
    a ∈ r and
    a ∈ d
  moreover have
    (e ∩ r = {}) ∧ (e ∩ d = {}) ∧ (r ∩ d = {}) ∧ (e ∪ r ∪ d = A)
    using assms
    by simp
  ultimately show False
    by blast
qed

lemma result-count:
  fixes A e r d :: 'a set
  assumes
```

```
    wf-result: well-formed-𝒮𝒞ℱ A (e, r, d) and
    fin-A: finite A
  shows card A = card e + card r + card d
proof −
  have e ∪ r ∪ d = A
    using wf-result
    by simp
  moreover have (e ∩ r = {}) ∧ (e ∩ d = {}) ∧ (r ∩ d = {})
    using wf-result
    by simp
  ultimately show ?thesis
    using fin-A Int-Un-distrib2 finite-Un card-Un-disjoint sup-bot.right-neutral
    by metis
qed

lemma defer-subset:
  fixes
    A :: 'a set and
    r :: 'a Result
  assumes well-formed-𝒮𝒞ℱ A r
  shows defer-r r ⊆ A
proof (safe)
  fix a :: 'a
  assume a ∈ defer-r r
  moreover obtain
    f :: 'a Result ⇒ 'a set ⇒ 'a set and
    g :: 'a Result ⇒ 'a set ⇒ 'a Result where
    A = f r A ∧ r = g r A ∧ disjoint3 (g r A) ∧ set-equals-partition (f r A) (g r A)
    using assms
    by simp
  moreover have
    ∀ p. ∃ e r d. set-equals-partition A p ⟶ (e, r, d) = p ∧ e ∪ r ∪ d = A
    by simp
  ultimately show a ∈ A
    using UnCI snd-conv
    by metis
qed

lemma elect-subset:
  fixes
    A :: 'a set and
    r :: 'a Result
  assumes well-formed-𝒮𝒞ℱ A r
  shows elect-r r ⊆ A
proof (safe)
  fix a :: 'a
  assume a ∈ elect-r r
  moreover obtain
    f :: 'a Result ⇒ 'a set ⇒ 'a set and
    g :: 'a Result ⇒ 'a set ⇒ 'a Result where
    A = f r A ∧ r = g r A ∧ disjoint3 (g r A) ∧ set-equals-partition (f r A) (g r A)
    using assms
    by simp
  moreover have
    ∀ p. ∃ e r d. set-equals-partition A p ⟶ (e, r, d) = p ∧ e ∪ r ∪ d = A
    by simp
  ultimately show a ∈ A
    using UnCI assms fst-conv
    by metis
qed

lemma reject-subset:
  fixes
    A :: 'a set and
    r :: 'a Result
  assumes well-formed-𝒮𝒞ℱ A r
  shows reject-r r ⊆ A
proof (safe)
  fix a :: 'a
  assume a ∈ reject-r r
  moreover obtain
    f :: 'a Result ⇒ 'a set ⇒ 'a set and
    g :: 'a Result ⇒ 'a set ⇒ 'a Result where
    A = f r A ∧ r = g r A ∧ disjoint3 (g r A) ∧ set-equals-partition (f r A) (g r A)
    using assms
    by simp
  moreover have
    ∀ p. ∃ e r d. set-equals-partition A p ⟶ (e, r, d) = p ∧ e ∪ r ∪ d = A
    by simp
  ultimately show a ∈ A
    using UnCI assms fst-conv snd-conv disjoint3.cases
    by metis
qed

end
```

1.7 Social Welfare Result

```
theory Social-Welfare-Result
  imports Result
          Preference-Relation
begin
```

A social welfare result contains three sets of relations: elected, rejected, and deferred. A well-formed social welfare result consists only of linear orders on the alternatives.

```
fun well-formed-SWF :: 'a set ⇒ ('a Preference-Relation) Result ⇒ bool where
  well-formed-SWF A res = (disjoint3 res ∧
                               set-equals-partition {r. linear-order-on A r} res)

fun limit-SWF :: 'a set ⇒ ('a Preference-Relation) set ⇒
        ('a Preference-Relation) set where
  limit-SWF A res = {limit A r | r. r ∈ res ∧ linear-order-on A (limit A r)}

end
```

1.8 Electoral Result Types

```
theory Result-Interpretations
  imports Social-Choice-Result
          Social-Welfare-Result
          Collections.Locale-Code
begin
```

Interpretations of the result locale are placed inside a Locale-Code block in order to enable code generation of later definitions in the locale. Those definitions need to be added via a Locale-Code block as well.

```
setup Locale-Code.open-block
```

Results from social choice functions ($\mathcal{SCF}s$), for the purpose of composability and modularity given as three sets of (potentially tied) alternatives. See `Social_Choice_Result.thy` for details.

```
global-interpretation SCF-result: result well-formed-SCF limit-SCF
proof (unfold-locales, safe)
  fix A e r d :: 'a set
  assume
    set-equals-partition (limit-SCF A UNIV) (e, r, d) and
    disjoint3 (e, r, d)
  thus well-formed-SCF A (e, r, d)
    by simp
qed
```

Results from committee functions, for the purpose of composability and modularity given as three sets of (potentially tied) sets of alternatives or committees. [[*Not actually used yet.*]]

```
global-interpretation committee-result: result
        λ A r. set-equals-partition (Pow A) r ∧ disjoint3 r
        λ A rs. {r ∩ A | r. r ∈ rs}
proof (unfold-locales, safe)
  fix
    A :: 'b set and
    e r d :: 'b set set
  assume set-equals-partition {r ∩ A |r. r ∈ UNIV} (e, r, d)
  thus set-equals-partition (Pow A) (e, r, d)
    by force
qed
```

Results from social welfare functions (𝒮𝒲ℱ*s*), for the purpose of composability and modularity given as three sets of (potentially tied) linear orders over the alternatives. See `Social_Welfare_Result.thy` for details.

```
global-interpretation 𝒮𝒲ℱ-result: result well-formed-𝒮𝒲ℱ limit-𝒮𝒲ℱ
proof (unfold-locales, safe)
  fix
    A :: 'a set and
    e r d :: ('a Preference-Relation) set
  assume
    set-equals-partition (limit-𝒮𝒲ℱ A UNIV) (e, r, d) and
    disjoint3 (e, r, d)
  moreover have
    limit-𝒮𝒲ℱ A UNIV = {limit A r' | r'. linear-order-on A (limit A r')}
    by simp
  moreover have ... = {r'. linear-order-on A r'}
  proof (safe)
    fix r' :: 'a Preference-Relation
    assume lin-ord: linear-order-on A r'
    hence ∀ (a, b) ∈ r'. (a, b) ∈ limit A r'
      unfolding linear-order-on-def partial-order-on-def preorder-on-def refl-on-def
      by force
    hence r' = limit A r'
      by force
    thus ∃ x. r' = limit A x ∧ linear-order-on A (limit A x)
      using lin-ord
      by metis
  qed
  ultimately show well-formed-𝒮𝒲ℱ A (e, r, d)
    by simp
qed

setup Locale-Code.close-block

end
```

1.9 Symmetry Properties of Functions

```
theory Symmetry-Of-Functions
  imports HOL−Algebra.Group-Action
          HOL−Algebra.Generated-Groups
```

begin

1.9.1 Functions

type-synonym $('x,\ 'y)\ \mathit{binary\text{-}fun} = 'x \Rightarrow 'y \Rightarrow 'y$

fun $\mathit{extensional\text{-}continuation} :: ('x \Rightarrow 'y) \Rightarrow 'x\ \mathit{set} \Rightarrow ('x \Rightarrow 'y)$ **where**
$\mathit{extensional\text{-}continuation}\ f\ S = (\lambda\ x.\ \mathit{if}\ (x \in S)\ \mathit{then}\ (f\ x)\ \mathit{else}\ \mathit{undefined})$

fun $\mathit{preimg} :: ('x \Rightarrow 'y) \Rightarrow 'x\ \mathit{set} \Rightarrow 'y \Rightarrow 'x\ \mathit{set}$ **where**
$\mathit{preimg}\ f\ S\ y = \{x \in S.\ f\ x = y\}$

1.9.2 Relations for Symmetry Constructions

fun $\mathit{restricted\text{-}rel} :: 'x\ \mathit{rel} \Rightarrow 'x\ \mathit{set} \Rightarrow 'x\ \mathit{set} \Rightarrow 'x\ \mathit{rel}$ **where**
$\mathit{restricted\text{-}rel}\ r\ S\ S' = r \cap (S \times S')$

fun $\mathit{closed\text{-}restricted\text{-}rel} :: 'x\ \mathit{rel} \Rightarrow 'x\ \mathit{set} \Rightarrow 'x\ \mathit{set} \Rightarrow \mathit{bool}$ **where**
$\mathit{closed\text{-}restricted\text{-}rel}\ r\ S\ T = ((\mathit{restricted\text{-}rel}\ r\ T\ S)\ ``\ T \subseteq T)$

fun $\mathit{action\text{-}induced\text{-}rel} :: 'x\ \mathit{set} \Rightarrow 'y\ \mathit{set} \Rightarrow ('x,\ 'y)\ \mathit{binary\text{-}fun} \Rightarrow 'y\ \mathit{rel}$ **where**
$\mathit{action\text{-}induced\text{-}rel}\ S\ T\ \varphi = \{(y,\ y').\ y \in T \wedge (\exists\ x \in S.\ \varphi\ x\ y = y')\}$

fun $\mathit{product} :: 'x\ \mathit{rel} \Rightarrow ('x * 'x)\ \mathit{rel}$ **where**
$\mathit{product}\ r = \{(p,\ p').\ (\mathit{fst}\ p,\ \mathit{fst}\ p') \in r \wedge (\mathit{snd}\ p,\ \mathit{snd}\ p') \in r\}$

fun $\mathit{equivariance} :: 'x\ \mathit{set} \Rightarrow 'y\ \mathit{set} \Rightarrow ('x,\ 'y)\ \mathit{binary\text{-}fun} \Rightarrow ('y * 'y)\ \mathit{rel}$ **where**
$\mathit{equivariance}\ S\ T\ \varphi =$
$\{((u,\ v),\ (x,\ y)).\ (u,\ v) \in T \times T \wedge (\exists\ z \in S.\ x = \varphi\ z\ u \wedge y = \varphi\ z\ v)\}$

fun $\mathit{singleton\text{-}set\text{-}system} :: 'x\ \mathit{set} \Rightarrow 'x\ \mathit{set}\ \mathit{set}$ **where**
$\mathit{singleton\text{-}set\text{-}system}\ S = \{\{x\} \mid x.\ x \in S\}$

fun $\mathit{set\text{-}action} :: ('x,\ 'r)\ \mathit{binary\text{-}fun} \Rightarrow ('x,\ 'r\ \mathit{set})\ \mathit{binary\text{-}fun}$ **where**
$\mathit{set\text{-}action}\ \psi\ x = \mathit{image}\ (\psi\ x)$

1.9.3 Invariance and Equivariance

Invariance and equivariance are symmetry properties of functions: Invariance means that related preimages have identical images and equivariance denotes consistent changes.

datatype $('x,\ 'y)\ \mathit{symmetry} =$
$\mathit{Invariance}\ 'x\ \mathit{rel}\ |$
$\mathit{Equivariance}\ 'x\ \mathit{set}\ (('x \Rightarrow 'x) \times ('y \Rightarrow 'y))\ \mathit{set}$

fun $\mathit{is\text{-}symmetry} :: ('x \Rightarrow 'y) \Rightarrow ('x,\ 'y)\ \mathit{symmetry} \Rightarrow \mathit{bool}$ **where**
$\mathit{is\text{-}symmetry}\ f\ (\mathit{Invariance}\ r) = (\forall\ x.\ \forall\ y.\ (x,\ y) \in r \longrightarrow f\ x = f\ y)\ |$
$\mathit{is\text{-}symmetry}\ f\ (\mathit{Equivariance}\ s\ \tau) = (\forall\ (\varphi,\ \psi) \in \tau.\ \forall\ x \in s.\ f\ (\varphi\ x) = \psi\ (f\ x))$

```
definition action-induced-equivariance :: 'z set ⇒ 'x set ⇒ ('z, 'x) binary-fun ⇒
    ('z, 'y) binary-fun ⇒ ('x, 'y) symmetry where
  action-induced-equivariance T S φ ψ ≡ Equivariance S {(φ z, ψ z) | z. z ∈ T}
```

1.9.4 Auxiliary Lemmas

```
lemma un-left-inv-singleton-set-system: ⋃ ∘ singleton-set-system = id
proof
  fix s :: 'x set
  have (⋃ ∘ singleton-set-system) s = {x. {x} ∈ singleton-set-system s}
    by force
  thus (⋃ ∘ singleton-set-system) s = id s
    by simp
qed

lemma preimg-comp:
  fixes
    f :: 'x ⇒ 'y and
    g :: 'x ⇒ 'x and
    S :: 'x set and
    x :: 'y
  shows preimg f (g ‘ S) x = g ‘ preimg (f ∘ g) S x
proof (safe)
  fix y :: 'x
  assume y ∈ preimg f (g ‘ S) x
  then obtain z :: 'x where
    g z = y and
    z ∈ preimg (f ∘ g) S x
    unfolding comp-def
    by fastforce
  thus y ∈ g ‘ preimg (f ∘ g) S x
    by blast
next
  fix y :: 'x
  assume y ∈ preimg (f ∘ g) S x
  thus g y ∈ preimg f (g ‘ S) x
    by simp
qed
```

1.9.5 Rewrite Rules

```
theorem rewrite-invar-as-equivar:
  fixes
    f :: 'x ⇒ 'y and
    S :: 'x set and
    T :: 'z set and
    φ :: ('z, 'x) binary-fun
  shows is-symmetry f (Invariance (action-induced-rel T S φ)) =
          is-symmetry f (action-induced-equivariance T S φ (λ g. id))
proof (unfold action-induced-equivariance-def is-symmetry.simps, safe)
```

```
  fix
    x :: 'x and
    g :: 'z
  assume
    x ∈ S and
    g ∈ T and
    ∀ x y. (x, y) ∈ action-induced-rel T S φ ⟶ f x = f y
  moreover with this have (x, φ g x) ∈ action-induced-rel T S φ
    unfolding action-induced-rel.simps
    by blast
  ultimately show f (φ g x) = id (f x)
    by simp
next
  fix x y :: 'x
  assume
    equivar:
      ∀ (φ, ψ) ∈ {(φ g, id) |g. g ∈ T}. ∀ x ∈ S. f (φ x) = ψ (f x) and
    rel: (x, y) ∈ action-induced-rel T S φ
  then obtain g :: 'z where
    img: φ g x = y and
    elt: g ∈ T
    unfolding action-induced-rel.simps
    by blast
  moreover have x ∈ S
    using rel
    by simp
  ultimately have f (φ g x) = id (f x)
    using equivar elt
    by blast
  thus f x = f y
    using img elt
    by simp
qed

lemma rewrite-invar-ind-by-act:
  fixes
    f :: 'x ⇒ 'y and
    S :: 'z set and
    T :: 'x set and
    φ :: ('z, 'x) binary-fun
  shows is-symmetry f (Invariance (action-induced-rel S T φ)) =
            (∀ x ∈ S. ∀ y ∈ T. f y = f (φ x y))
proof (safe)
  fix
    y :: 'x and
    x :: 'z
  assume
    is-symmetry f (Invariance (action-induced-rel S T φ)) and
    y ∈ T and
    x ∈ S
  moreover from this have (y, φ x y) ∈ action-induced-rel S T φ
    unfolding action-induced-rel.simps
    by blast
  ultimately show f y = f (φ x y)
    by simp
next
  assume ∀ x ∈ S. ∀ y ∈ T. f y = f (φ x y)
  moreover have
    ∀ (x, y) ∈ action-induced-rel S T φ. x ∈ T ∧ (∃ z ∈ S. y = φ z x)
    by auto
  ultimately show is-symmetry f (Invariance (action-induced-rel S T φ))
    by auto
qed

lemma rewrite-equivariance:
  fixes
    f :: 'x ⇒ 'y and
    S :: 'z set and
    T :: 'x set and
    φ :: ('z, 'x) binary-fun and
    ψ :: ('z, 'y) binary-fun
  shows is-symmetry f (action-induced-equivariance S T φ ψ) =
          (∀ x ∈ S. ∀ y ∈ T. f (φ x y) = ψ x (f y))
  unfolding action-induced-equivariance-def
  by auto

lemma rewrite-group-action-img:
  fixes
    m :: 'x monoid and
    S T :: 'y set and
    φ :: ('x, 'y) binary-fun and
    x y :: 'x
  assumes
    T ⊆ S and
    x ∈ carrier m and
    y ∈ carrier m and
    group-action m S φ
  shows φ (x ⊗ m y) ‘ T = φ x ‘ φ y ‘ T
proof (safe)
  fix z :: 'y
  assume z-in-t: z ∈ T
  hence φ (x ⊗ m y) z = φ x (φ y z)
    using assms group-action.composition-rule[of m S]
    by blast
  thus
    φ (x ⊗ m y) z ∈ φ x ‘ φ y ‘ T and
    φ x (φ y z) ∈ φ (x ⊗m y) ‘ T
    using z-in-t
    by (blast, force)
qed

lemma rewrite-carrier: carrier (BijGroup UNIV) = {f. bij f}
  unfolding BijGroup-def Bij-def
  by simp

lemma universal-set-carrier-imp-bij-group:
  fixes f :: 'a ⇒ 'a
  assumes f ∈ carrier (BijGroup UNIV)
  shows bij f
  using rewrite-carrier assms
  by blast

lemma rewrite-sym-group:
  fixes
    f g :: 'a ⇒ 'a and
    S :: 'a set
  assumes
    f ∈ carrier (BijGroup S) and
    g ∈ carrier (BijGroup S)
  shows
    rewrite-mult: f ⊗ BijGroup S g = extensional-continuation (f ∘ g) S and
    rewrite-mult-univ: S = UNIV ⟶ f ⊗ BijGroup S g = f ∘ g
  using assms
  unfolding BijGroup-def compose-def comp-def restrict-def
  by (simp, fastforce)

lemma simp-extensional-univ:
  fixes f :: 'a ⇒ 'b
  shows extensional-continuation f UNIV = f
  unfolding If-def
  by simp

lemma extensional-continuation-subset:
  fixes
    f :: 'a ⇒ 'b and
    S T :: 'a set and
    x :: 'a
  assumes
    T ⊆ S and
    x ∈ T
  shows extensional-continuation f S x = extensional-continuation f T x
  using assms
  unfolding subset-iff
  by simp

lemma rel-ind-by-coinciding-action-on-subset-eq-restr:
  fixes
```

```
    φ ψ :: ('a, 'b) binary-fun and
    S :: 'a set and
    T U :: 'b set
  assumes
    U ⊆ T and
    ∀ x ∈ S. ∀ y ∈ U. ψ x y = φ x y
  shows action-induced-rel S U ψ = restricted-rel (action-induced-rel S T φ) U
UNIV
proof (unfold action-induced-rel.simps restricted-rel.simps, safe)
  fix x :: 'b
  assume x ∈ U
  thus x ∈ T
    using assms
    by blast
next
  fix
    g :: 'a and
    x :: 'b
  assume
    g-in-S: g ∈ S and
    x-in-U: x ∈ U
  hence φ g x = ψ g x
    using assms
    by simp
  thus ∃ g' ∈ S. φ g' x = ψ g x
    using g-in-S
    by blast
next
  fix
    g :: 'a and
    x :: 'b
  show ψ g x ∈ UNIV
    by blast
next
  fix
    g :: 'a and
    x :: 'b
  assume
    g-in-S: g ∈ S and
    x-in-U: x ∈ U
  hence ψ g x = φ g x
    using assms
    by simp
  thus ∃ g' ∈ S. ψ g' x = φ g x
    using g-in-S
    by blast
qed

lemma coinciding-actions-ind-equal-rel:
  fixes
    S :: 'x set and
    T :: 'y set and
    φ ψ :: ('x, 'y) binary-fun
  assumes ∀ x ∈ S. ∀ y ∈ T. φ x y = ψ x y
  shows action-induced-rel S T φ = action-induced-rel S T ψ
  unfolding extensional-continuation.simps
  using assms
  by auto
```

1.9.6 Group Actions

```
lemma const-id-is-group-action:
  fixes m :: 'x monoid
  assumes group m
  shows group-action m UNIV (λ x. id)
  using assms
proof (unfold group-action-def group-hom-def group-hom-axioms-def hom-def, safe)
  show group (BijGroup UNIV)
    using group-BijGroup
    by metis
next
  show id ∈ carrier (BijGroup UNIV)
    unfolding BijGroup-def Bij-def
    by simp
  thus id = id ⊗ BijGroup UNIV id
    using rewrite-mult-univ comp-id
    by metis
qed

theorem group-act-induces-set-group-act:
  fixes
    m :: 'x monoid and
    S :: 'y set and
    φ :: ('x, 'y) binary-fun
  defines φ-img ≡ (λ x. extensional-continuation (image (φ x)) (Pow S))
  assumes group-action m S φ
  shows group-action m (Pow S) φ-img
proof (unfold group-action-def group-hom-def group-hom-axioms-def hom-def, safe)
  show group m
    using assms
    unfolding group-action-def group-hom-def
    by simp
next
  show group (BijGroup (Pow S))
    using group-BijGroup
    by metis
next
  {
```

```
  fix x :: 'x
  assume x ∈ carrier m
  hence bij-betw (φ x) S S
    using assms group-action.surj-prop
    unfolding bij-betw-def
    by (simp add: group-action.inj-prop)
  hence bij-betw (image (φ x)) (Pow S) (Pow S)
    using bij-betw-Pow
    by metis
  moreover have ∀ t ∈ Pow S. φ-img x t = image (φ x) t
    unfolding φ-img-def
    by simp
  ultimately have bij-betw (φ-img x) (Pow S) (Pow S)
    using bij-betw-cong
    by fastforce
  moreover have φ-img x ∈ extensional (Pow S)
    unfolding φ-img-def extensional-def
    by simp
  ultimately show φ-img x ∈ carrier (BijGroup (Pow S))
    unfolding BijGroup-def Bij-def
    by simp
}
fix x y :: 'x
note
  ‹x ∈ carrier m ⟹ φ-img x ∈ carrier (BijGroup (Pow S))› and
  ‹y ∈ carrier m ⟹ φ-img y ∈ carrier (BijGroup (Pow S))›
moreover assume
  carrier-x: x ∈ carrier m and
  carrier-y: y ∈ carrier m
ultimately have
  carrier-election-x: φ-img x ∈ carrier (BijGroup (Pow S)) and
  carrier-election-y: φ-img y ∈ carrier (BijGroup (Pow S))
  by (presburger, presburger)
hence h-closed: ∀ T ∈ Pow S. φ-img y T ∈ Pow S
  using bij-betw-apply Int-Collect
  unfolding BijGroup-def Bij-def partial-object.select-convs
  by (metis (no-types))
from carrier-election-x carrier-election-y
have φ-img x ⊗ BijGroup (Pow S) φ-img y =
      extensional-continuation (φ-img x ∘ φ-img y) (Pow S)
  using rewrite-mult
  by blast
moreover have
  ∀ T. T ∉ Pow S
    ⟶ extensional-continuation (φ-img x ∘ φ-img y) (Pow S) T = undefined
  by simp
moreover have
  ∀ T. T ∉ Pow S ⟶ φ-img (x ⊗ m y) T = undefined and
  ∀ T ∈ Pow S.
```

```
    extensional-continuation (φ-img x ∘ φ-img y) (Pow S) T = φ x ‘ φ y ‘ T
  using h-closed
  unfolding φ-img-def
  by (simp, simp)
moreover have ∀ T ∈ Pow S. φ-img (x ⊗ m y) T = φ x ‘ φ y ‘ T
  unfolding φ-img-def extensional-continuation.simps
  using rewrite-group-action-img carrier-x carrier-y assms PowD
  by metis
ultimately have
  ∀ T. φ-img (x ⊗ m y) T = (φ-img x ⊗ BijGroup (Pow S) φ-img y) T
  by metis
thus φ-img (x ⊗ m y) = φ-img x ⊗ BijGroup (Pow S) φ-img y
  by blast
qed
```

1.9.7 Invariance and Equivariance

It suffices to show equivariance under the group action of a generating set of a group to show equivariance under the group action of the whole group. For example, it is enough to show invariance under transpositions to show invariance under a complete finite symmetric group.

```
theorem equivar-generators-imp-equivar-group:
  fixes
    f :: 'x ⇒ 'y and
    m :: 'z monoid and
    S :: 'z set and
    T :: 'x set and
    φ :: ('z, 'x) binary-fun and
    ψ :: ('z, 'y) binary-fun
  assumes
    equivar: is-symmetry f (action-induced-equivariance S T φ ψ) and
    action-φ: group-action m T φ and
    action-ψ: group-action m (f ‘ T) ψ and
    gen: carrier m = generate m S
  shows is-symmetry f (action-induced-equivariance (carrier m) T φ ψ)
proof (unfold is-symmetry.simps action-induced-equivariance-def action-induced-rel.simps,
      safe)
  fix
    g :: 'z and
    x :: 'x
  assume
    group-elem: g ∈ carrier m and
    x-in-t: x ∈ T
  have g ∈ generate m S
    using group-elem gen
    by blast
  hence ∀ x ∈ T. f (φ g x) = ψ g (f x)
  proof (induct g rule: generate.induct)
  case one
  hence ∀ x ∈ T. φ 1 m x = x
    using action-φ group-action.id-eq-one restrict-apply
    by metis
  moreover with one have ∀ y ∈ (f ‘ T). ψ 1 m y = y
    using action-ψ  group-action.id-eq-one restrict-apply
    by metis
  ultimately show ?case
    by simp
next
  case (incl g)
  hence ∀ x ∈ T. φ g x ∈ T
    using action-φ gen generate.incl group-action.element-image
    by metis
  thus ?case
    using incl equivar rewrite-equivariance
    unfolding is-symmetry.simps
    by metis
next
  case (inv g)
  hence in-t: ∀ x ∈ T. φ (inv m g) x ∈ T
    using action-φ gen generate.inv group-action.element-image
    by metis
  hence ∀ x ∈ T. f (φ g (φ (inv m g) x)) = ψ g (f (φ (inv m g) x))
    using gen action-φ equivar local.inv rewrite-equivariance
    by metis
  moreover have ∀ x ∈ T. φ g (φ (inv m g) x) = x
    using action-φ gen generate.incl group.inv-closed group-action.orbit-sym-aux
          group.inv-inv group-action.group-hom local.inv
    unfolding group-hom-def
    by (metis (full-types))
  ultimately have ∀ x ∈ T. ψ g (f (φ (inv m g) x)) = f x
    by simp
  moreover have in-img-t: ∀ x ∈ T. f (φ (inv m g) x) ∈ f ‘ T
    using in-t
    by blast
  ultimately have
    ∀ x ∈ T. ψ (inv m g) (ψ g (f (φ (inv m g) x))) = ψ (inv m g) (f x)
    using action-ψ gen
    by metis
  moreover have
    ∀ x ∈ T. ψ (inv m g) (ψ g (f (φ (inv m g) x))) = f (φ (inv m g) x)
   using in-img-t action-ψ gen generate.incl group-action.orbit-sym-aux local.inv
    by metis
  ultimately show ?case
    by simp
next
  case (eng g1 g2)
  assume
```

```
      equivar_1: ∀ x ∈ T. f (φ g_1 x) = ψ g_1 (f x) and
      equivar_2: ∀ x ∈ T. f (φ g_2 x) = ψ g_2 (f x) and
      gen_1: g_1 ∈ generate m S and
      gen_2: g_2 ∈ generate m S
    hence ∀ x ∈ T. φ g_2 x ∈ T
      using gen action-φ group-action.element-image
      by metis
    hence ∀ x ∈ T. f (φ g_1 (φ g_2 x)) = ψ g_1 (f (φ g_2 x))
      using equivar_1
      by simp
    moreover have ∀ x ∈ T. f (φ g_2 x) = ψ g_2 (f x)
      using equivar_2
      by simp
    ultimately show ?case
      using action-φ action-ψ gen gen_1 gen_2 group-action.composition-rule imageI
      by (metis (no-types, lifting))
  qed
  thus f (φ g x) = ψ g (f x)
    using x-in-t
    by simp
qed

lemma invar-parameterized-fun:
  fixes
    f :: 'x ⇒ ('x ⇒ 'y) and
    r :: 'x rel
  assumes
    ∀ x. is-symmetry (f x) (Invariance r) and
    is-symmetry f (Invariance r)
  shows is-symmetry (λ x. f x x) (Invariance r)
  using assms
  by simp

lemma invar-under-subset-rel:
  fixes
    f :: 'x ⇒ 'y and
    r s :: 'x rel
  assumes
    subset: r ⊆ s and
    invar: is-symmetry f (Invariance s)
  shows is-symmetry f (Invariance r)
  using assms
  by auto

lemma equivar-ind-by-act-coincide:
  fixes
    S :: 'x set and
    T :: 'y set and
    f :: 'y ⇒ 'z and
    φ φ' :: ('x, 'y) binary-fun and
    ψ :: ('x, 'z) binary-fun
  assumes ∀ x ∈ S. ∀ y ∈ T. φ x y = φ' x y
  shows is-symmetry f (action-induced-equivariance S T φ ψ) =
          is-symmetry f (action-induced-equivariance S T φ' ψ)
  using assms
  unfolding rewrite-equivariance
  by simp

lemma equivar-under-subset:
  fixes
    f :: 'x ⇒ 'y and
    S T :: 'x set and
    τ :: (('x ⇒ 'x) × ('y ⇒ 'y)) set
  assumes
    is-symmetry f (Equivariance S τ) and
    T ⊆ S
  shows is-symmetry f (Equivariance T τ)
  using assms
  unfolding is-symmetry.simps
  by blast

lemma equivar-under-subset':
  fixes
    f :: 'x ⇒ 'y and
    S :: 'x set and
    τ υ :: (('x ⇒ 'x) × ('y ⇒ 'y)) set
  assumes
    is-symmetry f (Equivariance S τ) and
    υ ⊆ τ
  shows is-symmetry f (Equivariance S υ)
  using assms
  unfolding is-symmetry.simps
  by blast

theorem group-action-equivar-f-imp-equivar-preimg:
  fixes
    f :: 'x ⇒ 'y and
    𝒟_f S :: 'x set and
    m :: 'z monoid and
    φ :: ('z, 'x) binary-fun and
    ψ :: ('z, 'y) binary-fun and
    x :: 'z
  defines equivar-prop ≡ action-induced-equivariance (carrier m) 𝒟_f φ ψ
  assumes
    action-φ: group-action m S φ and
    action-res: group-action m UNIV ψ and
    dom-in-s: 𝒟_f ⊆ S and
    closed-domain:
      closed-restricted-rel (action-induced-rel (carrier m) S φ) S 𝒟_f and
    equivar-f: is-symmetry f equivar-prop and
    group-elem-x: x ∈ carrier m
  shows ∀ y. preimg f 𝒟_f (ψ x y) = (φ x) ‘ (preimg f 𝒟_f y)
proof (safe)
  interpret action-φ: group-action m S φ
    using action-φ
    by simp
  interpret action-results: group-action m UNIV ψ
    using action-res
    by simp
  have group-elem-inv: (inv_m x) ∈ carrier m
    using group.inv-closed action-φ.group-hom group-elem-x
    unfolding group-hom-def
    by metis
  fix
    y :: 'y and
    z :: 'x
  assume preimg-el: z ∈ preimg f 𝒟_f (ψ x y)
  obtain a :: 'x where
    img: a = φ (inv_m x) z
    by simp
  have domain: z ∈ 𝒟_f ∧ z ∈ S
    using preimg-el dom-in-s
    by auto
  hence a ∈ S
    using dom-in-s action-φ group-elem-inv preimg-el img action-φ.element-image
    by auto
  hence (z, a) ∈ (action-induced-rel (carrier m) S φ) ∩ (𝒟_f × S)
    using img preimg-el domain group-elem-inv
    by auto
  hence a ∈ ((action-induced-rel (carrier m) S φ) ∩ (𝒟_f × S)) “ 𝒟_f
    using img preimg-el domain group-elem-inv
    by auto
  hence a-in-domain: a ∈ 𝒟_f
    using closed-domain
    by auto
  moreover have (φ (inv_m x), ψ (inv_m x)) ∈ {(φ g, ψ g) | g. g ∈ carrier m}
    using group-elem-inv
    by auto
  ultimately have f a = ψ (inv_m x) (f z)
    using domain equivar-f img
    unfolding equivar-prop-def action-induced-equivariance-def
    by simp
  hence f a = y
    using preimg-el action-results.group-hom action-results.orbit-sym-aux
        group-elem-x
    by simp
  hence a ∈ preimg f 𝒟_f y
```

```
    using a-in-domain
    by simp
  moreover have z = φ x a
    using action-φ.group-hom action-φ.orbit-sym-aux img domain
          a-in-domain group-elem-x group-elem-inv group.inv-inv
    unfolding group-hom-def
    by metis
  ultimately show z ∈ (φ x) ‘ (preimg f 𝒟_f y)
    by simp
next
  fix
    y :: 'y and
    z :: 'x
  assume z ∈ preimg f 𝒟_f y
  hence domain: f z = y ∧ z ∈ 𝒟_f ∧ z ∈ S
    using dom-in-s
    by auto
  hence φ x z ∈ S
    using group-elem-x group-action.element-image action-φ
    by metis
  hence (z, φ x z) ∈ (action-induced-rel (carrier m) S φ) ∩ (𝒟_f × S) ∩ 𝒟_f × S
    using group-elem-x domain
    by auto
  hence φ x z ∈ 𝒟_f
    using closed-domain
    by auto
  moreover have (φ x, ψ x) ∈ {(φ a, ψ a) | a. a ∈ carrier m}
    using group-elem-x
    by blast
  ultimately show φ x z ∈ preimg f 𝒟_f (ψ x y)
    using equivar-f domain
    unfolding equivar-prop-def action-induced-equivariance-def
    by simp
qed
```

1.9.8 Function Composition

```
lemma invar-comp:
  fixes
    f :: 'x ⇒ 'y and
    g :: 'y ⇒ 'z and
    r :: 'x rel
  assumes is-symmetry f (Invariance r)
  shows is-symmetry (g ∘ f) (Invariance r)
  using assms
  by simp

lemma equivar-comp:
  fixes
```

$f :: 'x \Rightarrow 'y$ **and**
$g :: 'y \Rightarrow 'z$ **and**
$S :: 'x$ *set* **and**
$T :: 'y$ *set* **and**
$\tau :: (('x \Rightarrow 'x) \times ('y \Rightarrow 'y))$ *set* **and**
$\upsilon :: (('y \Rightarrow 'y) \times ('z \Rightarrow 'z))$ *set*
defines
transitive-acts $\equiv$
$\{(\varphi, \psi).\ \exists\ \chi :: 'y \Rightarrow 'y.\ (\varphi, \chi) \in \tau \wedge (\chi, \psi) \in \upsilon \wedge \chi\ `\ f\ `\ S \subseteq T\}$
assumes
$f\ `\ S \subseteq T$ **and**
is-symmetry f (*Equivariance S* τ) **and**
is-symmetry g (*Equivariance T* υ)
shows *is-symmetry* $(g \circ f)$ (*Equivariance S transitive-acts*)
proof (*unfold transitive-acts-def is-symmetry.simps comp-def*, *safe*)
fix
$\varphi :: 'x \Rightarrow 'x$ **and**
$\chi :: 'y \Rightarrow 'y$ **and**
$\psi :: 'z \Rightarrow 'z$ **and**
$x :: 'x$
assume
x-in-X: $x \in S$ **and**
χ-*img*$_f$-*img*$_s$-*in-t*: $\chi\ `\ f\ `\ S \subseteq T$ **and**
act-f: $(\varphi, \chi) \in \tau$ **and**
act-g: $(\chi, \psi) \in \upsilon$
hence $f\,x \in T \wedge \chi\ (f\,x) \in T$
using *assms*
by *blast*
hence $\psi\ (g\ (f\,x)) = g\ (\chi\ (f\,x))$
using *act-g assms*
by *fastforce*
also have $g\ (f\ (\varphi\ x)) = g\ (\chi\ (f\,x))$
using *assms act-f x-in-X*
by *fastforce*
finally show $g\ (f\ (\varphi\ x)) = \psi\ (g\ (f\,x))$
by *simp*
qed

lemma *equivar-ind-by-action-comp*:
fixes
$f :: 'x \Rightarrow 'y$ **and**
$g :: 'y \Rightarrow 'z$ **and**
$S :: 'w$ *set* **and**
$T :: 'x$ *set* **and**
$U :: 'y$ *set* **and**
$\varphi :: ('w, 'x)$ *binary-fun* **and**
$\chi :: ('w, 'y)$ *binary-fun* **and**
$\psi :: ('w, 'z)$ *binary-fun*
assumes

```
    f ‘ T ⊆ U and
    ∀ x ∈ S. χ x ‘ f ‘ T ⊆ U and
    is-symmetry f (action-induced-equivariance S T φ χ) and
    is-symmetry g (action-induced-equivariance S U χ ψ)
  shows is-symmetry (g ∘ f) (action-induced-equivariance S T φ ψ)
proof −
  let ?a_φ = {(φ a, χ a) | a. a ∈ S} and
      ?a_ψ = {(χ a, ψ a) | a. a ∈ S}
  have ∀ a ∈ S. (φ a, χ a) ∈ {(φ a, χ a) | b. b ∈ S}
          ∧ (χ a, ψ a) ∈ {(χ b, ψ b) | b. b ∈ S} ∧ χ a ‘ f ‘ T ⊆ U
    using assms
    by blast
  hence {(φ a, ψ a) | a. a ∈ S}
      ⊆ {(φ, ψ). ∃ υ. (φ, υ) ∈ ?a_φ ∧ (υ, ψ) ∈ ?a_ψ ∧ υ ‘ f ‘ T ⊆ U}
    by blast
  hence is-symmetry (g ∘ f) (Equivariance T {(φ a, ψ a) | a. a ∈ S})
    using assms equivar-comp[of - - - ?a_φ - ?a_ψ] equivar-under-subset′
    unfolding action-induced-equivariance-def
    by (metis (no-types, lifting))
  thus ?thesis
    unfolding action-induced-equivariance-def
    by blast
qed

lemma equivar-set-minus:
  fixes
    f g :: ′x ⇒ ′y set and
    S :: ′z set and
    T :: ′x set and
    φ :: (′z, ′x) binary-fun and
    ψ :: (′z, ′y) binary-fun
  assumes
     f-equivar: is-symmetry f (action-induced-equivariance S T φ (set-action ψ))
and
     g-equivar: is-symmetry g (action-induced-equivariance S T φ (set-action ψ))
and
    bij-a: ∀ a ∈ S. bij (ψ a)
  shows
    is-symmetry (λ b. f b − g b) (action-induced-equivariance S T φ (set-action ψ))
proof −
  have
    ∀ a ∈ S. ∀ x ∈ T. f (φ a x) = ψ a ‘ (f x) and
    ∀ a ∈ S. ∀ x ∈ T. g (φ a x) = ψ a ‘ (g x)
    using f-equivar g-equivar
    unfolding rewrite-equivariance
    by (simp, simp)
  hence ∀ a ∈ S. ∀ b ∈ T. f (φ a b) − g (φ a b) = ψ a ‘ (f b) − ψ a ‘ (g b)
    by blast
  moreover have ∀ a ∈ S. ∀ u v. ψ a ‘ u − ψ a ‘ v = ψ a ‘ (u − v)
```

```
    using bij-a image-set-diff
    unfolding bij-def
    by blast
  ultimately show ?thesis
    unfolding set-action.simps
    using rewrite-equivariance
    by fastforce
qed

lemma equivar-union-under-image-action:
  fixes
    f :: 'x ⇒ 'y and
    S :: 'z set and
    φ :: ('z, 'x) binary-fun
  shows is-symmetry ⋃ (action-induced-equivariance S UNIV
            (set-action (set-action φ)) (set-action φ))
  unfolding action-induced-equivariance-def is-symmetry.simps set-action.simps
  by blast

end
```

1.10 Symmetry Properties of Voting Rules

```
theory Voting-Symmetry
  imports Symmetry-Of-Functions
          Social-Choice-Result
          Social-Welfare-Result
          Profile
begin
```

1.10.1 Definitions

```
fun result-action :: ('x, 'r) binary-fun ⇒ ('x, 'r Result) binary-fun where
  result-action ψ x = (λ r. (ψ x ` elect-r r, ψ x ` reject-r r, ψ x ` defer-r r))
```

Anonymity

Bijection group on the set of voters.

```
definition bijection_𝒱𝒢 :: ('v ⇒ 'v) monoid where
  bijection_𝒱𝒢 ≡ BijGroup (UNIV :: 'v set)
```

Permutation action on the set of voters. Invariance under this action implies anonymity.

```
fun φ-anon :: ('a, 'v) Election set ⇒ ('v ⇒ 'v) ⇒
        (('a, 'v) Election ⇒ ('a, 'v) Election) where
  φ-anon ℰ π = extensional-continuation (rename π) ℰ
```

fun *anonymity*$_{\mathcal{R}}$ :: (*'a*, *'v*) *Election set* ⇒ (*'a*, *'v*) *Election rel* **where**
anonymity$_{\mathcal{R}}$ $\mathcal{E}$ = *action-induced-rel* (*carrier bijection*$_{\mathcal{VG}}$) $\mathcal{E}$ (φ*-anon* $\mathcal{E}$)

Neutrality

fun *rel-rename* :: (*'a* ⇒ *'a*, *'a Preference-Relation*) *binary-fun* **where**
rel-rename π *r* = {(π *a*, π *b*) | *a b*. (*a*, *b*) ∈ *r*}

fun *alternatives-rename* :: (*'a* ⇒ *'a*, (*'a*, *'v*) *Election*) *binary-fun* **where**
alternatives-rename π $\mathcal{E}$ =
(π ' (*alternatives-*$\mathcal{E}$ $\mathcal{E}$), *voters-*$\mathcal{E}$ $\mathcal{E}$, (*rel-rename* π) ∘ (*profile-*$\mathcal{E}$ $\mathcal{E}$))

Bijection group on the set of alternatives. Invariance under this action implies neutrality.

definition *bijection*$_{\mathcal{AG}}$ :: (*'a* ⇒ *'a*) *monoid* **where**
bijection$_{\mathcal{AG}}$ ≡ *BijGroup* (*UNIV* :: *'a set*)

Permutation action on the set of alternatives.

fun φ*-neutral* :: (*'a*, *'v*) *Election set* ⇒
(*'a* ⇒ *'a*, (*'a*, *'v*) *Election*) *binary-fun* **where**
φ*-neutral* $\mathcal{E}$ π = *extensional-continuation* (*alternatives-rename* π) $\mathcal{E}$

fun *neutrality*$_{\mathcal{R}}$ :: (*'a*, *'v*) *Election set* ⇒ (*'a*, *'v*) *Election rel* **where**
neutrality$_{\mathcal{R}}$ $\mathcal{E}$ = *action-induced-rel* (*carrier bijection*$_{\mathcal{AG}}$) $\mathcal{E}$ (φ*-neutral* $\mathcal{E}$)

fun ψ*-neutral*$_{\mathrm{c}}$ :: (*'a* ⇒ *'a*, *'a*) *binary-fun* **where**
ψ*-neutral*$_{\mathrm{c}}$ π *r* = π *r*

fun ψ*-neutral*$_{\mathrm{w}}$ :: (*'a* ⇒ *'a*, *'a rel*) *binary-fun* **where**
ψ*-neutral*$_{\mathrm{w}}$ π *r* = *rel-rename* π *r*

Homogeneity

fun *homogeneity*$_{\mathcal{R}}$ :: (*'a*, *'v*) *Election set* ⇒ (*'a*, *'v*) *Election rel* **where**
homogeneity$_{\mathcal{R}}$ $\mathcal{E}$ =
{(*E*, *E'*). *E* ∈ $\mathcal{E}$
∧ *alternatives-*$\mathcal{E}$ *E* = *alternatives-*$\mathcal{E}$ *E'*
∧ *finite* (*voters-*$\mathcal{E}$ *E*) ∧ *finite* (*voters-*$\mathcal{E}$ *E'*)
∧ (∃ *n* > *0*. ∀ *r* :: *'a Preference-Relation*.
vote-count r E = *n* ∗ (*vote-count r E'*))}

fun *copy-list* :: *nat* ⇒ *'x list* ⇒ *'x list* **where**
copy-list 0 l = [] |
copy-list (*Suc n*) *l* = *copy-list n l* @ *l*

fun *homogeneity*$_{\mathcal{R}}$*'* :: (*'a*, *'v* :: *linorder*) *Election set* ⇒ (*'a*, *'v*) *Election rel* **where**
homogeneity$_{\mathcal{R}}$*'* $\mathcal{E}$ =
{(*E*, *E'*). *E* ∈ $\mathcal{E}$

```
    ∧  alternatives-ℰ E = alternatives-ℰ E′
    ∧ finite (voters-ℰ E) ∧ finite (voters-ℰ E′)
    ∧ (∃ n > 0.
        to-list (voters-ℰ E′) (profile-ℰ E′) =
          copy-list n (to-list (voters-ℰ E) (profile-ℰ E)))}
```

Reversal Symmetry

```
fun reverse-rel :: ′a rel ⇒ ′a rel where
  reverse-rel r = {(a, b). (b, a) ∈ r}

fun rel-app :: (′a rel ⇒ ′a rel) ⇒ (′a, ′v) Election ⇒ (′a, ′v) Election where
  rel-app f (A, V, p) = (A, V, f ∘ p)

definition reversal_𝒢 :: (′a rel ⇒ ′a rel) monoid where
  reversal_𝒢 ≡ (|carrier = {reverse-rel, id}, monoid.mult = comp, one = id|)

fun φ-reverse :: (′a, ′v) Election set
              ⇒ (′a rel ⇒ ′a rel, (′a, ′v) Election) binary-fun where
  φ-reverse ℰ φ = extensional-continuation (rel-app φ) ℰ

fun ψ-reverse :: (′a rel ⇒ ′a rel, ′a rel) binary-fun where
  ψ-reverse φ r = φ r

fun reversal_ℛ :: (′a, ′v) Election set ⇒  (′a, ′v) Election rel where
  reversal_ℛ ℰ = action-induced-rel (carrier reversal_𝒢) ℰ (φ-reverse ℰ)
```

1.10.2 Auxiliary Lemmas

```
fun n-app :: nat ⇒ (′x ⇒ ′x) ⇒ (′x ⇒ ′x) where
  n-app-id: n-app 0 f = id |
  n-app-suc: n-app (Suc n) f = f ∘ n-app n f

lemma n-app-rewrite:
  fixes
    f :: ′x ⇒ ′x and
    n :: nat and
    x :: ′x
  shows (f ∘ n-app n f) x = (n-app n f ∘ f) x
proof (unfold comp-def, induction n f arbitrary: x rule: n-app.induct)
  case (1 f)
  fix
    f :: ′x ⇒ ′x and
    x :: ′x
  show f (n-app 0 f x) = n-app 0 f (f x)
    by simp
next
  case (2 n f)
  fix
    f :: ′x ⇒ ′x and
    n :: nat and
    x :: 'x
  assume ⋀ y. f (n-app n f y) = n-app n f (f y)
  thus f (n-app (Suc n) f x) = n-app (Suc n) f (f x)
    by simp
qed

lemma n-app-leaves-set:
  fixes
    A B :: 'x set and
    f :: 'x ⇒ 'x and
    x :: 'x
  assumes
    fin-A: finite A and
    fin-B: finite B and
    x-el: x ∈ A − B and
    bij-f: bij-betw f A B
  obtains n :: nat where
    n > 0 and
    n-app n f x ∈ B − A and
    ∀ m > 0. m < n ⟶ n-app m f x ∈ A ∩ B
proof −
  have n-app-f-x-in-A: n-app 0 f x ∈ A
    using x-el
    by simp
  moreover have ex-A:
    ∃ n > 0. n-app n f x ∈ B − A ∧ (∀ m > 0. m < n ⟶ n-app m f x ∈ A)
  proof (rule ccontr,
        unfold Diff-iff conj-assoc not-ex de-Morgan-conj not-gr-zero
              simp-thms not-all not-imp disj-not1 imp-disj2)
    assume nex:
      ∀ n. n-app n f x ∈ B
        ⟶ n = 0 ∨ n-app n f x ∈ A ∨ (∃ m > 0. m < n ∧ n-app m f x ∉ A)
    hence ∀ n > 0. n-app n f x ∈ B
          ⟶ n-app n f x ∈ A ∨ (∃ m > 0. m < n ∧ n-app m f x ∉ A)
      by blast
    moreover have ¬ (∀ n > 0. n-app n f x ∈ B ⟶ n-app n f x ∈ A)
    proof (safe)
      assume in-A: ∀ n > 0. n-app n f x ∈ B ⟶ n-app n f x ∈ A
      hence ∀ n > 0. n-app n f x ∈ A ⟶ n-app (Suc n) f x ∈ A
        using n-app.simps bij-f
        unfolding bij-betw-def
        by force
      hence in-AB-imp-in-AB:
        ∀ n > 0. n-app n f x ∈ A ∩ B ⟶ n-app (Suc n) f x ∈ A ∩ B
        using n-app.simps bij-f
        unfolding bij-betw-def
        by auto
      have in-int: ∀ n > 0. n-app n f x ∈ A ∩ B
```

```
proof (clarify)
  fix n :: nat
  assume n > 0
  thus n-app n f x ∈ A ∩ B
  proof (induction n)
    case 0
    thus ?case
      by safe
  next
    case (Suc n)
    assume 0 < n ⟹ n-app n f x ∈ A ∩ B
    moreover have n = 0 ⟶ n-app (Suc n) f x = f x
      by simp
    ultimately show n-app (Suc n) f x ∈ A ∩ B
      using x-el bij-f in-A in-AB-imp-in-AB
      unfolding bij-betw-def
      by blast
  qed
qed
hence {n-app n f x | n. n > 0} ⊆ A ∩ B
  by blast
hence finite {n-app n f x | n. n > 0}
  using fin-A fin-B rev-finite-subset
  by blast
moreover have
  inj-on (λ n. n-app n f x) {n. n > 0}
    ⟶ infinite ((λ n. n-app n f x) ‘ {n. n > 0})
  using diff-is-0-eq′ finite-imageD finite-nat-set-iff-bounded lessI
        less-imp-diff-less mem-Collect-eq nless-le
  by metis
moreover have (λ n. n-app n f x) ‘ {n. n > 0} = {n-app n f x | n. n > 0}
  by auto
ultimately have ∃ n > 0 . ∃ m > n. n-app n f x = n-app m f x
  using linorder-inj-onI′ mem-Collect-eq
  by (metis (no-types, lifting))
hence ∃ n-min > 0.
    (∃ m > n-min. n-app n-min f x = n-app m f x)
  ∧ (∀ n < n-min. ¬ (0 < n ∧ (∃ m > n. n-app n f x = n-app m f x)))
  using exists-least-iff[of
          λ n. n > 0 ∧ (∃ m > n. n-app n f x = n-app m f x)]
  by presburger
then obtain n-min :: nat where
  n-min-pos: n-min > 0 and
  ex-gt-n-min: ∃ m > n-min. n-app n-min f x = n-app m f x and
  neq: ∀ n < n-min. ¬ (n > 0 ∧ (∃ m > n. n-app n f x = n-app m f x))
  by blast
then obtain m :: nat where
  m-gt-n-min: m > n-min and
  n-app-n-min-eq: n-app n-min f x = f (n-app (m − 1) f x)
```

```
      using comp-apply diff-Suc-1 less-nat-zero-code n-app.elims
      by (metis (mono-tags, lifting))
    moreover have n-app n-min f x = f (n-app (n-min − 1) f x)
      using Suc-pred′ n-min-pos comp-eq-id-dest id-comp diff-Suc-1
            less-nat-zero-code n-app.elims
      by (metis (mono-tags, opaque-lifting))
    moreover have n-app (m − 1) f x ∈ A ∧ n-app (n-min − 1) f x ∈ A
      using in-int x-el n-min-pos m-gt-n-min Diff-iff IntD1 diff-le-self id-apply
            nless-le cancel-comm-monoid-add-class.diff-cancel n-app-id
      by metis
    ultimately have n-app (m − 1) f x = n-app (n-min − 1) f x
      using bij-f
      unfolding bij-betw-def inj-def inj-on-def
      by simp
    moreover have m − 1 > n-min − 1
      using m-gt-n-min n-min-pos
      by simp
    moreover have n-app (m − 1) f x ∈ B
      using in-int m-gt-n-min n-min-pos
      by simp
    ultimately show False
      using x-el neq n-min-pos diff-less zero-less-one Diff-iff
            bot-nat-0.not-eq-extremum id-apply n-app-id
      by metis
  qed
  ultimately obtain n :: nat where
    n-pos: n > 0 and
    not-in-A: n-app n f x ∉ A and
    less-in-A: ∀ m. (0 < m ∧ m < n) ⟶ n-app m f x ∈ A
    using exists-least-iff[of λ n. n > 0 ∧ n-app n f x ∉ A]
    by blast
  hence n-app (n − 1) f x ∈ A
    using n-app-f-x-in-A bot-nat-0.not-eq-extremum diff-less zero-less-one
    by metis
  moreover have n-app n f x = f (n-app (n − 1) f x)
    using n-app-suc Suc-pred′ n-pos comp-eq-id-dest fun.map-id
    by (metis (mono-tags, opaque-lifting))
  ultimately show False
    using bij-f nex not-in-A n-pos less-in-A
    unfolding bij-betw-def
    by blast
qed
ultimately have
  ∀ n. (∀ m > 0. m < n ⟶ n-app m f x ∈ A)
          ⟶ (∀ m > 0. m < n ⟶ n-app (m − 1) f x ∈ A)
  using bot-nat-0.not-eq-extremum less-imp-diff-less
  by metis
moreover have ∀ m > 0. n-app m f x = f (n-app (m − 1) f x)
  using bot-nat-0.not-eq-extremum comp-apply diff-Suc-1 n-app.elims
```

```
    by (metis (mono-tags, lifting))
  ultimately show ?thesis
    using that bij-f imageI IntI ex-A
    unfolding bij-betw-def
    by metis
qed

lemma n-app-rev:
  fixes
    A B :: 'x set and
    f :: 'x ⇒ 'x and
    m n :: nat and
    x y :: 'x
  assumes
    x-in-A: x ∈ A and
    y-in-A: y ∈ A and
    n-geq-m: n ≥ m and
    n-app-eq-m-n: n-app n f x = n-app m f y and
    n-app-x-in-A: ∀ n' < n. n-app n' f x ∈ A and
    n-app-y-in-A: ∀ m' < m. n-app m' f y ∈ A and
    fin-A: finite A and
    fin-B: finite B and
    bij-f-A-B: bij-betw f A B
  shows n-app (n − m) f x = y
  using assms
proof (induction n f arbitrary: m x y rule: n-app.induct)
  case (1 f)
  fix
    f :: 'x ⇒ 'x and
    m :: nat and
    x y :: 'x
  assume
    m ≤ 0 and
    n-app 0 f x = n-app m f y
  thus n-app (0 − m) f x = y
    by simp
next
  case (2 n f)
  fix
    f :: 'x ⇒ 'x and
    m n :: nat and
    x y :: 'x
  assume
    bij-f: bij-betw f A B and
    x-in-A: x ∈ A and
    y-in-A: y ∈ A and
    m-leq-suc-n: m ≤ Suc n and
    x-dom: ∀ n' < Suc n. n-app n' f x ∈ A and
    y-dom: ∀ m' < m. n-app m' f y ∈ A and
    eq: n-app (Suc n) f x = n-app m f y and
    hyp:
      ⋀ m x y.
          x ∈ A ⟹
          y ∈ A ⟹
          m ≤ n ⟹
          n-app n f x = n-app m f y ⟹
          ∀ n′ < n. n-app n′ f x ∈ A ⟹
          ∀ m′ < m. n-app m′ f y ∈ A ⟹
          finite A ⟹ finite B ⟹ bij-betw f A B ⟹ n-app (n − m) f x = y
  hence m > 0 ⟶ f (n-app n f x) = f (n-app (m − 1) f y)
    using Suc-pred′ comp-apply n-app-suc
    by (metis (mono-tags, opaque-lifting))
  moreover have n-app n f x ∈ A
    using x-in-A x-dom
    by blast
  moreover have m > 0 ⟶ n-app (m − 1) f y ∈ A
    using y-dom
    by simp
  ultimately have m > 0 ⟶ n-app n f x = n-app (m − 1) f y
    using bij-f
    unfolding bij-betw-def inj-on-def
    by blast
  moreover have m − 1 ≤ n
    using m-leq-suc-n
    by simp
  hence m > 0 ⟶ n-app (n − (m − 1)) f x = y
    using hyp x-in-A y-in-A x-dom y-dom Suc-pred fin-A fin-B
        bij-f calculation less-SucI
    unfolding One-nat-def
    by metis
  thus n-app (Suc n − m) f x = y
    using eq
    by force
qed

lemma n-app-inv:
  fixes
    A B :: ′x set and
    f :: ′x ⇒ ′x and
    n :: nat and
    x :: ′x
  assumes
    x ∈ B and
    ∀ m ≥ 0. m < n ⟶ n-app m (the-inv-into A f) x ∈ B and
    bij-betw f A B
  shows n-app n f (n-app n (the-inv-into A f) x) = x
  using assms
proof (induction n f arbitrary: x rule: n-app.induct)
```

```
  case (1 f)
  fix f :: 'x ⇒ 'x
  show ?case
    by simp
next
  case (2 n f)
  fix
    n :: nat and
    f :: 'x ⇒ 'x and
    x :: 'x
  assume
    x-in-B: x ∈ B and
    bij-f: bij-betw f A B and
    stays-in-B: ∀ m ≥ 0. m < Suc n ⟶ n-app m (the-inv-into A f) x ∈ B and
    hyp: ⋀ x. x ∈ B ⟹
            ∀ m ≥ 0. m < n ⟶ n-app m (the-inv-into A f) x ∈ B ⟹
            bij-betw f A B ⟹ n-app n f (n-app n (the-inv-into A f) x) = x
  have n-app (Suc n) f (n-app (Suc n) (the-inv-into A f) x) =
    n-app n f (f (n-app (Suc n) (the-inv-into A f) x))
    using n-app-rewrite
    by simp
  also have ... = n-app n f (n-app n (the-inv-into A f) x)
    using stays-in-B bij-f
    by (simp add: f-the-inv-into-f-bij-betw)
  finally show n-app (Suc n) f (n-app (Suc n) (the-inv-into A f) x) = x
    using hyp bij-f stays-in-B x-in-B
    by simp
qed

lemma bij-betw-finite-ind-global-bij:
  fixes
    A B :: 'x set and
    f :: 'x ⇒ 'x
  assumes
    fin-A: finite A and
    fin-B: finite B and
    bij-f: bij-betw f A B
  obtains g :: 'x ⇒ 'x where
    bij g and
    ∀ a ∈ A. g a = f a and
    ∀ b ∈ B − A. g b ∈ A − B ∧ (∃ n > 0. n-app n f (g b) = b) and
    ∀ x ∈ UNIV − A − B. g x = x
proof −
  assume existence-witness:
    ⋀ g. bij g ⟹
          ∀ a ∈ A. g a = f a ⟹
          ∀ b ∈ B − A. g b ∈ A − B ∧ (∃ n > 0. n-app n f (g b) = b) ⟹
          ∀ x ∈ UNIV − A − B. g x = x ⟹ ?thesis
  have bij-inv: bij-betw (the-inv-into A f) B A
    using bij-f bij-betw-the-inv-into
    by blast
  then obtain g′ :: ′x ⇒ nat where
    g′-greater-zero: ∀ x ∈ B − A. g′ x > 0 and
    in-set-diff: ∀ x ∈ B − A. n-app (g′ x) (the-inv-into A f) x ∈ A − B and
    minimal: ∀ x ∈ B − A. ∀ n > 0.
                n < g′ x ⟶ n-app n (the-inv-into A f) x ∈ B ∩ A
    using n-app-leaves-set fin-A fin-B
    by metis
  obtain g :: ′x ⇒ ′x where
    def-g:
      g = (λ x. if x ∈ A then f x else
               (if x ∈ B − A then n-app (g′ x) (the-inv-into A f) x else x))
    by simp
  hence coincide: ∀ a ∈ A. g a = f a
    by simp
  have id: ∀ x ∈ UNIV − A − B. g x = x
    using def-g
    by simp
  have ∀ x ∈ B − A. n-app 0 (the-inv-into A f) x ∈ B
    by simp
  moreover have
    ∀ x ∈ B − A. ∀ n > 0.
        n < g′ x ⟶ n-app n (the-inv-into A f) x ∈ B
    using minimal
    by blast
  ultimately have
    ∀ x ∈ B − A. n-app (g′ x) f (n-app (g′ x) (the-inv-into A f) x) = x
    using n-app-inv bij-f DiffD1 antisym-conv2
    by metis
  hence ∀ x ∈ B − A. n-app (g′ x) f (g x) = x
    using def-g
    by simp
  with g′-greater-zero in-set-diff
  have reverse: ∀ x ∈ B − A. g x ∈ A − B ∧ (∃ n > 0. n-app n f (g x) = x)
    using def-g
    by auto
  have ∀ x ∈ UNIV − A − B. g x = id x
    using def-g
    by simp
  hence g ‘ (UNIV − A − B) = UNIV − A − B
    by simp
  moreover have g ‘ A = B
    using def-g bij-f
    unfolding bij-betw-def
    by simp
  moreover have A ∪ (UNIV − A − B) = UNIV − (B − A)
               ∧ B ∪ (UNIV − A − B) = UNIV − (A − B)
    by blast
```

```
ultimately have surj-cases: g ‘ (UNIV − (B − A)) = UNIV − (A − B)
  using image-Un
  by metis
have inj-on g A ∧ inj-on g (UNIV − A − B)
  using def-g bij-f
  unfolding bij-betw-def inj-on-def
  by simp
hence inj-cases: inj-on g (UNIV − (B − A))
  unfolding inj-on-def
  using DiffD2 DiffI bij-f bij-betwE def-g
  by (metis (no-types, lifting))
have card A = card B
  using fin-A fin-B bij-f bij-betw-same-card
  by blast
with fin-A fin-B
have finite (B − A) ∧ finite (A − B) ∧ card (B − A) = card (A − B)
  using card-le-sym-Diff finite-Diff2 nle-le
  by metis
moreover have (λ x. n-app (g′ x) (the-inv-into A f) x) ‘ (B − A) ⊆ A − B
  using in-set-diff
  by blast
moreover have inj-on (λ x. n-app (g′ x) (the-inv-into A f) x) (B − A)
  proof (unfold inj-on-def, safe)
  fix x y :: ′x
  assume
    x-in-B: x ∈ B and
    x-not-in-A: x ∉ A and
    y-in-B: y ∈ B and
    y-not-in-A: y ∉ A and
    n-app (g′ x) (the-inv-into A f) x = n-app (g′ y) (the-inv-into A f) y
  moreover from this have
    ∀ n < g′ x. n-app n (the-inv-into A f) x ∈ B and
    ∀ n < g′ y. n-app n (the-inv-into A f) y ∈ B
    using minimal Diff-iff Int-iff bot-nat-0.not-eq-extremum eq-id-iff n-app-id
    by (metis, metis)
  ultimately have x-to-y:
    n-app (g′ x − g′ y) (the-inv-into A f) x = y
      ∨ n-app (g′ y − g′ x) (the-inv-into A f) y = x
    using x-in-B y-in-B bij-inv fin-A fin-B
          n-app-rev[of x] n-app-rev[of y B x g′ x g′ y]
    by fastforce
  hence g′ x ≠ g′ y ⟶
    ((∃ n > 0. n < g′ x ∧ n-app n (the-inv-into A f) x ∈ B − A) ∨
    (∃ n > 0. n < g′ y ∧ n-app n (the-inv-into A f) y ∈ B − A))
    using g′-greater-zero x-in-B x-not-in-A y-in-B y-not-in-A Diff-iff
          diff-less-mono2 diff-zero id-apply less-Suc-eq-0-disj n-app.elims
    by (metis (full-types))
  thus x = y
    using minimal x-in-B x-not-in-A y-in-B y-not-in-A x-to-y
```

```
      by force
  qed
  ultimately have
    bij-betw (λ x. n-app (g′ x) (the-inv-into A f) x) (B − A) (A − B)
    unfolding bij-betw-def
    by (simp add: card-image card-subset-eq)
  hence bij-case: bij-betw g (B − A) (A − B)
    using def-g
    unfolding bij-betw-def inj-on-def
    by simp
  hence g ‘ UNIV = UNIV
    using surj-cases Un-Diff-cancel2 image-Un sup-top-left
    unfolding bij-betw-def
    by metis
  moreover have inj g
    using inj-cases bij-case DiffD2 DiffI imageI surj-cases
    unfolding bij-betw-def inj-def inj-on-def
    by metis
  ultimately have bij g
    unfolding bij-def
    by safe
  thus ?thesis
    using coincide id reverse existence-witness
    by blast
qed

lemma bij-betw-ext:
  fixes
    f :: ′x ⇒ ′y and
    X :: ′x set and
    Y :: ′y set
  assumes bij-betw f X Y
  shows bij-betw (extensional-continuation f X) X Y
proof −
  have ∀ x ∈ X. extensional-continuation f X x = f x
    by simp
  thus ?thesis
    using assms bij-betw-cong
    by metis
qed
```

1.10.3 Anonymity Lemmas

```
lemma anon-rel-vote-count:
  fixes
    ℰ :: (′a, ′v) Election set and
    E E′ :: (′a, ′v) Election
  assumes
    finite (voters-ℰ E) and
```

```
   (E, E′) ∈ anonymity_ℛ ℰ
  shows alternatives-ℰ E = alternatives-ℰ E′ ∧ E ∈ ℰ
         ∧ (∀ p. vote-count p E = vote-count p E′)
proof –
  have E ∈ ℰ
    using assms
    unfolding anonymity_ℛ.simps action-induced-rel.simps
    by safe
  with assms
  obtain π :: ′v ⇒ ′v where
    bijection-π: bij π and
    renamed: E′ = rename π E
    unfolding anonymity_ℛ.simps bijection_𝒱𝒢-def
    using universal-set-carrier-imp-bij-group
    by auto
  have eq-alts: alternatives-ℰ E′ = alternatives-ℰ E
    using eq-fst-iff rename.simps alternatives-ℰ.elims renamed
    by (metis (no-types))
  have ∀ v ∈ voters-ℰ E′. (profile-ℰ E′) v = (profile-ℰ E) (the-inv π v)
    unfolding profile-ℰ.simps
    using renamed rename.simps comp-apply prod.collapse snd-conv
    by (metis (no-types, lifting))
  hence rewrite:
    ∀ p. {v ∈ (voters-ℰ E′). (profile-ℰ E′) v = p} =
          {v ∈ (voters-ℰ E′). (profile-ℰ E) (the-inv π v) = p}
    by blast
  have ∀ v ∈ voters-ℰ E′. the-inv π v ∈ voters-ℰ E
    unfolding voters-ℰ.simps
    using renamed UNIV-I bijection-π bij-betw-imp-surj bij-is-inj f-the-inv-into-f
          prod.sel inj-image-mem-iff prod.collapse rename.simps
    by (metis (no-types, lifting))
  hence
    ∀ p. ∀ v ∈ voters-ℰ E′. (profile-ℰ E) (the-inv π v) = p
        ⟶ v ∈ π ‘ {v ∈ voters-ℰ E. (profile-ℰ E) v = p}
    using bijection-π f-the-inv-into-f-bij-betw image-iff
    by fastforce
  hence subset:
    ∀ p. {v ∈ voters-ℰ E′. (profile-ℰ E) (the-inv π v) = p} ⊆
          π ‘ {v ∈ voters-ℰ E. (profile-ℰ E) v = p}
    by blast
  from renamed have ∀ v ∈ voters-ℰ E. π v ∈ voters-ℰ E′
    unfolding voters-ℰ.simps
   using bijection-π bij-is-inj prod.sel inj-image-mem-iff prod.collapse rename.simps
    by (metis (mono-tags, lifting))
  hence
    ∀ p. π ‘ {v ∈ voters-ℰ E. (profile-ℰ E) v = p} ⊆
          {v ∈ voters-ℰ E′. (profile-ℰ E) (the-inv π v) = p}
    using bijection-π bij-is-inj the-inv-f-f
    by fastforce
  hence
    ∀ p. {v ∈ voters-ℰ E′. (profile-ℰ E′) v = p} =
          π ‘ {v ∈ voters-ℰ E. (profile-ℰ E) v = p}
    using subset rewrite
    by (simp add: subset-antisym)
  moreover have
    ∀ p. card (π ‘ {v ∈ voters-ℰ E. (profile-ℰ E) v = p}) =
          card {v ∈ voters-ℰ E. (profile-ℰ E) v = p}
    using bijection-π bij-betw-same-card bij-betw-subset top-greatest
    by (metis (no-types, lifting))
  ultimately show
    alternatives-ℰ E =
       alternatives-ℰ E′ ∧ E ∈ ℰ ∧ (∀ p. vote-count p E = vote-count p E′)
    using eq-alts assms
    by simp
qed

lemma vote-count-anon-rel:
  fixes
    ℰ :: ('a, 'v) Election set and
    E E′ :: ('a, 'v) Election
  assumes
    fin-voters-E: finite (voters-ℰ E) and
    fin-voters-E′: finite (voters-ℰ E′) and
    default-non-v: ∀ v. v ∉ voters-ℰ E ⟶ profile-ℰ E v = {} and
    default-non-v′: ∀ v. v ∉ voters-ℰ E′ ⟶ profile-ℰ E′ v = {} and
    eq: alternatives-ℰ E = alternatives-ℰ E′ ∧ (E, E′) ∈ ℰ × ℰ
          ∧ (∀ p. vote-count p E = vote-count p E′)
  shows (E, E′) ∈ anonymity_ℛ ℰ
proof −
  have ∀ p. card {v ∈ voters-ℰ E. profile-ℰ E v = p} =
              card {v ∈ voters-ℰ E′. profile-ℰ E′ v = p}
    using eq
    unfolding vote-count.simps
    by blast
  moreover have
    ∀ p. finite {v ∈ voters-ℰ E. profile-ℰ E v = p}
          ∧ finite {v ∈ voters-ℰ E′. profile-ℰ E′ v = p}
    using assms
    by simp
  ultimately have
    ∀ p. ∃ π_p. bij-betw π_p
        {v ∈ voters-ℰ E. profile-ℰ E v = p}
          {v ∈ voters-ℰ E′. profile-ℰ E′ v = p}
    using bij-betw-iff-card
    by blast
  then obtain π :: 'a Preference-Relation ⇒ ('v ⇒ 'v) where
    bij-π: ∀ p. bij-betw (π p) {v ∈ voters-ℰ E. profile-ℰ E v = p}
                             {v ∈ voters-ℰ E′. profile-ℰ E′ v = p}
  by (metis (no-types))
obtain π′ :: ′v ⇒ ′v where
  π′-perm: ∀ v ∈ voters-ℰ E. π′ v = π (profile-ℰ E v) v
  by fastforce
hence ∀ v ∈ voters-ℰ E. ∀ v′ ∈ voters-ℰ E.
          π′ v = π′ v′ ⟶ π (profile-ℰ E v) v = π (profile-ℰ E v′) v′
  by simp
moreover have
  ∀ w ∈ voters-ℰ E. ∀ w′ ∈ voters-ℰ E.
      π (profile-ℰ E w) w = π (profile-ℰ E w′) w′
    ⟶ {v ∈ voters-ℰ E′. profile-ℰ E′ v = profile-ℰ E w}
        ∩ {v ∈ voters-ℰ E′. profile-ℰ E′ v = profile-ℰ E w′} ≠ {}
  using bij-π
  unfolding bij-betw-def
  by blast
moreover have
  ∀ w w′.
  {v ∈ voters-ℰ E′. profile-ℰ E′ v = profile-ℰ E w}
    ∩ {v ∈ voters-ℰ E′. profile-ℰ E′ v = profile-ℰ E w′} ≠ {}
      ⟶ profile-ℰ E w = profile-ℰ E w′
  by blast
ultimately have eq-prof:
  ∀ v ∈ voters-ℰ E. ∀ v′ ∈ voters-ℰ E.
      π′ v = π′ v′ ⟶ profile-ℰ E v = profile-ℰ E v′
  by blast
hence ∀ v ∈ voters-ℰ E. ∀ v′ ∈ voters-ℰ E.
          π′ v = π′ v′ ⟶ π (profile-ℰ E v) v = π (profile-ℰ E v) v′
  using π′-perm
  by metis
hence ∀ v ∈ voters-ℰ E. ∀ v′ ∈ voters-ℰ E. π′ v = π′ v′ ⟶ v = v′
  using bij-π eq-prof mem-Collect-eq
  unfolding bij-betw-def inj-on-def
  by (metis (mono-tags, lifting))
hence inj: inj-on π′ (voters-ℰ E)
  unfolding inj-on-def
  by simp
have π′ ‘ voters-ℰ E = {π (profile-ℰ E v) v | v. v ∈ voters-ℰ E}
  using π′-perm
  unfolding Setcompr-eq-image
  by simp
also have
  . . . = ⋃ {π p ‘ {v ∈ voters-ℰ E. profile-ℰ E v = p} | p. p ∈ UNIV}
  unfolding Union-eq
  by blast
also have
  . . . = ⋃ {{v ∈ voters-ℰ E′. profile-ℰ E′ v = p} | p. p ∈ UNIV}
  using bij-π
  unfolding bij-betw-def
  by (metis (mono-tags, lifting))
```

```
finally have π' ` voters-ℰ E = voters-ℰ E'
  by blast
with inj have bij': bij-betw π' (voters-ℰ E) (voters-ℰ E')
  using bij-π
  unfolding bij-betw-def
  by blast
then obtain π-global :: 'v ⇒ 'v where
  bijection-π_g: bij π-global and
  π-global-eq-π': ∀ v ∈ voters-ℰ E. π-global v = π' v and
  π-global-eq-n-app-π':
    ∀ v ∈ voters-ℰ E' − voters-ℰ E.
      π-global v ∈ voters-ℰ E − voters-ℰ E' ∧
      (∃ n > 0. n-app n π' (π-global v) = v) and
  π-global-non-voters: ∀ v ∈ UNIV − voters-ℰ E − voters-ℰ E'. π-global v = v
  using fin-voters-E fin-voters-E' bij-betw-finite-ind-global-bij
  by blast
hence inv: ∀ v v'. (π-global v' = v) = (v' = the-inv π-global v)
 using UNIV-I bij-betw-imp-inj-on bij-betw-imp-surj-on f-the-inv-into-f the-inv-f-f
  by metis
moreover have
  ∀ v ∈ UNIV − (voters-ℰ E' − voters-ℰ E).
      π-global v ∈ UNIV − (voters-ℰ E − voters-ℰ E')
  using π-global-eq-π' π-global-non-voters bij' bijection-π_g
        DiffD1 DiffD2 DiffI bij-betwE
  by (metis (no-types, lifting))
ultimately have
  ∀ v ∈ voters-ℰ E − voters-ℰ E'.
      the-inv π-global v ∈ voters-ℰ E' − voters-ℰ E
  using bijection-π_g π-global-eq-n-app-π' DiffD2 DiffI UNIV-I
  by metis
hence ∀ v ∈ voters-ℰ E − voters-ℰ E'. ∀ n > 0.
          profile-ℰ E (the-inv π-global v) = {}
  using default-non-v
  by simp
moreover have ∀ v ∈ voters-ℰ E − voters-ℰ E'. profile-ℰ E' v = {}
  using default-non-v'
  by simp
ultimately have comp-on-voters-diff:
  ∀ v ∈ voters-ℰ E − voters-ℰ E'.
      profile-ℰ E' v = (profile-ℰ E ∘ the-inv π-global) v
  by auto
have ∀ v ∈ voters-ℰ E'. ∃ v' ∈ voters-ℰ E. π-global v' = v ∧ π' v' = v
  using bij' imageE π-global-eq-π'
  unfolding bij-betw-def
  by (metis (mono-tags, opaque-lifting))
hence ∀ v ∈ voters-ℰ E'. ∃ v' ∈ voters-ℰ E. v' = the-inv π-global v ∧ π' v' = v
  using inv
  by metis
hence ∀ v ∈ voters-ℰ E'.
```

```
      the-inv π-global v ∈ voters-ℰ E ∧ π' (the-inv π-global v) = v
    by blast
  moreover have ∀ v' ∈ voters-ℰ E. profile-ℰ E' (π' v') = profile-ℰ E v'
    using π'-perm bij-π bij-betwE mem-Collect-eq
    by fastforce
  ultimately have comp-on-E'-voters:
    ∀ v ∈ voters-ℰ E'. profile-ℰ E' v = (profile-ℰ E ∘ the-inv π-global) v
    unfolding comp-def
    by metis
  have ∀ v ∈ UNIV − voters-ℰ E − voters-ℰ E'.
          profile-ℰ E' v = (profile-ℰ E ∘ the-inv π-global) v
    using π-global-non-voters default-non-v default-non-v' inv
    by simp
  hence profile-ℰ E' = profile-ℰ E ∘ the-inv π-global
    using comp-on-voters-diff comp-on-E'-voters
    by blast
  moreover have π-global ‘ (voters-ℰ E) = voters-ℰ E'
    using π-global-eq-π' bij' bij-betw-imp-surj-on
    by fastforce
  ultimately have E' = rename π-global E
    using rename.simps eq prod.collapse
    unfolding voters-ℰ.simps profile-ℰ.simps alternatives-ℰ.simps
    by metis
  thus ?thesis
    unfolding extensional-continuation.simps anonymity_ℛ.simps
              action-induced-rel.simps φ-anon.simps bijection_𝒱𝒢-def
    using eq bijection-π_g case-prodI rewrite-carrier
    by auto
qed

lemma rename-comp:
  fixes π π' :: 'v ⇒ 'v
  assumes
    bij π and
    bij π'
  shows rename π ∘ rename π' = rename (π ∘ π')
proof
  fix E :: ('a, 'v) Election
  have rename π' E =
      (alternatives-ℰ E, π' ‘ (voters-ℰ E), (profile-ℰ E) ∘ (the-inv π'))
    unfolding alternatives-ℰ.simps voters-ℰ.simps profile-ℰ.simps
    using prod.collapse rename.simps
    by metis
  hence
    (rename π ∘ rename π') E =
        rename π (alternatives-ℰ E, π' ‘ (voters-ℰ E), (profile-ℰ E) ∘ (the-inv π'))
    unfolding comp-def
    by presburger
  also have
```

```
    ... = (alternatives-ℰ E, π ‘ π′ ‘ (voters-ℰ E),
          (profile-ℰ E) ∘ (the-inv π′) ∘ (the-inv π))
    by simp
  also have
    ... = (alternatives-ℰ E, (π ∘ π′) ‘ (voters-ℰ E),
          (profile-ℰ E) ∘ the-inv (π ∘ π′))
    using assms the-inv-comp[of π - - π′]
    unfolding comp-def image-image
    by simp
  finally show (rename π ∘ rename π′) E = rename (π ∘ π′) E
    unfolding alternatives-ℰ.simps voters-ℰ.simps profile-ℰ.simps
    using prod.collapse rename.simps
    by metis
qed

interpretation anonymous-group-action:
  group-action bijection𝒱𝒢 well-formed-elections φ-anon well-formed-elections
proof (unfold group-action-def group-hom-def bijection𝒱𝒢-def
        group-hom-axioms-def hom-def, intro conjI group-BijGroup, safe)
  fix π :: ′v ⇒ ′v
  assume bij-carrier: π ∈ carrier (BijGroup UNIV)
  hence bij-π: bij π
    using rewrite-carrier
    by blast
  hence rename π ‘ well-formed-elections = well-formed-elections
    using rename-surj bij-π
    by blast
  moreover have inj-on (rename π) well-formed-elections
    using rename-inj bij-π subset-inj-on
    by blast
  ultimately have bij-betw (rename π) well-formed-elections well-formed-elections
    unfolding bij-betw-def
    by blast
  hence bij-betw (φ-anon well-formed-elections π) well-formed-elections well-formed-elections
    unfolding φ-anon.simps extensional-continuation.simps
    using bij-betw-ext
    by simp
  moreover have φ-anon well-formed-elections π ∈ extensional well-formed-elections
    unfolding extensional-def
    by force
  ultimately show bij-car-elect:
    φ-anon well-formed-elections π ∈ carrier (BijGroup well-formed-elections)
    unfolding BijGroup-def Bij-def
    by simp
  fix π′ :: ′v ⇒ ′v
  assume bij-carrier: π′ ∈ carrier (BijGroup UNIV)
  hence bij-π′: bij π′
    using rewrite-carrier
    by blast
```

```
hence rename π′ ‘ well-formed-elections = well-formed-elections
  using rename-surj bij-π
  by blast
moreover have inj-on (rename π′) well-formed-elections
  using rename-inj bij-π′ subset-inj-on
  by blast
ultimately have bij-betw (rename π′) well-formed-elections well-formed-elections
  unfolding bij-betw-def
  by blast
hence bij-betw (φ-anon well-formed-elections π′) well-formed-elections well-formed-elections
  unfolding φ-anon.simps extensional-continuation.simps
  using bij-betw-ext
  by simp
moreover from this have wf-closed′:
  φ-anon well-formed-elections π′ ‘ well-formed-elections ⊆ well-formed-elections
  using bij-betw-imp-surj-on
  by blast
moreover have φ-anon well-formed-elections π′ ∈ extensional well-formed-elections
  unfolding extensional-def
  by force
ultimately have bij-car-elect′:
  φ-anon well-formed-elections π′ ∈ carrier (BijGroup well-formed-elections)
  unfolding BijGroup-def Bij-def
  by simp
have
  φ-anon well-formed-elections π
     ⊗ BijGroup well-formed-elections (φ-anon well-formed-elections) π′ =
    extensional-continuation
   (φ-anon well-formed-elections π ∘ φ-anon well-formed-elections π′) well-formed-elections
  using rewrite-mult bij-car-elect bij-car-elect′
  by blast
moreover have
  ∀ E ∈ well-formed-elections.
    extensional-continuation
     (φ-anon well-formed-elections π ∘ φ-anon well-formed-elections π′)
       well-formed-elections E =
     (φ-anon well-formed-elections π ∘ φ-anon well-formed-elections π′) E
  by simp
moreover have
  ∀ E ∈ well-formed-elections.
       (φ-anon well-formed-elections π ∘ φ-anon well-formed-elections π′) E =
       rename π (rename π′ E)
  unfolding φ-anon.simps
  using wf-closed′
  by auto
moreover have
  ∀ E ∈ well-formed-elections. rename π (rename π′ E) = rename (π ∘ π′) E
  using rename-comp bij-π bij-π′ comp-apply
  by metis
  moreover have
    ∀ E ∈ well-formed-elections. rename (π ∘ π') E =
          φ-anon well-formed-elections (π ⊗ BijGroup UNIV π') E
    unfolding φ-anon.simps
    using rewrite-mult-univ bij-π bij-π' rewrite-carrier mem-Collect-eq
    by fastforce
  moreover have
    ∀ E. E ∉ well-formed-elections
        ⟶ extensional-continuation
              (φ-anon well-formed-elections π
                ∘ φ-anon well-formed-elections π') well-formed-elections E =
          undefined
    by simp
  moreover have
    ∀ E. E ∉ well-formed-elections
            ⟶ φ-anon well-formed-elections (π ⊗ BijGroup UNIV π') E =
                undefined
    by simp
  ultimately have
    ∀ E. φ-anon well-formed-elections (π ⊗ BijGroup UNIV π') E =
          (φ-anon well-formed-elections π
            ⊗ BijGroup well-formed-elections φ-anon well-formed-elections π') E
    by metis
  thus φ-anon well-formed-elections (π ⊗ BijGroup UNIV π') =
      φ-anon well-formed-elections π
        ⊗ BijGroup well-formed-elections φ-anon well-formed-elections π'
    by blast
qed

lemma (in result) anonymity-action-presv-symmetry: is-symmetry (λ E. limit
  (alternatives-ℰ E) UNIV) (Invariance (anonymityℛ well-formed-elections))
  unfolding anonymityℛ.simps
  by clarsimp
```

1.10.4 Neutrality Lemmas

```
lemma rel-rename-helper:
  fixes
    r :: 'a rel and
    π :: 'a ⇒ 'a and
    a b :: 'a
  assumes bij π
  shows (π a, π b) ∈ {(π x, π y) | x y. (x, y) ∈ r}
            ⟷ (a, b) ∈ {(x, y) | x y. (x, y) ∈ r}
proof (safe)
  fix x y :: 'a
  assume
    (x, y) ∈ r and
    π a = π x and
    π b = π y
  thus ∃ x y. (a, b) = (x, y) ∧ (x, y) ∈ r
    using assms bij-is-inj the-inv-f-f
    by metis
next
  fix x y :: 'a
  assume (a, b) ∈ r
  thus ∃ x y. (π a, π b) = (π x, π y) ∧ (x, y) ∈ r
    by metis
qed

lemma rel-rename-comp:
  fixes π π' :: 'a ⇒ 'a
  shows rel-rename (π ∘ π') = rel-rename π ∘ rel-rename π'
proof
  fix r :: 'a rel
  have rel-rename (π ∘ π') r = {(π (π' a), π (π' b)) | a b. (a, b) ∈ r}
    by simp
  also have ... = {(π a, π b) | a b. (a, b) ∈ rel-rename π' r}
    unfolding rel-rename.simps
    by blast
  finally show rel-rename (π ∘ π') r = (rel-rename π ∘ rel-rename π') r
    unfolding comp-def
    by simp
qed

lemma rel-rename-sound:
  fixes
    π :: 'a ⇒ 'a and
    r :: 'a rel and
    A :: 'a set
  assumes inj π
  shows
    refl-on A r ⟶ refl-on (π ` A) (rel-rename π r) and
    antisym r ⟶ antisym (rel-rename π r) and
    total-on A r ⟶ total-on (π ` A) (rel-rename π r) and
    Relation.trans r ⟶ Relation.trans (rel-rename π r)
proof (unfold antisym-def total-on-def Relation.trans-def, safe)
  assume refl-on A r
  thus refl-on (π ` A) (rel-rename π r)
    unfolding refl-on-def rel-rename.simps
    by blast
next
  fix a b :: 'a
  assume
    (a, b) ∈ rel-rename π r and
    (b, a) ∈ rel-rename π r
  then obtain
    c c' d d' :: 'a where
```

```
      c-rel-d: (c, d) ∈ r and
      d'-rel-c': (d', c') ∈ r and
      π_c-eq-a: π c = a and
      π_c'-eq-a: π c' = a and
      π_d-eq-b: π d = b and
      π_d'-eq-b: π d' = b
    unfolding rel-rename.simps
    by auto
  hence c = c' ∧ d = d'
    using assms
    unfolding inj-def
    by presburger
  moreover assume ∀ a b. (a, b) ∈ r ⟶ (b, a) ∈ r ⟶ a = b
  ultimately have c = d
    using d'-rel-c' c-rel-d
    by simp
  thus a = b
    using π_c-eq-a π_d-eq-b
    by simp
next
  fix a b :: 'a
  assume
    total: ∀ x ∈ A. ∀ y ∈ A. x ≠ y ⟶ (x, y) ∈ r ∨ (y, x) ∈ r and
    a-in-A: a ∈ A and
    b-in-A: b ∈ A and
    π_a-neq-π_b: π a ≠ π b and
    π_b-not-rel-π_a: (π b, π a) ∉ rel-rename π r
  hence (b, a) ∉ r ∧ a ≠ b
    unfolding rel-rename.simps
    by blast
  hence (a, b) ∈ r
    using a-in-A b-in-A total
    by blast
  thus (π a, π b) ∈ rel-rename π r
    unfolding rel-rename.simps
    by blast
next
  fix a b c :: 'a
  assume
    (a, b) ∈ rel-rename π r and
    (b, c) ∈ rel-rename π r
  then obtain
    d e s t :: 'a where
      d-rel-e: (d, e) ∈ r and
      s-rel-t: (s, t) ∈ r and
      π_d-eq-a: π d = a and
      π_s-eq-b: π s = b and
      π_t-eq-c: π t = c and
      π_e-eq-b: π e = b
    unfolding alternatives-ℰ.simps voters-ℰ.simps profile-ℰ.simps
    using rel-rename.simps Pair-inject mem-Collect-eq
    by auto
  hence s = e
    using assms rangeI range-ex1-eq
    by metis
  hence (d, e) ∈ r ∧ (e, t) ∈ r
    using d-rel-e s-rel-t
    by simp
  moreover assume ∀ x y z. (x, y) ∈ r ⟶ (y, z) ∈ r ⟶ (x, z) ∈ r
  ultimately have (d, t) ∈ r
    by blast
  thus (a, c) ∈ rel-rename π r
    unfolding rel-rename.simps
    using π_d-eq-a π_t-eq-c
    by blast
qed

lemma rename-subset:
  fixes
    r s :: 'a rel and
    a b :: 'a and
    π :: 'a ⇒ 'a
  assumes
    bij-π: bij π and
    rel-rename π r = rel-rename π s and
    (a, b) ∈ r
  shows (a, b) ∈ s
proof −
  have (π a, π b) ∈ {(π a, π b) | a b. (a, b) ∈ s}
    using assms
    unfolding rel-rename.simps
    by blast
  hence ∃ c d. (c, d) ∈ s ∧ π c = π a ∧ π d = π b
    by fastforce
  moreover have ∀ c d. π c = π d ⟶ c = d
    using bij-π bij-pointE
    by metis
  ultimately show (a, b) ∈ s
    by blast
qed

lemma rel-rename-bij:
  fixes π :: 'a ⇒ 'a
  assumes bij-π: bij π
  shows bij (rel-rename π)
proof (unfold bij-def inj-def surj-def, safe)
  fix
    r s :: 'a rel and
    a b :: 'a
  assume rename: rel-rename π r = rel-rename π s
  {
    moreover assume (a, b) ∈ r
    ultimately have (π a, π b) ∈ {(π a, π b) | a b. (a, b) ∈ s}
      unfolding rel-rename.simps
      by blast
    hence ∃ c d. (c, d) ∈ s ∧ π c = π a ∧ π d = π b
      by fastforce
    moreover have ∀ c d. π c = π d ⟶ c = d
      using bij-π bij-pointE
      by metis
    ultimately show subset: (a, b) ∈ s
      by blast
  }
  moreover assume (a, b) ∈ s
  ultimately show (a, b) ∈ r
    using rename rename-subset bij-π
    by (metis (no-types))
next
  fix r :: 'a rel
  have rel-rename π {((the-inv π) a, (the-inv π) b) | a b. (a, b) ∈ r} =
          {(π ((the-inv π) a), π ((the-inv π) b)) | a b. (a, b) ∈ r}
    by auto
  also have ... = {(a, b) | a b. (a, b) ∈ r}
    using the-inv-f-f bij-π
    by (simp add: f-the-inv-into-f-bij-betw)
  finally have rel-rename π (rel-rename (the-inv π) r) = r
    by simp
  thus ∃ s. r = rel-rename π s
    by blast
qed

lemma alternatives-rename-comp:
  fixes π π' :: 'a ⇒ 'a
  shows alternatives-rename π ∘ alternatives-rename π' =
          alternatives-rename (π ∘ π')
proof
  fix ℰ :: ('a, 'v) Election
  have (alternatives-rename π ∘ alternatives-rename π') ℰ =
      (π ` π' ` (alternatives-ℰ ℰ), voters-ℰ ℰ,
        (rel-rename π) ∘ (rel-rename π') ∘ (profile-ℰ ℰ))
    by (simp add: fun.map-comp)
  also have
    ... = ((π ∘ π') ` (alternatives-ℰ ℰ), voters-ℰ ℰ,
            (rel-rename (π ∘ π')) ∘ (profile-ℰ ℰ))
    using rel-rename-comp image-comp
    by metis
  also have ... = alternatives-rename (π ∘ π') ℰ
```

```
    by simp
  finally show
    (alternatives-rename π ∘ alternatives-rename π′) ℰ =
        alternatives-rename (π ∘ π′) ℰ
    by blast
qed

lemma alternatives-rename-sound:
  fixes
    A A′ :: ′a set and
    V V′ :: ′v set and
    p p′ :: (′a, ′v) Profile and
    π :: ′a ⇒ ′a
  assumes
    bij-π: bij π and
    wf-elects: (A, V, p) ∈ well-formed-elections and
    renamed: (A′, V′, p′) = alternatives-rename π (A, V, p)
  shows (A′, V′, p′) ∈ well-formed-elections
proof −
  have
    A′ = π ‘ A and
    V = V′
    using renamed
    by (simp, simp)
  moreover from this have ∀ v ∈ V′. linear-order-on A (p v)
    using wf-elects
    unfolding well-formed-elections-def profile-def
    by simp
  moreover have ∀ v ∈ V′. p′ v = rel-rename π (p v)
    using renamed
    by simp
  ultimately have ∀ v ∈ V′. linear-order-on A′ (p′ v)
    unfolding linear-order-on-def partial-order-on-def preorder-on-def
    using bij-π rel-rename-sound bij-is-inj
    by metis
  thus (A′, V′, p′) ∈ well-formed-elections
    unfolding well-formed-elections-def profile-def
    by simp
qed

lemma alternatives-rename-bij:
  fixes π :: (′a ⇒ ′a)
  assumes bij-π: bij π
 shows bij-betw (alternatives-rename π) well-formed-elections well-formed-elections
proof (unfold bij-betw-def, safe, intro inj-onI, clarify)
  fix
    A A′ :: ′a set and
    V V′ :: ′v set and
    p p′ :: (′a, ′v) Profile
  assume
    renamed: alternatives-rename π (A, V, p) = alternatives-rename π (A′, V′, p′)
  hence
    π-eq-img-A-A′: π ‘ A = π ‘ A′ and
    rel-rename-eq: rel-rename π ∘ p = rel-rename π ∘ p′
    by (simp, simp)
  hence (the-inv (rel-rename π)) ∘ rel-rename π ∘ p =
          (the-inv (rel-rename π)) ∘ rel-rename π ∘ p′
    using fun.map-comp
    by metis
  also have (the-inv (rel-rename π)) ∘ rel-rename π = id
    using bij-π rel-rename-bij inv-o-cancel surj-imp-inv-eq the-inv-f-f
    unfolding bij-betw-def
    by (metis (no-types, opaque-lifting))
  finally have p = p′
    by simp
  hence
    A = A′ and
    p = p′
    using bij-π π-eq-img-A-A′ bij-betw-imp-inj-on inj-image-eq-iff
    by (metis, safe)
  thus A = A′ ∧ (V, p) = (V′, p′)
    using renamed
    by simp
next
  fix
    A A′ :: ′a set and
    V V′ :: ′v set and
    p p′ :: (′a, ′v) Profile
  assume renamed: (A′, V′, p′) = alternatives-rename π (A, V, p)
  hence rewr: V = V′ ∧ A′ = π ‘ A
    by simp
  moreover assume (A, V, p) ∈ well-formed-elections
  ultimately have ∀ v ∈ V′. linear-order-on A (p v)
    unfolding well-formed-elections-def profile-def
    by simp
  moreover have ∀ v ∈ V′. p′ v = rel-rename π (p v)
    using renamed
    by simp
  ultimately have ∀ v ∈ V′. linear-order-on A′ (p′ v)
    unfolding linear-order-on-def partial-order-on-def preorder-on-def
    using rewr rel-rename-sound bij-is-inj assms
    by metis
  thus (A′, V′, p′) ∈ well-formed-elections
    unfolding well-formed-elections-def profile-def
    by simp
next
  fix
    A :: ′a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assume wf-elects: (A, V, p) ∈ well-formed-elections
  have rename-inv:
    alternatives-rename (the-inv π) (A, V, p) =
        ((the-inv π) ‘ A, V, rel-rename (the-inv π) ∘ p)
    by simp
  also have
    alternatives-rename π ((the-inv π) ‘ A, V, rel-rename (the-inv π) ∘ p) =
      (π ‘ (the-inv π) ‘ A, V, rel-rename π ∘ rel-rename (the-inv π) ∘ p)
    by auto
  also have ... = (A, V, rel-rename (π ∘ the-inv π) ∘ p)
    using bij-π rel-rename-comp[of π] the-inv-f-f
    by (simp add: bij-betw-imp-surj-on bij-is-inj f-the-inv-into-f image-comp)
  also have (A, V, rel-rename (π ∘ the-inv π) ∘ p) = (A, V, rel-rename id ∘ p)
    using UNIV-I assms comp-apply f-the-inv-into-f-bij-betw id-apply
    by metis
  finally have
    alternatives-rename π (alternatives-rename (the-inv π) (A, V, p)) =
        (A, V, p)
    unfolding rel-rename.simps
    by auto
  moreover have alternatives-rename (the-inv π) (A, V, p) ∈ well-formed-elections
    using rename-inv wf-elects alternatives-rename-sound bij-π bij-betw-the-inv-into
    by (metis (no-types))
  ultimately show (A, V, p) ∈ alternatives-rename π ‘ well-formed-elections
    using image-eqI
    by metis
qed

interpretation φ-neutral-action: group-action bijection_𝒜𝒢 well-formed-elections
        φ-neutral well-formed-elections
proof (unfold group-action-def group-hom-def group-hom-axioms-def hom-def
              bijection_𝒜𝒢-def, intro conjI group-BijGroup, safe)
  fix π :: 'a ⇒ 'a
  assume bij-carrier: π ∈ carrier (BijGroup UNIV)
  hence
   bij-betw (φ-neutral well-formed-elections π) well-formed-elections well-formed-elections
    using universal-set-carrier-imp-bij-group alternatives-rename-bij bij-betw-ext
    unfolding φ-neutral.simps
    by metis
  thus bij-carrier-elect:
    φ-neutral well-formed-elections π ∈ carrier (BijGroup well-formed-elections)
    unfolding φ-neutral.simps BijGroup-def Bij-def extensional-def
    by simp
  fix π' :: 'a ⇒ 'a
  assume bij-carrier': π' ∈ carrier (BijGroup UNIV)
  hence
   bij-betw (φ-neutral well-formed-elections π') well-formed-elections well-formed-elections
  using universal-set-carrier-imp-bij-group alternatives-rename-bij bij-betw-ext
  unfolding φ-neutral.simps
  by metis
hence bij-carrier-elect':
  φ-neutral well-formed-elections π' ∈ carrier (BijGroup well-formed-elections)
  unfolding φ-neutral.simps BijGroup-def Bij-def extensional-def
  by simp
hence carrier-elects:
  φ-neutral well-formed-elections π ∈ carrier (BijGroup well-formed-elections)
   ∧ φ-neutral well-formed-elections π' ∈ carrier (BijGroup well-formed-elections)
  using bij-carrier-elect
  by metis
hence bij-betw (φ-neutral well-formed-elections π') well-formed-elections well-formed-elections
  unfolding BijGroup-def Bij-def extensional-def
  by auto
hence wf-closed':
 φ-neutral well-formed-elections π' ` well-formed-elections ⊆ well-formed-elections
  using bij-betw-imp-surj-on
  by blast
have φ-neutral well-formed-elections π
        ⊗ BijGroup well-formed-elections φ-neutral well-formed-elections π' =
    extensional-continuation
      (φ-neutral well-formed-elections π ∘ φ-neutral well-formed-elections π')
           well-formed-elections
  using carrier-elects rewrite-mult
  by auto
moreover have
  ∀ ℰ ∈ well-formed-elections. extensional-continuation
      (φ-neutral well-formed-elections π ∘ φ-neutral well-formed-elections π')
         well-formed-elections ℰ =
       (φ-neutral well-formed-elections π ∘ φ-neutral well-formed-elections π') ℰ
  by simp
moreover have
  ∀ ℰ ∈ well-formed-elections.
    (φ-neutral well-formed-elections π ∘ φ-neutral well-formed-elections π') ℰ =
      alternatives-rename π (alternatives-rename π' ℰ)
  unfolding φ-neutral.simps
  using wf-closed'
  by auto
moreover have
  ∀ ℰ ∈ well-formed-elections.
      alternatives-rename π (alternatives-rename π' ℰ) =
         alternatives-rename (π ∘ π') ℰ
  using alternatives-rename-comp comp-apply
  by metis
moreover have
  ∀ ℰ ∈ well-formed-elections. alternatives-rename (π ∘ π') ℰ =
      φ-neutral well-formed-elections (π ⊗ BijGroup UNIV π') ℰ
  using rewrite-mult-univ bij-carrier bij-carrier'
```

```
    unfolding φ-anon.simps φ-neutral.simps extensional-continuation.simps
    by metis
  moreover have
    ∀ ℰ. ℰ ∉ well-formed-elections ⟶
      extensional-continuation
        (φ-neutral well-formed-elections π ∘ φ-neutral well-formed-elections π')
          well-formed-elections ℰ = undefined
    by simp
  moreover have
    ∀ ℰ. ℰ ∉ well-formed-elections
        ⟶ φ-neutral well-formed-elections (π ⊗ BijGroup UNIV π') ℰ = undefined
    by simp
  ultimately have
    ∀ ℰ. φ-neutral well-formed-elections (π ⊗ BijGroup UNIV π') ℰ =
      (φ-neutral well-formed-elections π
        ⊗ BijGroup well-formed-elections φ-neutral well-formed-elections π') ℰ
    by metis
  thus
    φ-neutral well-formed-elections (π ⊗ BijGroup UNIV π') =
      φ-neutral well-formed-elections π
        ⊗ BijGroup well-formed-elections φ-neutral well-formed-elections π'
    by blast
qed

interpretation ψ-neutral_c-action: group-action bijection_𝒜𝒢 UNIV ψ-neutral_c
proof (unfold group-action-def group-hom-def hom-def bijection_𝒜𝒢-def
            group-hom-axioms-def, intro conjI group-BijGroup, safe)
  fix π :: 'a ⇒ 'a
  assume π ∈ carrier (BijGroup UNIV)
  hence bij π
    unfolding BijGroup-def Bij-def
    by simp
  thus ψ-neutral_c π ∈ carrier (BijGroup UNIV)
    unfolding ψ-neutral_c.simps
    using rewrite-carrier
    by blast
  fix π' :: 'a ⇒ 'a
  show ψ-neutral_c (π ⊗ BijGroup UNIV π') =
          ψ-neutral_c π ⊗ BijGroup UNIV ψ-neutral_c π'
    unfolding ψ-neutral_c.simps
    by safe
qed

interpretation ψ-neutral_w-action: group-action bijection_𝒜𝒢 UNIV ψ-neutral_w
proof (unfold group-action-def group-hom-def hom-def bijection_𝒜𝒢-def
            group-hom-axioms-def, intro conjI group-BijGroup, safe)
  fix π :: 'a ⇒ 'a
  assume bij-carrier: π ∈ carrier (BijGroup UNIV)
  hence bij π
```

```
    unfolding bijection_𝒜𝒢-def BijGroup-def Bij-def
    by simp
  hence bij (ψ-neutral_w π)
    unfolding bijection_𝒜𝒢-def BijGroup-def Bij-def ψ-neutral_w.simps
    using rel-rename-bij
    by blast
  thus group-elem: ψ-neutral_w π ∈ carrier (BijGroup UNIV)
    using rewrite-carrier
    by blast
  moreover fix π' :: 'a ⇒ 'a
  assume bij-carrier': π' ∈ carrier (BijGroup UNIV)
  hence bij π'
    unfolding bijection_𝒜𝒢-def BijGroup-def Bij-def
    by simp
  hence bij (ψ-neutral_w π')
    unfolding bijection_𝒜𝒢-def BijGroup-def Bij-def ψ-neutral_w.simps
    using rel-rename-bij
    by blast
  hence group-elem': ψ-neutral_w π' ∈ carrier (BijGroup UNIV)
    using rewrite-carrier
    by blast
  moreover have ψ-neutral_w (π ⊗_BijGroup UNIV π') = ψ-neutral_w (π ∘ π')
    using bij-carrier bij-carrier' rewrite-mult-univ
    by metis
  ultimately show
    ψ-neutral_w (π ⊗ _BijGroup UNIV π') =
          ψ-neutral_w π ⊗ _BijGroup UNIV ψ-neutral_w π'
    using rewrite-mult-univ
    by fastforce
qed

lemma neutrality-action-presv-𝒮𝒞ℱ-symmetry: is-symmetry (λ ℰ. limit-𝒮𝒞ℱ (alternatives-ℰ
ℰ) UNIV)
            (action-induced-equivariance (carrier bijection_𝒜𝒢) well-formed-elections
                               (φ-neutral well-formed-elections) (set-action ψ-neutral_c))
proof (unfold rewrite-equivariance, safe)
  fix
    π :: 'a ⇒ 'a and
    A :: 'a set and
    V :: 'v set and
    p :: 'v ⇒ ('a × 'a) set and
    r :: 'a
  assume
    carrier-π: π ∈ carrier bijection_𝒜𝒢 and
    prof: (A, V, p) ∈ well-formed-elections
  {
    moreover assume
      r ∈ limit-𝒮𝒞ℱ
        (alternatives-ℰ (φ-neutral well-formed-elections π (A, V, p))) UNIV
    ultimately show
      r ∈ set-action ψ-neutral_c π (limit-𝒮𝒞ℱ (alternatives-ℰ (A, V, p)) UNIV)
      by auto
  }
  {
    moreover assume
      r ∈ set-action ψ-neutral_c π (limit-𝒮𝒞ℱ (alternatives-ℰ (A, V, p)) UNIV)
    ultimately show
      r ∈ limit-𝒮𝒞ℱ
        (alternatives-ℰ (φ-neutral well-formed-elections π (A, V, p))) UNIV
      using prof
      by simp
  }
qed

lemma neutrality-action-presv-𝒮𝒲ℱ-symmetry: is-symmetry (λ ℰ. limit-𝒮𝒲ℱ
  (alternatives-ℰ ℰ) UNIV) (action-induced-equivariance (carrier bijection_𝒜𝒢) well-formed-elections
                              (φ-neutral well-formed-elections) (set-action ψ-neutral_w))
proof (unfold rewrite-equivariance voters-ℰ.simps profile-ℰ.simps set-action.simps,
        safe)
  show ⋀ π A V p r.
          π ∈ carrier bijection_𝒜𝒢 ⟹ (A, V, p) ∈ well-formed-elections
        ⟹ r ∈ limit-𝒮𝒲ℱ
          (alternatives-ℰ (φ-neutral well-formed-elections π (A, V , p))) UNIV
        ⟹ r ∈ ψ-neutral_w π ‘ limit-𝒮𝒲ℱ (alternatives-ℰ (A, V, p)) UNIV
  proof −
    fix
      π :: 'c ⇒ 'c and
      A :: 'c set and
      V :: 'v set and
      p :: ('c, 'v) Profile and
      r :: 'c rel
    let ?r-inv = ψ-neutral_w (the-inv π) r
    assume
      carrier-π: π ∈ carrier bijection_𝒜𝒢 and
      prof: (A, V, p) ∈ well-formed-elections
    have inv-carrier: the-inv π ∈ carrier bijection_𝒜𝒢
      using carrier-π bij-betw-the-inv-into
      unfolding bijection_𝒜𝒢-def rewrite-carrier
      by simp
    moreover have the-inv π ∘ π = id
      using carrier-π universal-set-carrier-imp-bij-group bij-is-inj the-inv-f-f
      unfolding bijection_𝒜𝒢-def
      by fastforce
    moreover have 1 bijection_𝒜𝒢 = id
      unfolding bijection_𝒜𝒢-def BijGroup-def
      by auto
    ultimately have the-inv π ⊗ bijection_𝒜𝒢 π = 1 bijection_𝒜𝒢
      using carrier-π rewrite-mult-univ
      unfolding bijection_AG-def
      by metis
    hence inv bijection_AG π = the-inv π
      using carrier-π inv-carrier ψ-neutral_c-action.group-hom group.inv-closed
          group.inv-solve-right group.l-inv group-BijGroup group-hom.hom-one
          group-hom.one-closed
      unfolding bijection_AG-def
      by metis
    hence neutral-r: r = ψ-neutral_w π ?r-inv
     using carrier-π inv-carrier iso-tuple-UNIV-I ψ-neutral_w-action.orbit-sym-aux
      by metis
    have bij-inv: bij (the-inv π)
      using carrier-π bij-betw-the-inv-into universal-set-carrier-imp-bij-group
      unfolding bijection_AG-def
      by blast
    hence the-inv-π: (the-inv π) ‘ π ‘ A = A
      using carrier-π UNIV-I bij-betw-imp-surj universal-set-carrier-imp-bij-group
          f-the-inv-into-f-bij-betw image-f-inv-f surj-imp-inv-eq
      unfolding bijection_AG-def
      by metis
    assume
      r ∈ limit-SWF
        (alternatives-E (φ-neutral well-formed-elections π (A, V, p))) UNIV
    hence r ∈ limit-SWF (π ‘ A) UNIV
      unfolding φ-neutral.simps
      using prof
      by simp
    hence linear-order-on (π ‘ A) r
      by auto
    hence lin-inv: linear-order-on A ?r-inv
      using rel-rename-sound bij-inv bij-is-inj the-inv-π
    unfolding ψ-neutral_w.simps linear-order-on-def preorder-on-def partial-order-on-def
      by metis
    hence ∀ (a, b) ∈ ?r-inv. a ∈ A ∧ b ∈ A
      unfolding linear-order-on-def partial-order-on-def preorder-on-def
      using refl-on-def'
      by metis
    hence limit A ?r-inv = {(a, b). (a, b) ∈ ?r-inv}
      by auto
    also have ... = ?r-inv
      by blast
    finally have ... = limit A ?r-inv
      by blast
    hence ?r-inv ∈ limit-SWF (alternatives-E (A, V, p)) UNIV
      unfolding limit-SWF.simps alternatives-E.simps
      using lin-inv UNIV-I fst-conv mem-Collect-eq iso-tuple-UNIV-I CollectI
      by (metis (mono-tags, lifting))
    thus lim-el-π:
      r ∈ ψ-neutral_w π ‘ limit-SWF (alternatives-E (A, V, p)) UNIV
```

```
      using neutral-r
      by blast
  qed
  moreover
  fix
    π :: 'a ⇒ 'a and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    r :: 'a rel
  assume
    carrier-π: π ∈ carrier bijection_𝒜𝒢 and
    prof: (A, V, p) ∈ well-formed-elections
  hence prof-π:
    φ-neutral well-formed-elections π (A, V, p) ∈ well-formed-elections
    using φ-neutral-action.element-image
    by blast
  moreover have inv-group-elem: inv _bijection_𝒜𝒢 π ∈ carrier bijection_𝒜𝒢
    using carrier-π ψ-neutral_c-action.group-hom group.inv-closed
    unfolding group-hom-def
    by metis
  moreover have φ-neutral well-formed-elections (inv _bijection_𝒜𝒢 π)
        (φ-neutral well-formed-elections π (A, V, p)) ∈ well-formed-elections
    using prof φ-neutral-action.element-image inv-group-elem prof-π
    by metis
  moreover assume r ∈ limit-𝒮𝒲ℱ (alternatives-ℰ (A, V, p)) UNIV
  hence r ∈ limit-𝒮𝒲ℱ
      (alternatives-ℰ (φ-neutral well-formed-elections (inv _bijection_𝒜𝒢 π)
        (φ-neutral well-formed-elections π (A, V, p)))) UNIV
    using φ-neutral-action.orbit-sym-aux carrier-π prof
    by metis
  ultimately have
    r ∈ ψ-neutral_w (inv _bijection_𝒜𝒢 π) `
      limit-𝒮𝒲ℱ
        (alternatives-ℰ (φ-neutral well-formed-elections π (A, V, p))) UNIV
    using prod.collapse
    by metis
  thus ψ-neutral_w π r ∈ limit-𝒮𝒲ℱ
          (alternatives-ℰ (φ-neutral well-formed-elections π (A, V, p))) UNIV
    using carrier-π ψ-neutral_w-action.group-action-axioms
          ψ-neutral_w-action.inj-prop group-action.orbit-sym-aux
          inj-image-mem-iff inv-group-elem iso-tuple-UNIV-I
    by (metis (no-types, lifting))
qed
```

1.10.5 Homogeneity Lemmas

```
definition reflp-on' :: 'a set ⇒ 'a rel ⇒ bool where
    reflp-on' A r ≡ reflp-on A (λ x y. (x, y) ∈ r)
```

```
lemma refl-homogeneity_ℛ:
  fixes ℰ :: ('a, 'v) Election set
  assumes ℰ ⊆ finite-elections-𝒱
  shows reflp-on' ℰ (homogeneity_ℛ ℰ)
  using assms
  unfolding reflp-on'-def reflp-on-def finite-elections-𝒱-def
  by auto

lemma (in result) homogeneity-action-presv-symmetry:
  is-symmetry (λ ℰ. limit (alternatives-ℰ ℰ) UNIV)
        (Invariance (homogeneity_ℛ UNIV))
  by simp

lemma refl-homogeneity_ℛ':
  fixes ℰ :: ('a, 'v :: linorder) Election set
  assumes ℰ ⊆ finite-elections-𝒱
  shows reflp-on' ℰ (homogeneity_ℛ' ℰ)
  using assms
  unfolding homogeneity_ℛ'.simps reflp-on'-def reflp-on-def finite-elections-𝒱-def
  by auto

lemma (in result) homogeneity'-action-presv-symmetry:
  is-symmetry (λ ℰ. limit (alternatives-ℰ ℰ) UNIV)
        (Invariance (homogeneity_ℛ' UNIV))
  by simp
```

1.10.6 Reversal Symmetry Lemmas

```
lemma reverse-reverse-id: reverse-rel ∘ reverse-rel = id
  by auto

lemma reverse-rel-limit:
  fixes
    A :: 'a set and
    r :: 'a rel
  shows reverse-rel (limit A r) = limit A (reverse-rel r)
  unfolding reverse-rel.simps limit.simps
  by blast

lemma reverse-rel-lin-ord:
  fixes
    A :: 'a set and
    r :: 'a rel
  assumes linear-order-on A r
  shows linear-order-on A (reverse-rel r)
  using assms
  unfolding reverse-rel.simps linear-order-on-def partial-order-on-def
          total-on-def antisym-def preorder-on-def refl-on-def trans-def
  by blast

interpretation reversal𝒢-group: group reversal𝒢
proof
  show 1 reversal𝒢 ∈ carrier reversal𝒢
    unfolding reversal𝒢-def
    by simp
next
  show carrier reversal𝒢 ⊆ Units reversal𝒢
    unfolding reversal𝒢-def Units-def
    using reverse-reverse-id
    by auto
next
  fix α :: 'a rel ⇒ 'a rel
  show α ⊗ reversal𝒢 1 reversal𝒢 = α
    unfolding reversal𝒢-def
    by auto
  assume α-elem: α ∈ carrier reversal𝒢
  thus 1 reversal𝒢 ⊗ reversal𝒢 α = α
    unfolding reversal𝒢-def
    by auto
  fix α' :: 'a rel ⇒ 'a rel
  assume α'-elem: α' ∈ carrier reversal𝒢
  thus α ⊗ reversal𝒢 α' ∈ carrier reversal𝒢
    using α-elem reverse-reverse-id
    unfolding reversal𝒢-def
    by auto
  fix z :: 'a rel ⇒ 'a rel
  assume z ∈ carrier reversal𝒢
  thus α ⊗ reversal𝒢 α' ⊗ reversal𝒢 z = α ⊗ reversal𝒢 (α' ⊗ reversal𝒢 z)
    using α-elem α'-elem
    unfolding reversal𝒢-def
    by auto
qed

interpretation φ-reverse-action: group-action reversal𝒢 well-formed-elections
      φ-reverse well-formed-elections
proof (unfold group-action-def group-hom-def group-hom-axioms-def hom-def,
      intro conjI group-BijGroup CollectI ballI funcsetI)
  show Group.group reversal𝒢
    by safe
next
  show carrier-elect-gen:
    ⋀ π. π ∈ carrier reversal𝒢
      ⟹ φ-reverse well-formed-elections π ∈ carrier (BijGroup well-formed-elections)
  proof −
    fix π :: 'c rel ⇒ 'c rel
    assume π ∈ carrier reversal𝒢
    hence π-cases: π ∈ {id, reverse-rel}
```

```
      unfolding reversal_𝒢-def
      by auto
    hence [simp]: rel-app π ∘ rel-app π = id
      using reverse-reverse-id
      by fastforce
    have ∀ ℰ. rel-app π (rel-app π ℰ) = ℰ
      by (simp add: pointfree-idE)
   moreover have ∀ ℰ ∈ well-formed-elections. rel-app π ℰ ∈ well-formed-elections
      unfolding well-formed-elections-def profile-def
      using π-cases reverse-rel-lin-ord rel-app.simps fun.map-id
      by fastforce
    hence rel-app π ‘ well-formed-elections ⊆ well-formed-elections
      by blast
   ultimately have bij-betw (rel-app π) well-formed-elections well-formed-elections
      using bij-betw-byWitness[of well-formed-elections]
      by blast
    hence bij-betw (φ-reverse well-formed-elections π)
              well-formed-elections well-formed-elections
      unfolding φ-reverse.simps
      using bij-betw-ext
      by blast
   moreover have φ-reverse well-formed-elections π ∈ extensional well-formed-elections
      unfolding extensional-def
      by simp
    ultimately show
      φ-reverse well-formed-elections π ∈ carrier (BijGroup well-formed-elections)
      unfolding BijGroup-def Bij-def
      by simp
  qed
  moreover fix π π' :: 'a rel ⇒ 'a rel
  assume
    rev: π ∈ carrier reversal_𝒢 and
    rev': π' ∈ carrier reversal_𝒢
  ultimately have carrier-elect:
    φ-reverse well-formed-elections π ∈ carrier (BijGroup well-formed-elections)
    by blast
  have φ-reverse well-formed-elections (π ⊗ reversal_𝒢 π') =
          extensional-continuation (rel-app (π ∘ π')) well-formed-elections
    unfolding reversal_𝒢-def
    by simp
  moreover have rel-app (π ∘ π') = rel-app π ∘ rel-app π'
    using rel-app.simps
    by fastforce
  ultimately have
    φ-reverse well-formed-elections (π ⊗ reversal_𝒢 π') =
      extensional-continuation (rel-app π ∘ rel-app π') well-formed-elections
    by metis
  moreover have
    ∀ A V p. ∀ v ∈ V. linear-order-on A (p v) ⟶ linear-order-on A (π' (p v))
    using empty-iff id-apply insert-iff rev′ reverse-rel-lin-ord
    unfolding partial-object.simps reversal_𝒢-def
    by metis
  hence extensional-continuation
      (φ-reverse well-formed-elections π ∘ φ-reverse well-formed-elections π′)
          well-formed-elections =
            extensional-continuation (rel-app π ∘ rel-app π′) well-formed-elections
    unfolding well-formed-elections-def profile-def
    by fastforce
  moreover have extensional-continuation
      (φ-reverse well-formed-elections π ∘ φ-reverse well-formed-elections π′)
          well-formed-elections =
        φ-reverse well-formed-elections π
            ⊗ BijGroup well-formed-elections φ-reverse well-formed-elections π′
    using carrier-elect-gen carrier-elect rev′ rewrite-mult
    by metis
  ultimately show
    φ-reverse well-formed-elections (π ⊗ reversal_𝒢 π′) =
        φ-reverse well-formed-elections π
            ⊗ BijGroup well-formed-elections φ-reverse well-formed-elections π′
    by metis
qed

interpretation ψ-reverse-action: group-action reversal_𝒢 UNIV ψ-reverse
proof (unfold group-action-def group-hom-def group-hom-axioms-def hom-def ψ-reverse.simps,
       intro conjI group-BijGroup CollectI ballI funcsetI)
  show Group.group reversal_𝒢
    by safe
next
  fix π :: ′a rel ⇒ ′a rel
  assume π ∈ carrier reversal_𝒢
  hence π ∈ {id, reverse-rel}
    unfolding reversal_𝒢-def
    by force
  hence bij π
    using reverse-reverse-id bij-id insertE o-bij singleton-iff
    by metis
  thus π ∈ carrier (BijGroup UNIV)
    using rewrite-carrier
    by blast
next
  fix π π′ :: ′a rel ⇒ ′a rel
  assume
    π ∈ carrier reversal_𝒢 and
    π′ ∈ carrier reversal_𝒢
  hence bij π′ ∧ bij π
    using singleton-iff comp-apply id-apply involuntory-imp-bij reverse-reverse-id
    unfolding bij-id insert-iff reversal_𝒢-def partial-object.select-convs
    by (metis (mono-tags, opaque-lifting))
  hence π ⊗ BijGroup UNIV π' = π ∘ π'
    using rewrite-carrier rewrite-mult-univ
    by blast
  also have ... = π ⊗ reversal_𝒢 π'
    unfolding reversal_𝒢-def
    by force
  finally show π ⊗ reversal_𝒢 π' = π ⊗ BijGroup UNIV π'
    by presburger
qed

lemma reversal-symmetry-action-presv-symmetry: is-symmetry (λ ℰ. limit-𝒮𝒲ℱ
(alternatives-ℰ ℰ) UNIV)
        (action-induced-equivariance (carrier reversal_𝒢) well-formed-elections
            (φ-reverse well-formed-elections) (set-action ψ-reverse))
proof (unfold rewrite-equivariance, clarify)
  fix
    π :: 'a rel ⇒ 'a rel and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assume π ∈ carrier reversal_𝒢
  hence cases: π ∈ {id, reverse-rel}
    unfolding reversal_𝒢-def
    by force
  assume (A, V, p) ∈ well-formed-elections
  hence eq-A:
    alternatives-ℰ (φ-reverse well-formed-elections π (A, V, p)) = A
    by simp
  have
    ∀ r ∈ {limit A r | r. r ∈ UNIV ∧ linear-order-on A (limit A r)}.
      ∃ r' ∈ UNIV. reverse-rel r = limit A (reverse-rel r')
              ∧ reverse-rel r' ∈ UNIV ∧ linear-order-on A (limit A (reverse-rel r'))
    using reverse-rel-limit[of A] reverse-rel-lin-ord
    by force
  hence
    ∀ r ∈ {limit A r | r. r ∈ UNIV ∧ linear-order-on A (limit A r)}.
      reverse-rel r ∈ {limit A (reverse-rel r')
              | r'. reverse-rel r' ∈ UNIV
                  ∧ linear-order-on A (limit A (reverse-rel r'))}
    by blast
  moreover have
    {limit A (reverse-rel r') |
        r'. reverse-rel r' ∈ UNIV ∧ linear-order-on A (limit A (reverse-rel r'))}
      ⊆ {limit A r | r. r ∈ UNIV ∧ linear-order-on A (limit A r)}
    by blast
  ultimately have
    ∀ r ∈ limit-𝒮𝒲ℱ A UNIV. reverse-rel r ∈ limit-𝒮𝒲ℱ A UNIV
    unfolding limit-𝒮𝒲ℱ.simps
    by blast
  hence subset:
    ∀ r ∈ limit-SWF A UNIV. π r ∈ limit-SWF A UNIV
    using cases
    by fastforce
  hence ∀ r ∈ limit-SWF A UNIV. r ∈ π ‘ limit-SWF A UNIV
   using reverse-reverse-id comp-apply empty-iff id-apply image-eqI insert-iff cases
    by metis
  hence π ‘ limit-SWF A UNIV = limit-SWF A UNIV
    using subset
    by blast
  hence set-action ψ-reverse π (limit-SWF A UNIV) = limit-SWF A UNIV
    unfolding set-action.simps
    by simp
  also have
    . . . = limit-SWF
          (alternatives-E (φ-reverse well-formed-elections π (A, V, p))) UNIV
    using eq-A
    by simp
  finally show
   limit-SWF (alternatives-E (φ-reverse well-formed-elections π (A, V, p))) UNIV
=
      set-action ψ-reverse π (limit-SWF (alternatives-E (A, V, p)) UNIV)
    by simp
qed

end
```

1.11 Result-Dependent Voting Rule Properties

```
theory Property-Interpretations
  imports Voting-Symmetry
          Result-Interpretations
begin
```

1.11.1 Property Definitions

The interpretation of equivariance properties generally depends on the result type. For example, neutrality for social choice rules means that single winners are renamed when the candidates in the votes are consistently renamed. For social welfare results, the complete result rankings must be renamed.

New result-type-dependent definitions for properties can be added here.

```
locale result-properties = result +
  fixes ψ :: ('a ⇒ 'a, 'b) binary-fun and
        ν :: 'v itself
  assumes
  action-neutral: group-action bijection_𝒜𝒢 UNIV ψ and
  neutrality:
    is-symmetry (λ ℰ :: ('a, 'v) Election. limit (alternatives-ℰ ℰ) UNIV)
             (action-induced-equivariance (carrier bijection_𝒜𝒢)
                 well-formed-elections
                 (φ-neutral well-formed-elections) (set-action ψ))

sublocale result-properties ⊆ result
  using result-axioms
  by safe
```

1.11.2 Interpretations

```
global-interpretation 𝒮𝒞ℱ-properties: result-properties well-formed-𝒮𝒞ℱ
        limit-𝒮𝒞ℱ ψ-neutral_c
  unfolding result-properties-def result-properties-axioms-def
 using neutrality-action-presv-𝒮𝒞ℱ-symmetry ψ-neutral_c-action.group-action-axioms
        𝒮𝒞ℱ-result.result-axioms
  by blast

global-interpretation 𝒮𝒲ℱ-properties: result-properties well-formed-𝒮𝒲ℱ
        limit-𝒮𝒲ℱ ψ-neutral_w
  unfolding result-properties-def result-properties-axioms-def
 using neutrality-action-presv-𝒮𝒲ℱ-symmetry ψ-neutral_w-action.group-action-axioms
        𝒮𝒲ℱ-result.result-axioms
  by blast

end
```

Chapter 2

Refined Types

2.1 Preference List

```
theory Preference-List
  imports ../Preference-Relation
          HOL−Combinatorics.Multiset-Permutations
          List−Index.List-Index
begin
```

Preference lists derive from preference relations, ordered from most to least preferred alternative.

2.1.1 Well-Formedness

```
type-synonym 'a Preference-List = 'a list

abbreviation well-formed-l :: 'a Preference-List ⇒ bool where
  well-formed-l l ≡ distinct l
```

2.1.2 Auxiliary Lemmas About Lists

```
lemma is-arg-min-equal:
  fixes
    f g :: 'a ⇒ 'b :: ord and
    S :: 'a set and
    x :: 'a
  assumes ∀ x ∈ S. f x = g x
  shows is-arg-min f (λ s. s ∈ S) x = is-arg-min g (λ s. s ∈ S) x
proof (unfold is-arg-min-def, cases x ∉ S)
  case True
  thus (x ∈ S ∧ (∄y. y ∈ S ∧ f y < f x)) = (x ∈ S ∧ (∄y. y ∈ S ∧ g y < g x))
    by safe
next
  case x-in-S: False
  thus (x ∈ S ∧ (∄ y. y ∈ S ∧ f y < f x)) = (x ∈ S ∧ (∄ y. y ∈ S ∧ g y < g x))
  proof (cases ∃ y. (λ s. s ∈ S) y ∧ f y < f x)
    case y: True
    then obtain y :: 'a where
      (λ s. s ∈ S) y ∧ f y < f x
      by metis
    hence (λ s. s ∈ S) y ∧ g y < g x
      using x-in-S assms
      by metis
    thus ?thesis
      using y
      by metis
  next
    case not-y: False
    have ¬ (∃ y. (λ s. s ∈ S) y ∧ g y < g x)
    proof (safe)
      fix y :: 'a
      assume
        y ∈ S and
        g y < g x
      moreover have ∀ a ∈ S. f a = g a
        using assms
        by simp
      moreover from this have g x = f x
        using x-in-S
        by metis
      ultimately show False
        using not-y
        by (metis (no-types))
    qed
    thus ?thesis
      using x-in-S not-y
      by simp
  qed
qed

lemma list-cons-presv-finiteness:
  fixes
    A :: 'a set and
    S :: 'a list set
  assumes
    fin-A: finite A and
    fin-B: finite S
  shows finite {a#l | a l. a ∈ A ∧ l ∈ S}
proof −
  let ?P = λ A. finite {a#l | a l. a ∈ A ∧ l ∈ S}
  have ∀ a A'. finite A' ⟶ a ∉ A' ⟶ ?P A' ⟶ ?P (insert a A')
  proof (safe)
    fix
      a :: 'a and
      A′ :: ′a set
    assume finite {a#l | a l. a ∈ A′ ∧ l ∈ S}
    moreover have
      {a′#l | a′ l. a′ ∈ insert a A′ ∧ l ∈ S} =
          {a#l | a l. a ∈ A′ ∧ l ∈ S} ∪ {a#l | l. l ∈ S}
      by blast
    moreover have finite {a#l | l. l ∈ S}
      using fin-B
      by simp
    ultimately show ?P (insert a A′)
      by simp
  qed
  thus ?P A
    using finite-induct[of - ?P] fin-A
    by simp
qed

lemma listset-finiteness:
  fixes l :: ′a set list
  assumes ∀ i :: nat. i < length l ⟶ finite (l!i)
  shows finite (listset l)
  using assms
proof (induct l)
  case Nil
  show finite (listset [])
    by simp
next
  case (Cons a l)
  fix
    a :: ′a set and
    l :: ′a set list
  assume ∀ i :: nat < length (a#l). finite ((a#l)!i)
  hence
    finite a and
    ∀ i < length l. finite (l!i)
    by auto
  moreover assume
    ∀ i :: nat < length l. finite (l!i) ⟹ finite (listset l)
  ultimately have
    finite (listset l) and
    finite {a′#l′ | a′ l′. a′ ∈ a ∧ l′ ∈ (listset l)}
    using list-cons-presv-finiteness
    by (blast, blast)
  thus finite (listset (a#l))
    by (simp add: set-Cons-def)
qed

lemma all-ls-elems-same-len:
  fixes l :: ′a set list
  shows ∀ l' :: 'a list. l' ∈ listset l ⟶ length l' = length l
proof (induct l, safe)
  case Nil
  fix l :: 'a list
  assume l ∈ listset []
  thus length l = length []
    by simp
next
  case (Cons a l)
  moreover fix
    a :: 'a set and
    l :: 'a set list and
    m :: 'a list
  assume
    ∀ l'. l' ∈ listset l ⟶ length l' = length l and
    m ∈ listset (a#l)
  moreover have
    ∀ a' l' :: 'a set list. listset (a'#l') =
      {b#m | b m. b ∈ a' ∧ m ∈ listset l'}
    by (simp add: set-Cons-def)
  ultimately show length m = length (a#l)
    by force
qed

lemma all-ls-elems-in-ls-set:
  fixes l :: 'a set list
  shows ∀ l' ∈ listset l. ∀ i :: nat < length l'. l'!i ∈ l!i
proof (induct l, safe)
  case Nil
  fix
    l' :: 'a list and
    i :: nat
  assume
    l' ∈ listset [] and
    i < length l'
  thus l'!i ∈ []!i
    by simp
next
  case (Cons a l)
  moreover fix
    a :: 'a set and
    l :: 'a set list and
    l' :: 'a list and
    i :: nat
  assume
    ∀ l' ∈ listset l. ∀ i :: nat < length l'. l'!i ∈ l!i and
    l' ∈ listset (a#l) and
    i < length l'
  moreover from this have l' ∈ set-Cons a (listset l)
```

```
    by simp
  hence ∃ b m. l′ = b#m ∧ b ∈ a ∧ m ∈ (listset l)
    unfolding set-Cons-def
    by simp
  ultimately show l′!i ∈ (a#l)!i
    using nth-Cons-Suc Suc-less-eq gr0-conv-Suc
          length-Cons nth-non-equal-first-eq
    by metis
qed

lemma all-ls-in-ls-set:
  fixes l :: ′a set list
  shows ∀ l′. length l′ = length l
          ∧ (∀ i < length l′. l′!i ∈ l!i) ⟶ l′ ∈ listset l
proof (induction l, safe)
  case Nil
  fix l′ :: ′a list
  assume length l′ = length []
  thus l′ ∈ listset []
    by simp
next
  case (Cons a l)
  fix
    l :: ′a set list and
    l′ :: ′a list and
    s :: ′a set
  assume length l′ = length (s#l)
  moreover then obtain
    t :: ′a list and
    x :: ′a where
    l′-cons: l′ = x#t
    using length-Suc-conv
    by metis
  moreover assume
    ∀ m. length m = length l ∧ (∀ i < length m. m!i ∈ l!i)
          ⟶ m ∈ listset l and
    ∀ i < length l′. l′!i ∈ (s#l)!i
  ultimately have
    x ∈ s and
    t ∈ listset l
    using diff-Suc-1 diff-Suc-eq-diff-pred zero-less-diff
          zero-less-Suc length-Cons
    by (metis nth-Cons-0, metis nth-Cons-Suc)
  thus l′ ∈ listset (s#l)
    using l′-cons
    unfolding listset-def set-Cons-def
    by simp
qed
```

2.1.3 Ranking

Rank 1 is the top preference, rank 2 the second, and so on. Rank 0 does not exist.

```
fun rank-l :: 'a Preference-List ⇒ 'a ⇒ nat where
  rank-l l a = (if a ∈ set l then index l a + 1 else 0)

fun rank-l-idx :: 'a Preference-List ⇒ 'a ⇒ nat where
  rank-l-idx l a =
    (let i = index l a in
      if i = length l then 0 else i + 1)

lemma rank-l-equiv: rank-l = rank-l-idx
  unfolding member-def
  by (simp add: ext index-size-conv)

lemma rank-zero-imp-not-present:
  fixes
    p :: 'a Preference-List and
    a :: 'a
  assumes rank-l p a = 0
  shows a ∉ set p
  using assms
  by force

definition above-l :: 'a Preference-List ⇒ 'a ⇒ 'a Preference-List where
  above-l r a ≡ take (rank-l r a) r
```

2.1.4 Definition

```
fun is-less-preferred-than-l :: 'a ⇒ 'a Preference-List ⇒ 'a ⇒ bool
        (- ≲- - [50, 1000, 51] 50) where
    a ≲l b = (a ∈ set l ∧ b ∈ set l ∧ index l a ≥ index l b)

lemma rank-gt-zero:
  fixes
    l :: 'a Preference-List and
    a :: 'a
  assumes a ≲l a
  shows rank-l l a ≥ 1
  using assms
  by simp

definition pl-α :: 'a Preference-List ⇒ 'a Preference-Relation where
  pl-α l ≡ {(a, b). a ≲l b}

lemma rel-trans:
  fixes l :: 'a Preference-List
  shows trans (pl-α l)
  unfolding Relation.trans-def pl-α-def
  by simp

lemma pl-α-lin-order:
  fixes
    A :: 'a set and
    r :: 'a rel
  assumes r ∈ pl-α ‘ permutations-of-set A
  shows linear-order-on A r
proof (cases A = {}, unfold linear-order-on-def total-on-def
        partial-order-on-def antisym-def preorder-on-def,
        intro conjI impI allI ballI)
  case True
  fix x y :: 'a
  show
    refl-on A r and
    trans r and
    (x, y) ∈ r ⟹ x = y and
    x ∈ A ⟹ (x, y) ∈ r ∨ (y, x) ∈ r
    using assms True
    unfolding pl-α-def
    by (simp, simp, simp, simp)
next
  case False
  fix x y :: 'a
  show ((refl-on A r ∧ trans r)
      ∧ (∀ x y. (x, y) ∈ r ⟶ (y, x) ∈ r ⟶ x = y))
      ∧ (∀ x ∈ A. ∀ y ∈ A. x ≠ y ⟶ (x, y) ∈ r ∨ (y, x) ∈ r)
  proof (intro conjI ballI allI impI)
    have ∀ l ∈ permutations-of-set A. l ≠ []
      using assms False permutations-of-setD
      by force
    hence ∀ a ∈ A. ∀ l ∈ permutations-of-set A. (a, a) ∈ pl-α l
      unfolding is-less-preferred-than-l.simps
              permutations-of-set-def pl-α-def
      by simp
    hence ∀ a ∈ A. (a, a) ∈ r
      using assms
      by blast
    moreover have r ⊆ A × A
      using assms
      unfolding pl-α-def permutations-of-set-def
      by auto
    ultimately show refl-on A r
      unfolding refl-on-def
      by safe
  next
    show trans r
      using assms rel-trans
      by safe
  next
    fix x y :: 'a
    assume
      (x, y) ∈ r and
      (y, x) ∈ r
    moreover have
      ∀ x y. ∀ l ∈ permutations-of-set A. x ≲_l y ∧ y ≲_l x ⟶ x = y
      using is-less-preferred-than-l.simps index-eq-index-conv nle-le
      unfolding permutations-of-set-def
      by metis
    hence ∀ x y. ∀ l ∈ pl-α ` permutations-of-set A.
              (x, y) ∈ l ∧ (y, x) ∈ l ⟶ x = y
      unfolding pl-α-def permutations-of-set-def antisym-on-def
      by blast
    ultimately show x = y
      using assms
      by metis
  next
    fix x y :: 'a
    assume
      x ∈ A and
      y ∈ A and
      x ≠ y
    moreover have
      ∀ x ∈ A. ∀ y ∈ A. ∀ l ∈ permutations-of-set A.
            x ≠ y ∧ (¬ y ≲_l x) ⟶ x ≲_l y
      using is-less-preferred-than-l.simps
      unfolding permutations-of-set-def
      by auto
    hence ∀ x ∈ A. ∀ y ∈ A. ∀ l ∈ pl-α ` permutations-of-set A.
            x ≠ y ∧ (y, x) ∉ l ⟶ (x, y) ∈ l
      using is-less-preferred-than-l.simps
      unfolding permutations-of-set-def
      unfolding pl-α-def permutations-of-set-def
      by blast
    ultimately show (x, y) ∈ r ∨ (y, x) ∈ r
      using assms
      by metis
  qed
qed

lemma lin-order-pl-α:
  fixes
    r :: 'a rel and
    A :: 'a set
  assumes
    lin-order: linear-order-on A r and
    fin: finite A
  shows r ∈ pl-α ‘ permutations-of-set A
proof −
  let ?φ = λ a. card ((underS r a) ∩ A)
  let ?inv = the-inv-into A ?φ
  let ?l = map (λ x. ?inv x) (rev [0 ..< card A])
  have antisym:
    ∀ a ∈ A. ∀ b ∈ A.
        a ∈ (underS r b) ∧ b ∈ (underS r a) ⟶ False
    using lin-order
    unfolding underS-def linear-order-on-def partial-order-on-def antisym-def
    by blast
  hence ∀ a ∈ A. ∀ b ∈ A. ∀ c ∈ A.
          a ∈ (underS r b) ⟶ b ∈ (underS r c) ⟶ a ∈ (underS r c)
    using lin-order CollectD CollectI transD
    unfolding underS-def linear-order-on-def
            partial-order-on-def preorder-on-def
    by (metis (mono-tags, lifting))
  hence a-lt-b-imp:
    ∀ a ∈ A. ∀ b ∈ A. a ∈ (underS r b) ⟶ (underS r a) ⊂ (underS r b)
    using preorder-on-def partial-order-on-def linear-order-on-def
        antisym lin-order psubsetI underS-E underS-incr
    by metis
  hence mon: ∀ a ∈ A. ∀ b ∈ A. a ∈ (underS r b) ⟶ ?φ a < ?φ b
      using Int-iff Int-mono a-lt-b-imp card-mono card-subset-eq
          fin finite-Int order-le-imp-less-or-eq underS-E
          subset-iff-psubset-eq
      by metis
  moreover have total-underS:
    ∀ a ∈ A. ∀ b ∈ A. a ≠ b ⟶ a ∈ (underS r b) ∨ b ∈ (underS r a)
    using lin-order totalp-onD totalp-on-total-on-eq
    unfolding underS-def linear-order-on-def partial-order-on-def antisym-def
    by fastforce
  ultimately have ∀ a ∈ A. ∀ b ∈ A. a ≠ b ⟶ ?φ a ≠ ?φ b
    using order-less-imp-not-eq2
    by metis
  hence inj: inj-on ?φ A
    using inj-on-def
    by blast
  have in-bounds: ∀ a ∈ A. ?φ a < card A
    using CollectD IntD1 card-seteq fin inf-le2 linorder-le-less-linear
    unfolding underS-def
    by (metis (mono-tags, lifting))
  hence ?φ ‘ A ⊆ {0 ..< card A}
    using atLeast0LessThan
    by blast
  moreover have card (?φ ‘ A) = card A
    using inj fin card-image
    by blast
  ultimately have ?φ ‘ A = {0 ..< card A}
  by (simp add: card-subset-eq)
hence bij-A: bij-betw ?φ A {0 ..< card A}
  using inj
  unfolding bij-betw-def
  by safe
hence bij-inv: bij-betw ?inv {0 ..< card A} A
  using bij-betw-the-inv-into
  by metis
hence ?inv ‘ {0 ..< card A} = A
  unfolding bij-betw-def
  by metis
hence set-eq-A: set ?l = A
  by simp
moreover have dist-l: distinct ?l
  using bij-inv
  unfolding distinct-map
  using bij-betw-imp-inj-on
  by simp
ultimately have ?l ∈ permutations-of-set A
  by auto
moreover have index-eq: ∀ a ∈ A. index ?l a = card A − 1 − ?φ a
proof
  fix a :: 'a
  assume a-in-A: a ∈ A
  have ∀ l. ∀ i < length l. (rev l)!i = l!(length l − 1 − i)
    using rev-nth
    by auto
  hence ∀ i < length [0 ..< card A]. (rev [0 ..< card A])!i =
            [0 ..< card A]!(length [0 ..< card A] − 1 − i)
    by blast
  moreover have ∀ i < card A. [0 ..< card A]!i = i
    by simp
  moreover have card-A-len: length [0 ..< card A] = card A
    by simp
  ultimately have ∀ i < card A. (rev [0 ..< card A])!i = card A − 1 − i
    using diff-Suc-eq-diff-pred diff-less diff-self-eq-0
          less-imp-diff-less zero-less-Suc
    by metis
  moreover have ∀ i < card A. ?l!i = ?inv ((rev [0 ..< card A])!i)
    by simp
  ultimately have ∀ i < card A. ?l!i = ?inv (card A − 1 − i)
    by presburger
  moreover have
    card A − 1 − (card A − 1 − card (underS r a ∩ A)) =
        card (underS r a ∩ A)
    using in-bounds a-in-A
    by auto
  moreover have ?inv (card (underS r a ∩ A)) = a
    using a-in-A inj the-inv-into-f-f
    by fastforce
  ultimately have ?l!(card A − 1 − card (underS r a ∩ A)) = a
    using in-bounds a-in-A card-Diff-singleton
          card-Suc-Diff1 diff-less-Suc fin
    by metis
  thus index ?l a = card A − 1 − card (underS r a ∩ A)
    using bij-inv dist-l a-in-A card-A-len card-Diff-singleton card-Suc-Diff1
          diff-less-Suc fin index-nth-id length-map length-rev
    by metis
qed
moreover have pl-α ?l = r
proof (intro equalityI, unfold pl-α-def is-less-preferred-than-l.simps, safe)
  fix a b :: 'a
  assume
    in-bounds-a: a ∈ set ?l and
    in-bounds-b: b ∈ set ?l
  moreover have element-a: ?inv (index ?l a) ∈ A
    using bij-inv in-bounds-a atLeast0LessThan set-eq-A bij-inv
          cancel-comm-monoid-add-class.diff-cancel diff-Suc-eq-diff-pred
          diff-less in-bounds index-eq lessThan-iff less-imp-diff-less
          zero-less-Suc inj dist-l image-eqI image-eqI length-upt
    unfolding bij-betw-def
    by (metis (no-types, lifting))
  moreover have el-b: ?inv (index ?l b) ∈ A
    using bij-inv in-bounds-b atLeast0LessThan set-eq-A bij-inv
          cancel-comm-monoid-add-class.diff-cancel diff-Suc-eq-diff-pred
          diff-less in-bounds index-eq lessThan-iff less-imp-diff-less
          zero-less-Suc inj dist-l image-eqI image-eqI length-upt
    unfolding bij-betw-def
    by (metis (no-types, lifting))
  moreover assume index ?l b ≤ index ?l a
  ultimately have card A − 1 − (?φ b) ≤ card A − 1 − (?φ a)
    using index-eq set-eq-A
    by metis
  moreover have ∀ a < card A. ?φ (?inv a) < card A
    using fin bij-inv bij-A
    unfolding bij-betw-def
    by fastforce
  hence ?φ b ≤ card A − 1 ∧ ?φ a ≤ card A − 1
    using in-bounds-a in-bounds-b fin
    by fastforce
  ultimately have ?φ b ≥ ?φ a
    using fin le-diff-iff'
    by blast
  hence ?φ a < ?φ b ∨ ?φ a = ?φ b
    by auto
  moreover have
    ∀ a ∈ A. ∀ b ∈ A. ?φ a < ?φ b ⟶ a ∈ underS r b
    using mon total-underS antisym order-less-not-sym
```

```
      by metis
    hence ?φ a < ?φ b ⟶ a ∈ underS r b
      using element-a el-b in-bounds-a in-bounds-b set-eq-A
      by blast
    hence ?φ a < ?φ b ⟶ (a, b) ∈ r
      unfolding underS-def
      by simp
    moreover have ∀ a ∈ A. ∀ b ∈ A. ?φ a = ?φ b ⟶ a = b
      using mon total-underS antisym order-less-not-sym
      by metis
    hence ?φ a = ?φ b ⟶ a = b
      using element-a el-b in-bounds-a in-bounds-b set-eq-A
      by blast
    hence ?φ a = ?φ b ⟶ (a, b) ∈ r
      using lin-order element-a el-b in-bounds-a
            in-bounds-b set-eq-A
      unfolding linear-order-on-def partial-order-on-def
                preorder-on-def refl-on-def
      by auto
    ultimately show (a, b) ∈ r
      by auto
  next
    fix a b :: 'a
    assume a-b-rel: (a, b) ∈ r
    hence
      a-in-A: a ∈ A and
      b-in-A: b ∈ A and
      a-under-b-or-eq: a ∈ underS r b ∨ a = b
      using lin-order
      unfolding linear-order-on-def partial-order-on-def
                preorder-on-def refl-on-def underS-def
      by auto
    thus
      a ∈ set ?l and
      b ∈ set ?l
      using bij-inv set-eq-A
      by (metis, metis)
    hence ?φ a ≤ ?φ b
      using mon le-eq-less-or-eq a-under-b-or-eq
            a-in-A b-in-A
      by auto
    thus index ?l b ≤ index ?l a
      using index-eq a-in-A b-in-A diff-le-mono2
      by metis
  qed
  ultimately show r ∈ pl-α ‘ permutations-of-set A
    by auto
qed
```

```
lemma index-helper:
  fixes
    l :: 'x list and
    x :: 'x
  assumes
    finite (set l) and
    distinct l and
    x ∈ set l
  shows index l x = card {y ∈ set l. index l y < index l x}
proof −
  have bij-l: bij-betw (index l) (set l) {0 ..< length l}
    using assms bij-betw-index
    by blast
  hence card {y ∈ set l. index l y < index l x} =
          card (index l ‘ {y ∈ set l. index l y < index l x})
    using CollectD bij-betw-same-card bij-betw-subset subsetI
    by (metis (no-types, lifting))
  also have index l ‘ {y ∈ set l. index l y < index l x} =
      {m | m. m ∈ index l ‘ (set l) ∧ m < index l x}
    by blast
  also have
    {m | m. m ∈ index l ‘ (set l) ∧ m < index l x} =
      {m | m. m < index l x}
    using bij-l assms atLeastLessThan-iff bot-nat-0.extremum
        index-image index-less-size-conv order-less-trans
    by metis
  also have card {m | m. m < index l x} = index l x
    by simp
  finally show ?thesis
    by simp
qed

lemma pl-α-eq-imp-list-eq:
  fixes l l' :: 'x list
  assumes
    fin-set-l: finite (set l) and
    set-eq: set l = set l' and
    dist-l: distinct l and
    dist-l': distinct l' and
    pl-α-eq: pl-α l = pl-α l'
  shows l = l'
proof (rule ccontr)
  assume l ≠ l'
  moreover with set-eq
  have l ≠ [] ∧ l' ≠ []
    by auto
  ultimately obtain
    i :: nat and
    x :: 'x where
```

```
      i < length l and
      l!i ≠ l'!i and
      x = l!i and
    x-in-l: x ∈ set l
    using dist-l dist-l' distinct-remdups-id
          length-remdups-card-conv nth-equalityI
          nth-mem set-eq
    by metis
  moreover with set-eq
    have neq-ind: index l x ≠ index l' x
    using dist-l index-nth-id nth-index
    by metis
  ultimately have
    card {y ∈ set l. index l y < index l x} ≠
      card {y ∈ set l. index l' y < index l' x}
    using dist-l dist-l' set-eq index-helper fin-set-l
    by (metis (mono-tags))
  then obtain y :: 'x where
    y-in-set-l: y ∈ set l and
    y-neq-x: y ≠ x and
    neq-indices:
      (index l y < index l x ∧ index l' y > index l' x)
      ∨ (index l' y < index l' x ∧ index l y > index l x)
    using index-eq-index-conv not-less-iff-gr-or-eq set-eq
    by (metis (mono-tags, lifting))
  hence
    (is-less-preferred-than-l x l y ∧ is-less-preferred-than-l y l' x)
    ∨ (is-less-preferred-than-l x l' y ∧ is-less-preferred-than-l y l x)
    unfolding is-less-preferred-than-l.simps
    using y-in-set-l less-imp-le-nat set-eq x-in-l
    by blast
  hence ((x, y) ∈ pl-α l ∧ (x, y) ∉ pl-α l')
        ∨ ((x, y) ∈ pl-α l' ∧ (x, y) ∉ pl-α l)
    unfolding pl-α-def
    using is-less-preferred-than-l.simps y-neq-x neq-indices
          case-prod-conv linorder-not-less mem-Collect-eq
    by metis
  thus False
    using pl-α-eq
    by blast
qed

lemma pl-α-bij-betw:
  fixes X :: 'x set
  assumes finite X
  shows bij-betw pl-α (permutations-of-set X) {r. linear-order-on X r}
proof (unfold bij-betw-def, safe)
  show inj-on pl-α (permutations-of-set X)
    unfolding inj-on-def permutations-of-set-def
    using pl-α-eq-imp-list-eq assms
    by fastforce
next
  fix l :: 'x list
  assume l ∈ permutations-of-set X
  thus linear-order-on X (pl-α l)
    using assms pl-α-lin-order
    by blast
next
  fix r :: 'x rel
  assume linear-order-on X r
  thus r ∈ pl-α ‘ permutations-of-set X
    using assms lin-order-pl-α
    by blast
qed
```

2.1.5 Limited Preference

```
definition limited :: 'a set ⇒ 'a Preference-List ⇒ bool where
  limited A r ≡ ∀ a. a ∈ set r ⟶ a ∈ A

fun limit-l :: 'a set ⇒ 'a Preference-List ⇒ 'a Preference-List where
  limit-l A l = List.filter (λ a. a ∈ A) l

lemma limited-dest:
  fixes
    A :: 'a set and
    l :: 'a Preference-List and
    a b :: 'a
  assumes
    a ≲l b and
    limited A l
  shows a ∈ A ∧ b ∈ A
  using assms
  unfolding limited-def
  by simp

lemma limit-equiv:
  fixes
    A :: 'a set and
    l :: 'a list
  assumes well-formed-l l
  shows pl-α (limit-l A l) = limit A (pl-α l)
  using assms
proof (induction l)
  case Nil
  show pl-α (limit-l A []) = limit A (pl-α [])
    unfolding pl-α-def
    by simp
```

```
next
  case (Cons a l)
  fix
    a :: 'a and
    l :: 'a list
  assume
    wf-imp-limit: well-formed-l l ⟹ pl-α (limit-l A l) = limit A (pl-α l) and
    wf-a-l: well-formed-l (a#l)
  show pl-α (limit-l A (a#l)) = limit A (pl-α (a#l))
  proof (unfold limit-l.simps limit.simps, intro equalityI, safe)
    fix b c :: 'a
    assume b-less-c: (b, c) ∈ pl-α (filter (λ a. a ∈ A) (a#l))
    moreover have limit-preference-list-assoc:
      pl-α (limit-l A l) = limit A (pl-α l)
      using wf-a-l wf-imp-limit
      by simp
    ultimately have
      b ∈ set (a#l) and
      c ∈ set (a#l)
      using case-prodD filter-set mem-Collect-eq member-filter
            is-less-preferred-than-l.simps
      unfolding pl-α-def
      by (metis, metis)
    thus (b, c) ∈ pl-α (a#l)
    proof (unfold pl-α-def is-less-preferred-than-l.simps, safe)
      have idx-set-eq:
        ∀ a' l' a''. (a' :: 'a) ≲_l' a'' =
            (a' ∈ set l' ∧ a'' ∈ set l' ∧ index l' a'' ≤ index l' a')
        using is-less-preferred-than-l.simps
        by blast
      moreover from this
      have {(a', b'). a' ≲_(limit-l A l) b'} =
        {(a', a''). a' ∈ set (limit-l A l) ∧ a'' ∈ set (limit-l A l) ∧
            index (limit-l A l) a'' ≤ index (limit-l A l) a'}
        by presburger
      moreover from this
      have {(a', b'). a' ≲_l b'} =
        {(a', a''). a' ∈ set l ∧ a'' ∈ set l ∧ index l a'' ≤ index l a'}
        using is-less-preferred-than-l.simps
        by auto
      ultimately have {(a', b').
              a' ∈ set (limit-l A l) ∧ b' ∈ set (limit-l A l)
              ∧ index (limit-l A l) b' ≤ index (limit-l A l) a'} =
                  limit A {(a', b'). a' ∈ set l
              ∧ b' ∈ set l ∧ index l b' ≤ index l a'}
        using pl-α-def limit-preference-list-assoc
        by (metis (no-types))
      hence idx-imp:
        b ∈ set (limit-l A l) ∧ c ∈ set (limit-l A l)
      ∧ index (limit-l A l) c ≤ index (limit-l A l) b
  ⟶ b ∈ set l ∧ c ∈ set l ∧ index l c ≤ index l b
  by auto
have b ≲(filter (λ a. a ∈ A) (a#l)) c
  using b-less-c case-prodD mem-Collect-eq
  unfolding pl-α-def
  by (metis (no-types))
moreover obtain
  f h :: 'a ⇒ 'a list ⇒ 'a ⇒ 'a and
  g :: 'a ⇒ 'a list ⇒ 'a ⇒ 'a list where
  ∀ d s e. d ≲s e ⟶
    d = f e s d ∧ s = g e s d ∧ e = h e s d
    ∧ f e s d ∈ set (g e s d) ∧ h e s d ∈ set (g e s d)
    ∧ index (g e s d) (h e s d) ≤ index (g e s d) (f e s d)
  by fastforce
ultimately have
  b = f c (filter (λ a. a ∈ A) (a#l)) b
    ∧ filter (λ a. a ∈ A) (a#l) =
        g c (filter (λ a. a ∈ A) (a#l)) b
    ∧ c = h c (filter (λ a. a ∈ A) (a#l)) b
    ∧ f c (filter (λ a. a ∈ A) (a#l)) b
        ∈ set (g c (filter (λ a. a ∈ A) (a#l)) b)
    ∧ h c (filter (λ a. a ∈ A) (a#l)) b
        ∈ set (g c (filter (λ a. a ∈ A) (a#l)) b)
    ∧ index (g c (filter (λ a. a ∈ A) (a#l)) b)
        (h c (filter (λ a. a ∈ A) (a#l)) b)
      ≤ index (g c (filter (λ a. a ∈ A) (a#l)) b)
          (f c (filter (λ a. a ∈ A) (a#l)) b)
  by blast
moreover have filter (λ a. a ∈ A) l = limit-l A l
  by simp
moreover have
  index (limit-l A l) c ≠
    index (g c (filter (λ a. a ∈ A) (a#l)) b)
        (h c (filter (λ a. a ∈ A) (a # l)) b)
  ∨ index (limit-l A l) b ≠
    index (g c (filter (λ a. a ∈ A) (a#l)) b)
        (f c (filter (λ a. a ∈ A) (a#l)) b)
  ∨ index (limit-l A l) c ≤ index (limit-l A l) b
  ∨ ¬ index (g c (filter (λ a. a ∈ A) (a # l)) b)
    (h c (filter (λ a. a ∈ A) (a#l)) b)
      ≤ index (g c (filter (λ a. a ∈ A) (a#l)) b)
          (f c (filter (λ a. a ∈ A) (a#l)) b)
  by presburger
ultimately have a ≠ c ⟶ index (a#l) c ≤ index (a#l) b
    using add-le-cancel-right idx-imp index-Cons le-zero-eq
          nth-index set-ConsD wf-a-l
    unfolding filter.simps is-less-preferred-than-l.elims
              distinct.simps
```

```
     by metis
   thus index (a#l) c ≤ index (a#l) b
    by force
  qed
  show
   b ∈ A and
   c ∈ A
   using b-less-c case-prodD mem-Collect-eq set-filter
   unfolding pl-α-def is-less-preferred-than-l.simps
   by (metis (no-types, lifting),
     metis (no-types, lifting))
 next
  fix b c :: 'a
  assume
   b-less-c: (b, c) ∈ pl-α (a#l) and
   b-in-A: b ∈ A and
   c-in-A: c ∈ A
  have (b, c) ∈ pl-α (a#l)
   by (simp add: b-less-c)
  hence b ≲(a#l) c
   using case-prodD mem-Collect-eq
   unfolding pl-α-def
   by metis
  moreover have
   pl-α (filter (λ a. a ∈ A) l) =
     {(a, b). (a, b) ∈ pl-α l ∧ a ∈ A ∧ b ∈ A}
   using wf-a-l wf-imp-limit
   by simp
  ultimately have
   index (filter (λ a. a ∈ A) (a#l)) c
     ≤ index (filter (λ a. a ∈ A) (a#l)) b
   unfolding pl-α-def
   using add-leE add-le-cancel-right case-prodI c-in-A
      b-in-A index-Cons set-ConsD not-one-le-zero
      in-rel-Collect-case-prod-eq mem-Collect-eq
      linorder-le-cases
   by fastforce
  moreover have
   b ∈ set (filter (λ a. a ∈ A) (a#l)) and
   c ∈ set (filter (λ a. a ∈ A) (a#l))
   using b-less-c b-in-A c-in-A
   unfolding pl-α-def
   by (fastforce, fastforce)
  ultimately show (b, c) ∈ pl-α (filter (λ a. a ∈ A) (a#l))
   unfolding pl-α-def
   by simp
 qed
qed
```

2.1.6 Auxiliary Definitions

definition *total-on-l* :: $'a\ set \Rightarrow 'a\ Preference\text{-}List \Rightarrow bool$ **where**
$total\text{-}on\text{-}l\ A\ l \equiv \forall\ a \in A.\ a \in set\ l$

definition *refl-on-l* :: $'a\ set \Rightarrow 'a\ Preference\text{-}List \Rightarrow bool$ **where**
$refl\text{-}on\text{-}l\ A\ l \equiv (\forall\ a.\ a \in set\ l \longrightarrow a \in A) \land (\forall\ a \in A.\ a \lesssim_l a)$

definition *trans* :: $'a\ Preference\text{-}List \Rightarrow bool$ **where**
$trans\ l \equiv \forall\ (a,\ b,\ c) \in set\ l \times set\ l \times set\ l.\ a \lesssim_l b \land b \lesssim_l c \longrightarrow a \lesssim_l c$

definition *preorder-on-l* :: $'a\ set \Rightarrow 'a\ Preference\text{-}List \Rightarrow bool$ **where**
$preorder\text{-}on\text{-}l\ A\ l \equiv refl\text{-}on\text{-}l\ A\ l \land trans\ l$

definition *antisym-l* :: $'a\ list \Rightarrow bool$ **where**
$antisym\text{-}l\ l \equiv \forall\ a\ b.\ a \lesssim_l b \land b \lesssim_l a \longrightarrow a = b$

definition *partial-order-on-l* :: $'a\ set \Rightarrow 'a\ Preference\text{-}List \Rightarrow bool$ **where**
$partial\text{-}order\text{-}on\text{-}l\ A\ l \equiv preorder\text{-}on\text{-}l\ A\ l \land antisym\text{-}l\ l$

definition *linear-order-on-l* :: $'a\ set \Rightarrow 'a\ Preference\text{-}List \Rightarrow bool$ **where**
$linear\text{-}order\text{-}on\text{-}l\ A\ l \equiv partial\text{-}order\text{-}on\text{-}l\ A\ l \land total\text{-}on\text{-}l\ A\ l$

definition *connex-l* :: $'a\ set \Rightarrow 'a\ Preference\text{-}List \Rightarrow bool$ **where**
$connex\text{-}l\ A\ l \equiv limited\ A\ l \land (\forall\ a \in A.\ \forall\ b \in A.\ a \lesssim_l b \lor b \lesssim_l a)$

abbreviation *ballot-on* :: $'a\ set \Rightarrow 'a\ Preference\text{-}List \Rightarrow bool$ **where**
$ballot\text{-}on\ A\ l \equiv well\text{-}formed\text{-}l\ l \land linear\text{-}order\text{-}on\text{-}l\ A\ l$

2.1.7 Auxiliary Lemmas

lemma *list-trans*[*simp*]:
fixes l :: *'a Preference-List*
shows *trans l*
unfolding *trans-def*
by *simp*

lemma *list-antisym*[*simp*]:
fixes l :: *'a Preference-List*
shows *antisym-l l*
unfolding *antisym-l-def*
by *auto*

lemma *lin-order-equiv-list-of-alts*:
fixes
A :: *'a set* **and**
l :: *'a Preference-List*
shows $linear\text{-}order\text{-}on\text{-}l\ A\ l = (A = set\ l)$
unfolding *linear-order-on-l-def total-on-l-def*
partial-order-on-l-def preorder-on-l-def

```
          refl-on-l-def
  by auto

lemma connex-imp-refl:
  fixes
    A :: 'a set and
    l :: 'a Preference-List
  assumes connex-l A l
  shows refl-on-l A l
  unfolding refl-on-l-def
  using assms connex-l-def Preference-List.limited-def
  by metis

lemma lin-ord-imp-connex-l:
  fixes
    A :: 'a set and
    l :: 'a Preference-List
  assumes linear-order-on-l A l
  shows connex-l A l
  using assms linorder-le-cases
  unfolding connex-l-def linear-order-on-l-def preorder-on-l-def
          limited-def refl-on-l-def partial-order-on-l-def
          is-less-preferred-than-l.simps
  by metis

lemma above-trans:
  fixes
    l :: 'a Preference-List and
    a b :: 'a
  assumes
    trans l and
    a ≲_l b
  shows set (above-l l b) ⊆ set (above-l l a)
  using assms set-take-subset-set-take rank-l.simps
       Suc-le-mono add.commute add-0 add-Suc
  unfolding Preference-List.is-less-preferred-than-l.simps
          above-l-def One-nat-def
  by metis

lemma less-preferred-l-rel-equiv:
  fixes
    l :: 'a Preference-List and
    a b :: 'a
  shows a ≲_l b =
    Preference-Relation.is-less-preferred-than a (pl-α l) b
  unfolding pl-α-def
  by simp

theorem above-equiv:
  fixes
    l :: 'a Preference-List and
    a :: 'a
  shows set (above-l l a) = above (pl-α l) a
proof (safe)
  fix b :: 'a
  assume b-member: b ∈ set (above-l l a)
  hence index l b ≤ index l a
    unfolding rank-l.simps above-l-def
    using Suc-eq-plus1 Suc-le-eq index-take linorder-not-less
          bot-nat-0.extremum-strict
    by (metis (full-types))
  hence a ≲_l b
    using Suc-le-mono add-Suc le-antisym take-0 b-member
          in-set-takeD index-take le0 rank-l.simps
    unfolding above-l-def is-less-preferred-than-l.simps
    by metis
  thus b ∈ above (pl-α l) a
    using less-preferred-l-rel-equiv pref-imp-in-above
    by metis
next
  fix b :: 'a
  assume b ∈ above (pl-α l) a
  hence a ≲_l b
    using pref-imp-in-above less-preferred-l-rel-equiv
    by metis
  thus b ∈ set (above-l l a)
    unfolding above-l-def is-less-preferred-than-l.simps
              rank-l.simps
    using Suc-eq-plus1 Suc-le-eq index-less-size-conv
          set-take-if-index le-imp-less-Suc
    by (metis (full-types))
qed

theorem rank-equiv:
  fixes
    l :: 'a Preference-List and
    a :: 'a
  assumes well-formed-l l
  shows rank-l l a = rank (pl-α l) a
proof (unfold rank-l.simps rank.simps, cases a ∈ set l)
  case True
  moreover have above (pl-α l) a = set (above-l l a)
    unfolding above-equiv
    by simp
  moreover have distinct (above-l l a)
    unfolding above-l-def
    using assms distinct-take
    by blast
  moreover from this
  have card (set (above-l l a)) = length (above-l l a)
    using distinct-card
    by blast
  moreover have length (above-l l a) = rank-l l a
    unfolding above-l-def
    using Suc-le-eq
    by (simp add: in-set-member)
  ultimately show
    (if a ∈ set l then index l a + 1 else 0) =
        card (above (pl-α l) a)
    by simp
next
  case False
  hence above (pl-α l) a = {}
    unfolding above-def
    using less-preferred-l-rel-equiv
    by fastforce
  thus (if a ∈ set l then index l a + 1 else 0) =
          card (above (pl-α l) a)
    using False
    by fastforce
qed

lemma lin-ord-equiv:
  fixes
    A :: 'a set and
    l :: 'a Preference-List
  shows linear-order-on-l A l = linear-order-on A (pl-α l)
  unfolding is-less-preferred-than-l.simps antisym-def total-on-def
            pl-α-def linear-order-on-l-def linear-order-on-def
            refl-on-l-def Relation.trans-def preorder-on-l-def
            partial-order-on-l-def partial-order-on-def
            total-on-l-def preorder-on-def refl-on-def
  by auto
```

2.1.8 First Occurrence Indices

```
lemma pos-in-list-yields-rank:
  fixes
    l :: 'a Preference-List and
    a :: 'a and
    n :: nat
  assumes
    ∀ (j :: nat) ≤ n. l!j ≠ a and
    l!(n − 1) = a
  shows rank-l l a = n
  using assms
proof (induction l arbitrary: n)
```

```
  case Nil
  thus ?case
    by simp
next
  fix
    l :: 'a Preference-List and
    a :: 'a
  case (Cons a l)
  thus ?case
    by simp
qed

lemma ranked-alt-not-at-pos-before:
  fixes
    l :: 'a Preference-List and
    a :: 'a and
    n :: nat
  assumes
    a ∈ set l and
    n < (rank-l l a) − 1
  shows l!n ≠ a
  using index-first member-def rank-l.simps
        assms add-diff-cancel-right'
  by metis

lemma pos-in-list-yields-pos:
  fixes
    l :: 'a Preference-List and
    a :: 'a
  assumes a ∈ set l
  shows l!(rank-l l a − 1) = a
  using assms
proof (induction l)
  case Nil
  thus ?case
    by simp
next
  fix
    l :: 'a Preference-List and
    b :: 'a
  case (Cons b l)
  assume a ∈ set (b#l)
  moreover from this
  have rank-l (b#l) a = 1 + index (b#l) a
    using Suc-eq-plus1 add-Suc add-cancel-left-left
          rank-l.simps
    by metis
  ultimately show (b#l)!(rank-l (b#l) a − 1) = a
    using diff-add-inverse nth-index
    by metis
qed


lemma rel-of-pref-pred-for-set-eq-list-to-rel:
  fixes l :: 'a Preference-List
  shows relation-of (λ y z. y ≲_l z) (set l) = pl-α l
proof (unfold relation-of-def, safe)
  fix a b :: 'a
  assume a ≲_l b
  moreover have (a ≲_l b) = (a ⪯_(pl-α l) b)
    using less-preferred-l-rel-equiv
    by (metis (no-types))
  ultimately show (a, b) ∈ pl-α l
    by simp
next
  fix a b :: 'a
  assume (a, b) ∈ pl-α l
  thus a ≲_l b
    using less-preferred-l-rel-equiv
    unfolding is-less-preferred-than.simps
    by metis
  thus
    a ∈ set l and
    b ∈ set l
    by (simp, simp)
qed

end
```

2.2 Preference (List) Profile

```
theory Profile-List
  imports ../Profile
          Preference-List
begin
```

2.2.1 Definition

A profile (list) contains one ballot for each voter.

```
type-synonym 'a Profile-List = 'a Preference-List list

type-synonym 'a Election-List = 'a set × 'a Profile-List
```

Abstraction from profile list to profile.

```
fun pl-to-pr-α :: 'a Profile-List ⇒ ('a, nat) Profile where
```

```
  pl-to-pr-α pl = (λ n. if n < length pl ∧ n ≥ 0
                        then (map pl-α pl)!n
                        else {})

lemma prof-abstr-presv-size:
  fixes p :: 'a Profile-List
  shows length p = length (to-list {0 ..< length p} (pl-to-pr-α p))
  by simp
```

2.2.2 Refinement Proof

A profile on a finite set of alternatives A contains only ballots that are lists of linear orders on A.

```
definition profile-l :: 'a set ⇒ 'a Profile-List ⇒ bool where
  profile-l A p ≡ ∀ i < length p. ballot-on A (p!i)

lemma profile-list-refines-profile:
  fixes
    A :: 'a set and
    p :: 'a Profile-List
  assumes profile-l A p
  shows profile {0 ..< length p} A (pl-to-pr-α p)
proof (unfold profile-def, safe)
  fix i :: nat
  assume in-range: i ∈ {0 ..< length p}
  moreover have well-formed-l (p!i)
    using assms in-range
    unfolding profile-l-def
    by simp
  moreover have linear-order-on-l A (p!i)
    using assms in-range
    unfolding profile-l-def
    by simp
  ultimately show linear-order-on A (pl-to-pr-α p i)
    using lin-ord-equiv length-map nth-map
    by auto
qed

end
```

2.3 Ordered Relation Type

```
theory Ordered-Relation
  imports Preference-Relation
          ./Refined-Types/Preference-List
          HOL−Combinatorics.Multiset-Permutations
```

```
begin

lemma fin-ordered:
  fixes X :: 'x set
  assumes finite X
  obtains ord :: 'x rel where
    linear-order-on X ord
proof −
  obtain l :: 'x list where
    set-l: set l = X
    using finite-list assms
    by blast
  let ?r = pl-α l
  have antisym ?r
    using set-l Collect-mono-iff antisym index-eq-index-conv pl-α-def
    unfolding antisym-def
    by fastforce
  moreover have refl-on X ?r
    using set-l
    unfolding refl-on-def pl-α-def is-less-preferred-than-l.simps
    by blast
  moreover have Relation.trans ?r
    unfolding Relation.trans-def pl-α-def is-less-preferred-than-l.simps
    by auto
  moreover have total-on X ?r
    using set-l
    unfolding total-on-def pl-α-def is-less-preferred-than-l.simps
    by force
  ultimately have linear-order-on X ?r
    unfolding linear-order-on-def preorder-on-def partial-order-on-def
    by blast
  moreover assume
    ⋀ ord. linear-order-on X ord ⟹ ?thesis
  ultimately show ?thesis
    by blast
qed

typedef 'a Ordered-Preference =
  {p :: 'a :: finite Preference-Relation. linear-order-on (UNIV :: 'a set) p}
  morphisms ord2pref pref2ord
proof (unfold mem-Collect-eq)
  have finite (UNIV :: 'a set)
    by simp
  then obtain p :: 'a Preference-Relation where
    linear-order-on (UNIV :: 'a set) p
    using fin-ordered
    by metis
  thus ∃ p :: 'a Preference-Relation. linear-order p
    by blast
```

```
qed

instance Ordered-Preference :: (finite) finite
proof
  have (UNIV :: 'a Ordered-Preference set) =
          pref2ord ‘ {p :: 'a Preference-Relation.
              linear-order-on (UNIV :: 'a set) p}
    using type-definition.Abs-image
          type-definition-Ordered-Preference
    by blast
  moreover have
    finite {p :: 'a Preference-Relation.
        linear-order-on (UNIV :: 'a set) p}
    by simp
  ultimately show
    finite (UNIV :: 'a Ordered-Preference set)
    using finite-imageI
    by metis
qed

lemma range-ord2pref: range ord2pref = {p. linear-order p}
  using type-definition-Ordered-Preference type-definition.Rep-range
  by metis

lemma card-ord-pref: card (UNIV :: 'a :: finite Ordered-Preference set) =
                        fact (card (UNIV :: 'a set))
proof −
  let ?n = card (UNIV :: 'a set) and
      ?perm = permutations-of-set (UNIV :: 'a set)
  have (UNIV :: 'a Ordered-Preference set) =
    pref2ord ‘ {p :: 'a Preference-Relation.
                linear-order-on (UNIV :: 'a set) p}
    using type-definition-Ordered-Preference type-definition.Abs-image
    by blast
  moreover have
    inj-on pref2ord {p :: 'a Preference-Relation.
        linear-order-on (UNIV :: 'a set) p}
    using inj-onCI pref2ord-inject
    by metis
  ultimately have
    bij-betw pref2ord
      {p :: 'a Preference-Relation.
        linear-order-on (UNIV :: 'a set) p}
          (UNIV :: 'a Ordered-Preference set)
    using bij-betw-imageI
    by metis
  hence card (UNIV :: 'a Ordered-Preference set) =
    card {p :: 'a Preference-Relation.
            linear-order-on (UNIV :: 'a set) p}
    using bij-betw-same-card
    by metis
  moreover have card ?perm = fact ?n
    by simp
  ultimately show ?thesis
    using bij-betw-same-card pl-α-bij-betw finite
    by metis
qed

end
```

2.4 Alternative Election Type

```
theory Quotient-Type-Election
  imports Profile
begin

lemma election-equality-equiv:
  election-equality E E and
  election-equality E E' ⟶ election-equality E' E and
  election-equality E E' ⟶ election-equality E' F
      ⟶ election-equality E F
proof (safe)
  have election-equality
    (fst E, fst (snd E), snd (snd E)) (fst E, fst (snd E), snd (snd E))
    unfolding election-equality.simps
    by safe
  thus election-equality E E
    by clarsimp
next
  assume election-equality E E'
  hence election-equality
    (fst E, fst (snd E), snd (snd E)) (fst E', fst (snd E'), snd (snd E'))
    by simp
  hence election-equality
    (fst E', fst (snd E'), snd (snd E')) (fst E, fst (snd E), snd (snd E))
    unfolding election-equality.simps
    by (metis (mono-tags, lifting))
  thus election-equality E' E
    by clarsimp
next
  assume
    election-equality E E' and
    election-equality E' F
  hence
    election-equality
      (fst E, fst (snd E), snd (snd E)) (fst E', fst (snd E'), snd (snd E')) and
    election-equality
      (fst E', fst (snd E'), snd (snd E')) (fst F, fst (snd F), snd (snd F))
    by (simp, simp)
  hence election-equality
    (fst E, fst (snd E), snd (snd E)) (fst F, fst (snd F), snd (snd F))
    unfolding election-equality.simps
    by (metis (no-types, lifting))
  thus election-equality E F
    by clarsimp
qed

quotient-type ('a, 'v) Election_Q =
  'a set × 'v set × ('a, 'v) Profile / election-equality
  unfolding equivp-reflp-symp-transp reflp-def symp-def transp-def
  using election-equality-equiv
  by simp

fun fst_Q :: ('a, 'v) Election_Q ⇒ 'a set where
  fst_Q E = fst (rep-Election_Q E)

fun snd_Q :: ('a, 'v) Election_Q ⇒ 'v set × ('a, 'v) Profile where
  snd_Q E = snd (rep-Election_Q E)

abbreviation alternatives-ℰ_Q :: ('a, 'v) Election_Q ⇒ 'a set where
  alternatives-ℰ_Q E ≡ fst_Q E

abbreviation voters-ℰ_Q :: ('a, 'v) Election_Q ⇒ 'v set where
  voters-ℰ_Q E ≡ fst (snd_Q E)

abbreviation profile-ℰ_Q :: ('a, 'v) Election_Q ⇒ ('a, 'v) Profile where
  profile-ℰ_Q E ≡ snd (snd_Q E)

end
```

Chapter 3

Quotient Rules

3.1 Quotients of Equivalence Relations

```
theory Relation-Quotients
  imports ../Social-Choice-Types/Symmetry-Of-Functions
begin
```

3.1.1 Definitions

```
fun singleton-set :: 'x set ⇒ 'x where
  singleton-set s = (if (card s = 1) then (the-inv (λ x. {x}) s) else undefined)
— This is undefined if card s ≠ 1. Note that "undefined = undefined" is the only
provable equality for undefined.
```

For a given function, we define a function on sets that maps each set to the unique image under f of its elements, if one exists. Otherwise, the result is undefined.

```
fun π𝒬 :: ('x ⇒ 'y) ⇒ ('x set ⇒ 'y) where
  π𝒬 f s = singleton-set (f ` s)
```

For a given function f on sets and a mapping from elements to sets, we define a function on the set element type that maps each element to the image of its corresponding set under f. A natural mapping is from elements to their classes under a relation.

```
fun inv-π𝒬 :: ('x ⇒ 'x set) ⇒ ('x set ⇒ 'y) ⇒ ('x ⇒ 'y) where
  inv-π𝒬 cls f x = f (cls x)

fun relation-class :: 'x rel ⇒ 'x ⇒ 'x set where
  relation-class r x = r `` {x}
```

3.1.2 Well-Definedness

```
lemma singleton-set-undef-if-card-neq-one:
  fixes s :: 'x set
  assumes card s ≠ 1
  shows singleton-set s = undefined
  using assms
  by simp

lemma singleton-set-def-if-card-one:
  fixes s :: 'x set
  assumes card s = 1
  shows ∃! x. x = singleton-set s ∧ {x} = s
  using assms card-1-singletonE inj-def singleton-inject the-inv-f-f
  unfolding singleton-set.simps
  by (metis (mono-tags, lifting))
```

If the given function is invariant under an equivalence relation, the induced function on sets is well-defined for all equivalence classes of that relation.

```
theorem pass-to-quotient:
  fixes
    f :: 'x ⇒ 'y and
    r :: 'x rel and
    s :: 'x set
  assumes
    f respects r and
    equiv s r
  shows ∀ t ∈ s // r. ∀ x ∈ t. π_Q f t = f x
proof (safe)
  fix
    t :: 'x set and
    x :: 'x
  have ∀ y ∈ r``{x}. (x, y) ∈ r
    unfolding Image-def
    by simp
  hence func-eq-x:
    {f y | y. y ∈ r``{x}} = {f x | y. y ∈ r``{x}}
    using assms
    unfolding congruent-def
    by fastforce
  assume
    t ∈ s // r and
    x-in-t: x ∈ t
  moreover from this have r `` {x} ∈ s // r
    using assms quotient-eq-iff equiv-class-eq-iff quotientI
    by metis
  ultimately have r-img-elem-x-eq-t: r `` {x} = t
    using assms quotient-eq-iff Image-singleton-iff
    by metis
  hence {f x | y. y ∈ r``{x}} = {f x}
    using x-in-t
    by blast
  hence f ` t = {f x}
```

```
    using Setcompr-eq-image r-img-elem-x-eq-t func-eq-x
    by metis
  thus π_Q f t = f x
    using singleton-set-def-if-card-one is-singletonI
          is-singleton-altdef the-elem-eq
    unfolding π_Q.simps
    by metis
qed
```

A function on sets induces a function on the element type that is invariant under a given equivalence relation.

```
theorem pass-to-quotient-inv:
  fixes
    f :: 'x set ⇒ 'x and
    r :: 'x rel and
    s :: 'x set
  assumes equiv s r
  defines induced-fun ≡ (inv-π_Q (relation-class r) f)
  shows
    induced-fun respects r and
    ∀ A ∈ s // r. π_Q induced-fun A = f A
proof (safe)
  have ∀ (a, b) ∈ r. relation-class r a = relation-class r b
    using assms equiv-class-eq
    unfolding relation-class.simps
    by fastforce
  hence ∀ (a, b) ∈ r. induced-fun a = induced-fun b
    unfolding induced-fun-def inv-π_Q.simps
    by auto
  thus induced-fun respects r
    unfolding congruent-def
    by metis
  moreover fix A :: 'x set
  assume A ∈ s // r
  moreover with assms
  obtain a :: 'x where
    a ∈ A and
    A-eq-rel-class-r-a: A = relation-class r a
    using equiv-Eps-in proj-Eps
    unfolding proj-def relation-class.simps
    by metis
  ultimately have π_Q induced-fun A = induced-fun a
    using pass-to-quotient assms
    by blast
  thus π_Q induced-fun A = f A
    using A-eq-rel-class-r-a
    unfolding induced-fun-def
    by simp
qed
```

3.1.3 Equivalence Relations

```
lemma restr-equals-restricted-rel:
  fixes
    s t :: 'a set and
    r :: 'a rel
  assumes
    closed-restricted-rel r s t and
    t ⊆ s
  shows restricted-rel r t s = Restr r t
proof (unfold restricted-rel.simps, safe)
  fix a b :: 'a
  assume
    (a, b) ∈ r and
    a ∈ t and
    b ∈ s
  thus b ∈ t
    using assms
    unfolding closed-restricted-rel.simps restricted-rel.simps
    by blast
next
  fix a b :: 'a
  assume b ∈ t
  thus b ∈ s
    using assms
    by blast
qed

lemma equiv-rel-restr:
  fixes
    s t :: 'x set and
    r :: 'x rel
  assumes
    equiv s r and
    t ⊆ s
  shows equiv t (Restr r t)
proof (unfold equiv-def refl-on-def, safe)
  fix x :: 'x
  assume x ∈ t
  thus (x, x) ∈ r
    using assms
    unfolding equiv-def refl-on-def
    by blast
next
  show sym (Restr r t)
    using assms
    unfolding equiv-def sym-def
    by blast
next
  show Relation.trans (Restr r t)
```

```
    using assms
    unfolding equiv-def Relation.trans-def
    by blast
qed

lemma rel-ind-by-group-act-equiv:
  fixes
    m :: 'x monoid and
    s :: 'y set and
    φ :: ('x, 'y) binary-fun
  assumes group-action m s φ
  shows equiv s (action-induced-rel (carrier m) s φ)
proof (unfold equiv-def refl-on-def sym-def Relation.trans-def
              action-induced-rel.simps, safe)
  fix y :: 'y
  assume y ∈ s
  hence φ 1 m y = y
    using assms group-action.id-eq-one restrict-apply'
    by metis
  thus ∃ g ∈ carrier m. φ g y = y
    using assms group.is-monoid group-hom.axioms
    unfolding group-action-def
    by blast
next
  fix
    y :: 'y and
    g :: 'x
  assume
    y-in-s: y ∈ s and
    carrier-g: g ∈ carrier m
  hence y = φ (inv m g) (φ g y)
    using assms
    by (simp add: group-action.orbit-sym-aux)
  thus ∃ h ∈ carrier m. φ h (φ g y) = y
    using assms carrier-g group.inv-closed
          group-action.group-hom
    unfolding group-hom-def
    by metis
next
  fix
    y :: 'y and
    g h :: 'x
  assume
    y-in-s: y ∈ s and
    carrier-g: g ∈ carrier m and
    carrier-h: h ∈ carrier m
  hence φ (h ⊗ m g) y = φ h (φ g y)
    using assms
    by (simp add: group-action.composition-rule)
  thus ∃ f ∈ carrier m. φ f y = φ h (φ g y)
    using assms carrier-g carrier-h group-action.group-hom
          monoid.m-closed
    unfolding group-def group-hom-def
    by metis
next
  fix
    y :: 'y and
    g :: 'x
  assume
    y ∈ s and
    g ∈ carrier m
  thus φ g y ∈ s
    using assms group-action.element-image
    by metis
next
  fix
    y :: 'y and
    g :: 'x
  assume
    y ∈ s and
    g ∈ carrier m
  thus φ g y ∈ s
    using assms group-action.element-image
    by metis
qed

end
```

3.2 Quotients of Election Set Equivalences

```
theory Election-Quotients
  imports Relation-Quotients
          ../Social-Choice-Types/Voting-Symmetry
          ../Social-Choice-Types/Ordered-Relation
          HOL−Analysis.Convex
          HOL−Analysis.Cartesian-Space
begin
```

3.2.1 Auxiliary Lemmas

```
lemma obtain-partition:
  fixes
    A :: 'a set and
    B :: 'b set and
    f :: 'b ⇒ nat
  assumes
```

```
    finite A and
    finite B and
    (∑ x ∈ B. f x) = card A
  shows ∃ 𝒳. A = ⋃ {𝒳 i | i. i ∈ B} ∧ (∀ i ∈ B. card (𝒳 i) = f i) ∧
              (∀ i j. i ≠ j ⟶ i ∈ B ∧ j ∈ B ⟶ 𝒳 i ∩ 𝒳 j = {})
  using assms
proof (induction card B arbitrary: A B)
  case 0
  fix
    A :: 'a set and
    B :: 'b set
  let ?𝒳 = λ y. {}
  assume
    finite B and
    0 = card B
  hence Y-empty: B = {}
    using 0
    by simp
  hence (∑ x ∈ B. f x) = 0
    by simp
  moreover assume
    finite A and
    (∑ x ∈ B. f x) = card A
  ultimately have A = {}
    by simp
  hence A = ⋃ {?𝒳 i | i. i ∈ B}
    by blast
  moreover have ∀ i j. i ≠ j ⟶ i ∈ B ∧ j ∈ B ⟶ ?𝒳 i ∩ ?𝒳 j = {}
    by blast
  ultimately show
    ∃ 𝒳. A = ⋃ {𝒳 i | i. i ∈ B} ∧
              (∀ i ∈ B. card (𝒳 i) = f i) ∧
              (∀ i j. i ≠ j ⟶ i ∈ B ∧ j ∈ B ⟶ 𝒳 i ∩ 𝒳 j = {})
    using Y-empty
    by simp
next
  case (Suc x)
  fix
    x :: nat and
    A :: 'a set and
    B :: 'b set
  assume
    card-B: Suc x = card B and
    fin-B: finite B and
    fin-A: finite A and
    card-A: (∑ x ∈ B. f x) = card A and
    hyp:
      ⋀ Y (X :: 'a set).
        x = card Y ⟹
        finite X ⟹
        finite Y ⟹
        (∑ y ∈ Y. f y) = card X ⟹
        ∃ 𝒳.
         X = ⋃ {𝒳 i | i. i ∈ Y} ∧
              (∀ i ∈ Y. card (𝒳 i) = f i) ∧
              (∀ i j. i ≠ j ⟶ i ∈ Y ∧ j ∈ Y ⟶ 𝒳 i ∩ 𝒳 j = {})
  then obtain
    B′ :: ′b set and
    y :: ′b where
    ins-B: B = insert y B′ and
    card-B′: card B′ = x and
    fin-B′: finite B′ and
    y-not-in-B′: y ∉ B′
    using card-Suc-eq-finite
    by (metis (no-types, lifting))
  hence f y ≤ card A
    using card-A le-add1 n-not-Suc-n sum.insert
    by metis
  then obtain A′ :: ′a set where
    X′-in-X: A′ ⊆ A and
    card-X′: card A′ = f y
    using fin-A ex-card
    by metis
  hence finite (A − A′) ∧ card (A − A′) = (∑ y ∈ B′. f y)
    using card-B card-A fin-A ins-B card-B′ fin-B′ Suc-n-not-n
        add-diff-cancel-left′ card-Diff-subset card-insert-if
        finite-Diff finite-subset sum.insert
    by metis
  then obtain 𝒳 :: ′b ⇒ ′a set where
    part: A − A′ = ⋃ {𝒳 i | i. i ∈ B′} and
    disj: ∀ i j. i ≠ j ⟶ i ∈ B′ ∧ j ∈ B′ ⟶ 𝒳 i ∩ 𝒳 j = {} and
    card: ∀ i ∈ B′. card (𝒳 i) = f i
    using hyp[of B′ A − A′] fin-B′ card-B′
    by auto
  then obtain 𝒳′ :: ′b ⇒ ′a set where
    map′: 𝒳′ = (λ z. if z = y then A′ else 𝒳 z)
    by simp
  hence eq-𝒳: ∀ i ∈ B′. 𝒳′ i = 𝒳 i
    using y-not-in-B′
    by simp
  have B = {y} ∪ B′
    using ins-B
    by simp
  hence ⋃ {𝒳′ i | i. i ∈ B} = 𝒳′ y ∪ ⋃ {𝒳′ i | i. i ∈ B′}
    by blast
  also have ... = A′ ∪ ⋃ {𝒳 i | i. i ∈ B′}
    using map′ eq-𝒳
    by fastforce
    finally have part′: A = ⋃ {𝒳′ i | i. i ∈ B}
      using part Diff-partition X′-in-X
      by metis
    have ∀ i ∈ B′. 𝒳′ i ⊆ A − A′
      using part eq-𝒳 Setcompr-eq-image UN-upper
      by metis
    hence ∀ i ∈ B′. 𝒳′ i ∩ A′ = {}
      by blast
    hence ∀ i ∈ B′. 𝒳′ i ∩ 𝒳′ y = {}
      using map′
      by simp
    hence ∀ i j. i ≠ j ⟶ i ∈ B ∧ j ∈ B ⟶ 𝒳′ i ∩ 𝒳′ j = {}
      using map′ disj ins-B inf.commute insertE
      by (metis (no-types, lifting))
    moreover have ∀ i ∈ B. card (𝒳′ i) = f i
      using map′ card card-X′ ins-B
      by simp
    ultimately show
      ∃ 𝒳. A = ⋃ {𝒳 i | i. i ∈ B} ∧
                (∀ i ∈ B. card (𝒳 i) = f i) ∧
                   (∀ i j. i ≠ j ⟶ i ∈ B ∧ j ∈ B ⟶ 𝒳 i ∩ 𝒳 j = {})
      using part′
      by blast
  qed
```

3.2.2 Anonymity Quotient: Grid

```
fun anonymity_𝒬 :: 'a set ⇒ ('a, 'v) Election set set where
  anonymity_𝒬 A = quotient (elections-𝒜 A) (anonymity_ℛ (elections-𝒜 A))
```

— Here, we count the occurrences of a ballot per election in a set of elections for which the occurrences of the ballot per election coincide for all elections in the set.

```
fun vote-count_𝒬 :: 'a Preference-Relation ⇒ ('a, 'v) Election set ⇒ nat where
  vote-count_𝒬 p = π_𝒬 (vote-count p)

fun anonymity-class :: ('a :: finite, 'v) Election set ⇒
        (nat, 'a Ordered-Preference) vec where
  anonymity-class X = (χ p. vote-count_𝒬 (ord2pref p) X)

lemma anon-rel-equiv: equiv (elections-𝒜 UNIV) (anonymity_ℛ (elections-𝒜 UNIV))
proof −
  have subset: elections-𝒜 UNIV ⊆ well-formed-elections
    by simp
  have equiv: equiv well-formed-elections (anonymity_ℛ well-formed-elections)
   using anonymous-group-action.group-action-axioms rel-ind-by-group-act-equiv[of
           bijection_𝒱𝒢 well-formed-elections φ-anon well-formed-elections]
         rel-ind-by-coinciding-action-on-subset-eq-restr
    by fastforce
  have closed:
```

```
  closed-restricted-rel
    (anonymity_ℛ well-formed-elections) well-formed-elections (elections-𝒜 UNIV)
proof (unfold closed-restricted-rel.simps restricted-rel.simps, safe)
  fix
    A A' :: 'c set and
    V V' :: 'd set and
    p p' :: ('c, 'd) Profile
  assume elt: (A, V, p) ∈ elections-𝒜 UNIV
  hence wf-elections: (A, V, p) ∈ well-formed-elections
    unfolding elections-𝒜.simps
    by blast
  assume ((A, V, p), A', V', p') ∈ anonymity_ℛ well-formed-elections
  then obtain π :: 'd ⇒ 'd where
    bij-π: bij π and
    img: (A', V', p') = rename π (A, V, p)
    unfolding anonymity_ℛ.simps action-induced-rel.simps
              bijection_𝒱𝒢-def φ-anon.simps rewrite-carrier
              extensional-continuation.simps
    by auto
  hence (A', V', p') ∈ well-formed-elections
    using wf-elections rename-sound
    unfolding well-formed-elections-def
    by fastforce
  moreover have A' = A ∧ finite V
    using bij-π elt img rename.simps wf-elections well-formed-elections-def
    by auto
  moreover have ∀ v. v ∉ V' ⟶ (the-inv π v) ∉ V
    using elt Pair-inject UNIV-I bij-π rename.simps
          f-the-inv-into-f-bij-betw image-eqI img
    unfolding elections-𝒜.simps
    by (metis (mono-tags, opaque-lifting))
  moreover have ∀ v. v ∉ V' ⟶ p' v = p (the-inv π v)
    using img
    by simp
  ultimately show (A', V', p') ∈ elections-𝒜 UNIV
    using elt img
    unfolding elections-𝒜.simps rename.simps
    by auto
qed
have
  anonymity_ℛ (elections-𝒜 UNIV) =
    restricted-rel (anonymity_ℛ well-formed-elections) (elections-𝒜 UNIV)
      well-formed-elections
proof (unfold restricted-rel.simps, safe)
  fix
    A A' :: 'c set and
    V V' :: 'd set and
    p p' :: ('c, 'd) Profile
  assume rel: ((A, V, p), A', V', p') ∈ anonymity_ℛ (elections-𝒜 UNIV)
  hence (A, V, p) ∈ well-formed-elections
    unfolding anonymity_ℛ.simps action-induced-rel.simps elections-𝒜.simps
    by blast
  moreover obtain π :: 'd ⇒ 'd where
    bij π and
    (A', V', p') = rename π (A, V, p)
    using rel
    unfolding anonymity_ℛ.simps action-induced-rel.simps
              bijection_𝒱𝒢-def φ-anon.simps rewrite-carrier
              extensional-continuation.simps
    by auto
  ultimately show ((A, V, p), A', V', p') ∈ anonymity_ℛ well-formed-elections
    unfolding anonymity_ℛ.simps action-induced-rel.simps
              bijection_𝒱𝒢-def φ-anon.simps rewrite-carrier
    by auto
next
  fix
    A A' :: 'c set and
    V V' :: 'd set and
    p p' :: ('c, 'd) Profile
  assume ((A, V, p), A', V', p') ∈ anonymity_ℛ (elections-𝒜 UNIV)
  thus (A, V, p) ∈ elections-𝒜 UNIV
    unfolding anonymity_ℛ.simps action-induced-rel.simps
    by blast
next
  fix
    A A' :: 'c set and
    V V' :: 'd set and
    p p' :: ('c, 'd) Profile
  assume
    rel: ((A, V, p), A', V', p') ∈ anonymity_ℛ (elections-𝒜 UNIV)
  hence ((A, V, p), A', V', p') ∈ anonymity_ℛ well-formed-elections
    unfolding anonymity_ℛ.simps action-induced-rel.simps
    by fastforce
  moreover have elt: (A, V, p) ∈ elections-𝒜 UNIV
    using rel
    unfolding anonymity_ℛ.simps action-induced-rel.simps
    by blast
  ultimately have
    ((A, V, p), A', V', p') ∈ restricted-rel
    (anonymity_ℛ well-formed-elections) (elections-𝒜 UNIV) well-formed-elections
    using equiv
    unfolding restricted-rel.simps equiv-def refl-on-def
    by blast
  hence (A', V', p') ∈ elections-𝒜 UNIV
    using closed elt
    unfolding closed-restricted-rel.simps
    by blast
  thus (A', V', p') ∈ well-formed-elections
```

```
      using subset
      by blast
  next
    fix
      A A' :: 'c set and
      V V' :: 'd set and
      p p' :: ('c, 'd) Profile
    assume
      (A, V, p) ∈ elections-𝒜 UNIV and
      ((A, V, p), A', V', p') ∈ anonymity_ℛ well-formed-elections
    moreover from this
    obtain π :: 'd ⇒ 'd where
      bij π and
      (A', V', p') = rename π (A, V, p)
      unfolding anonymity_ℛ.simps action-induced-rel.simps
                bijection_𝒱𝒢-def φ-anon.simps rewrite-carrier
                extensional-continuation.simps
      by auto
    ultimately show ((A, V, p), A', V', p') ∈ anonymity_ℛ (elections-𝒜 UNIV)
      unfolding anonymity_ℛ.simps action-induced-rel.simps
                bijection_𝒱𝒢-def φ-anon.simps rewrite-carrier
                extensional-continuation.simps
      by auto
  qed
  also have ... = Restr (anonymity_ℛ well-formed-elections) (elections-𝒜 UNIV)
    using restr-equals-restricted-rel closed subset
    by blast
  finally have
    anonymity_ℛ (elections-𝒜 UNIV) =
      Restr (anonymity_ℛ well-formed-elections) (elections-𝒜 UNIV)
    by simp
  thus ?thesis
    using equiv-rel-restr subset equiv
    by metis
qed
```

We assume that all elections consist of a fixed finite alternative set of size n and finite subsets of an infinite voter universe. Profiles are linear orders on the alternatives. Then, we can operate on the natural-number-vectors of dimension $n!$ instead of the equivalence classes of the anonymity relation: Each dimension corresponds to one possible linear order on the alternative set, i.e., the possible preferences. Each equivalence class of elections corresponds to a vector whose entries denote the amount of voters per election in that class who vote the respective corresponding preference.

```
theorem anonymity_𝒬-isomorphism:
  assumes infinite (UNIV :: 'v set)
  shows bij-betw (anonymity-class :: ('a :: finite, 'v) Election set
          ⇒ nat^('a Ordered-Preference)) (anonymity_𝒬 (UNIV :: 'a set))
          (UNIV :: (nat^('a Ordered-Preference)) set)
proof (unfold bij-betw-def inj-on-def, intro conjI ballI impI)
  fix X Y :: ('a, 'v) Election set
  assume
    class-X: X ∈ anonymity_𝒬 UNIV and
    class-Y: Y ∈ anonymity_𝒬 UNIV and
    eq-vec: anonymity-class X = anonymity-class Y
  have ∀ E ∈ elections-𝒜 UNIV. finite (voters-ℰ E)
    by simp
  hence ∀ (E, E') ∈ anonymity_ℛ (elections-𝒜 UNIV). finite (voters-ℰ E)
    by simp
  moreover have subset: elections-𝒜 UNIV ⊆ well-formed-elections
    by simp
  ultimately have
    ∀ (E, E') ∈ anonymity_ℛ (elections-𝒜 UNIV).
        ∀ p. vote-count p E = vote-count p E'
    using anon-rel-vote-count
    by blast
  hence vote-count-invar:
    ∀ p. (vote-count p) respects (anonymity_ℛ (elections-𝒜 UNIV))
    unfolding congruent-def
    by blast
  have quotient-count:
    ∀ X ∈ anonymity_𝒬 UNIV. ∀ p. ∀ E ∈ X. vote-count_𝒬 p X = vote-count p E
    using pass-to-quotient[of anonymity_ℛ (elections-𝒜 UNIV)]
        vote-count-invar anon-rel-equiv
    unfolding anonymity_𝒬.simps anonymity_ℛ.simps vote-count_𝒬.simps
    by metis
  moreover from anon-rel-equiv
  obtain
    E E' :: ('a, 'v) Election where
    E-in-X: E ∈ X and
    E'-in-Y: E' ∈ Y
    using class-X class-Y equiv-Eps-in
    unfolding anonymity_𝒬.simps
    by metis
  ultimately have
    ∀ p. vote-count_𝒬 p X = vote-count p E ∧ vote-count_𝒬 p Y = vote-count p E'
    using class-X class-Y
    by blast
  moreover with eq-vec have
    ∀ p. vote-count_𝒬 (ord2pref p) X = vote-count_𝒬 (ord2pref p) Y
    unfolding anonymity-class.simps
    using UNIV-I vec-lambda-inverse
    by metis
  ultimately have ∀ p. vote-count (ord2pref p) E = vote-count (ord2pref p) E'
    by simp
  hence eq: ∀ p ∈ {r. linear-order-on (UNIV :: 'a set) r}.
          vote-count p E = vote-count p E'
    using pref2ord-inverse
    by metis
  from anon-rel-equiv class-X class-Y have subset-fixed-alts:
    X ⊆ elections-𝒜 UNIV ∧ Y ⊆ elections-𝒜 UNIV
    unfolding anonymity_𝒬.simps
    using in-quotient-imp-subset
    by blast
  hence eq-alts: alternatives-ℰ E = UNIV ∧ alternatives-ℰ E' = UNIV
    using E-in-X E'-in-Y
    unfolding elections-𝒜.simps
    by blast
  with subset-fixed-alts have eq-complement:
    ∀ p ∈ UNIV − {r. linear-order-on (UNIV :: 'a set) r}.
      {v ∈ voters-ℰ E. profile-ℰ E v = p} = {}
      ∧ {v ∈ voters-ℰ E'. profile-ℰ E' v = p} = {}
    using E-in-X E'-in-Y
    unfolding elections-𝒜.simps well-formed-elections-def profile-def
    by auto
  hence ∀ p ∈ UNIV − {r. linear-order-on (UNIV :: 'a set) r}.
        vote-count p E = 0 ∧ vote-count p E' = 0
    unfolding card-eq-0-iff vote-count.simps
    by simp
  with eq have eq-vote-count: ∀ p. vote-count p E = vote-count p E'
    using DiffI UNIV-I
    by metis
  moreover from subset-fixed-alts E-in-X E'-in-Y
    have finite (voters-ℰ E) ∧ finite (voters-ℰ E')
    unfolding elections-𝒜.simps
    by blast
  moreover from subset-fixed-alts E-in-X E'-in-Y
    have (E, E') ∈ (elections-𝒜 UNIV) × (elections-𝒜 UNIV)
    by blast
  moreover from this
  have (∀ v. v ∉ voters-ℰ E ⟶ profile-ℰ E v = {})
      ∧ (∀ v. v ∉ voters-ℰ E' ⟶ profile-ℰ E' v = {})
    by simp
  ultimately have (E, E') ∈ anonymity_ℛ (elections-𝒜 UNIV)
    using eq-alts vote-count-anon-rel
    by metis
  hence anonymity_ℛ (elections-𝒜 UNIV) `` {E} =
            anonymity_ℛ (elections-𝒜 UNIV) `` {E'}
    using anon-rel-equiv equiv-class-eq
    by metis
  also have anonymity_ℛ (elections-𝒜 UNIV) `` {E} = X
   using E-in-X class-X anon-rel-equiv Image-singleton-iff equiv-class-eq quotientE
    unfolding anonymity_𝒬.simps
    by (metis (no-types, lifting))
  also have anonymity_ℛ (elections-𝒜 UNIV) `` {E'} = Y
   using E'-in-Y class-Y anon-rel-equiv Image-singleton-iff equiv-class-eq quotientE
```

```
    unfolding anonymity_Q.simps
    by (metis (no-types, lifting))
  finally show X = Y
    by simp
next
  have (UNIV :: (nat, 'a Ordered-Preference) vec set) ⊆
      (anonymity-class :: ('a, 'v) Election set ⇒ (nat, 'a Ordered-Preference) vec) `
      anonymity_Q UNIV
  proof (unfold anonymity-class.simps, safe)
    fix x :: (nat, 'a Ordered-Preference) vec
    have finite (UNIV :: 'a Ordered-Preference set)
      by simp
    hence finite {x$i | i. i ∈ UNIV}
      using finite-Atleast-Atmost-nat
      by blast
    hence (∑ i ∈ UNIV. x$i) < ∞
      using enat-ord-code
      by simp
    moreover have 0 ≤ (∑ i ∈ UNIV. x$i)
      by blast
    ultimately obtain V :: 'v set where
      fin-V: finite V and
      card V = (∑ i ∈ UNIV. x$i)
      using assms infinite-arbitrarily-large
      by metis
    then obtain X' :: 'a Ordered-Preference ⇒ 'v set where
      card': ∀ i. card (X' i) = x$i and
      partition': V = ⋃ {X' i | i. i ∈ UNIV} and
      disjoint': ∀ i j. i ≠ j ⟶ X' i ∩ X' j = {}
      using obtain-partition[of V UNIV ($) x]
      by auto
    obtain X :: 'a Preference-Relation ⇒ 'v set where
      def-X: X = (λ i. if i ∈ {r. linear-order r}
                      then X' (pref2ord i) else {})
      by simp
    hence {X i | i. i ∉ {r. linear-order r}} ⊆ {{}}
      by auto
    moreover have
      {X i | i. i ∈ {r. linear-order r}} =
          {X' (pref2ord i) | i. i ∈ {r. linear-order r}}
      using def-X
      by metis
    moreover have
      {X i | i. i ∈ UNIV} =
          {X i | i. i ∈ {r. linear-order r}}
          ∪ {X i | i. i ∈ UNIV − {r. linear-order r}}
      by blast
    ultimately have
      {X i | i. i ∈ UNIV} = {X' (pref2ord i) | i. i ∈ {r. linear-order r}}
      ∨ {X i | i. i ∈ UNIV} =
          {X′ (pref2ord i) | i. i ∈ {r. linear-order r}} ∪ {{}}
      by auto
    also have
      {X′ (pref2ord i) | i. i ∈ {r. linear-order r}} = {X′ i | i. i ∈ UNIV}
      using iso-tuple-UNIV-I pref2ord-cases
      by metis
    finally have
      {X i | i. i ∈ UNIV} = {X′ i | i. i ∈ UNIV} ∨
        {X i | i. i ∈ UNIV} = {X′ i | i. i ∈ UNIV} ∪ {{}}
      by simp
    hence ⋃ {X i | i. i ∈ UNIV} = ⋃ {X′ i | i. i ∈ UNIV}
      using Sup-union-distrib ccpo-Sup-singleton sup-bot.right-neutral
      by (metis (no-types, lifting))
    hence partition: V = ⋃ {X i | i. i ∈ UNIV}
      using partition′
      by simp
    moreover have ∀ i j. i ≠ j ⟶ X i ∩ X j = {}
      using disjoint′ def-X pref2ord-inject
      by auto
    ultimately have ∀ v ∈ V. ∃! i. v ∈ X i
      by auto
    then obtain p′ :: ′v ⇒ ′a Preference-Relation where
      p-X: ∀ v ∈ V. v ∈ X (p′ v) and
      p-disj: ∀ v ∈ V. ∀ i. i ≠ p′ v ⟶ v ∉ X i
      by metis
    then obtain p :: ′v ⇒ ′a Preference-Relation where
      p-in-V-then-p′: p = (λ v. if v ∈ V then p′ v else {})
      by simp
    hence lin-ord: ∀ v ∈ V. linear-order (p v)
      using def-X p-X p-disj
      by fastforce
    hence wf-elections: (UNIV, V, p) ∈ elections-𝒜 UNIV
      using fin-V
      unfolding p-in-V-then-p′ elections-𝒜.simps
              well-formed-elections-def profile-def
      by auto
    hence ∀ i. ∀ E ∈ anonymity_ℛ (elections-𝒜 UNIV) ‘‘ {(UNIV, V, p)}.
            vote-count i E = vote-count i (UNIV, V, p)
      using fin-V anon-rel-vote-count[of (UNIV, V, p) - elections-𝒜 UNIV]
      by simp
    moreover have
      (UNIV, V, p) ∈ anonymity_ℛ (elections-𝒜 UNIV) ‘‘ {(UNIV, V, p)}
      using anon-rel-equiv wf-elections
      unfolding Image-def equiv-def refl-on-def
      by blast
    ultimately have eq-vote-count:
      ∀ i. vote-count i ‘
          (anonymity_ℛ (elections-𝒜 UNIV) ‘‘ {(UNIV, V, p)}) =
              {vote-count i (UNIV, V, p)}
          by blast
      have ∀ i. ∀ v ∈ V. p v = i ⟷ v ∈ X i
          using p-X p-disj
          unfolding p-in-V-then-p'
          by metis
      hence ∀ i. {v ∈ V. p v = i} = {v ∈ V. v ∈ X i}
          by blast
      moreover have ∀ i. X i ⊆ V
          using partition
          by blast
      ultimately have rewr-preimg: ∀ i. {v ∈ V. p v = i} = X i
          by auto
      hence ∀ i ∈ {r. linear-order r}.
                vote-count i (UNIV, V, p) = x$(pref2ord i)
          using def-X card'
          by simp
      hence ∀ i ∈ {r. linear-order r}.
         vote-count i ` (anonymity_ℛ (elections-𝒜 UNIV) `` {(UNIV, V, p)}) =
            {x$(pref2ord i)}
          using eq-vote-count
          by metis
      hence
          ∀ i ∈ {r. linear-order r}.
          vote-count_𝒬 i (anonymity_ℛ (elections-𝒜 UNIV) `` {(UNIV, V, p)}) =
              x$(pref2ord i)
          unfolding vote-count_𝒬.simps π_𝒬.simps singleton-set.simps
          using is-singleton-altdef singleton-set-def-if-card-one
          by fastforce
      hence ∀ i. vote-count_𝒬 (ord2pref i)
          (anonymity_ℛ (elections-𝒜 UNIV) `` {(UNIV, V, p)}) = x$i
          using ord2pref ord2pref-inverse
          by metis
      hence anonymity-class
          (anonymity_ℛ (elections-𝒜 UNIV) `` {(UNIV, V, p)}) = x
          using anonymity-class.simps vec-lambda-unique
          by (metis (no-types, lifting))
      moreover have
          anonymity_ℛ (elections-𝒜 UNIV) `` {(UNIV, V, p)} ∈ anonymity_𝒬 UNIV
          using wf-elections
          unfolding anonymity_𝒬.simps quotient-def
          by blast
      ultimately show
          x ∈ (λ X :: ('a, 'v) Election set. χ p. vote-count_𝒬 (ord2pref p) X)
                ` anonymity_𝒬 UNIV
          using anonymity-class.elims
          by blast
    qed
    thus (anonymity-class :: ('a, 'v) Election set
            ⇒ (nat, 'a Ordered-Preference) vec) ‘
            anonymity_𝒬 UNIV =
              (UNIV :: (nat, 'a Ordered-Preference) vec set)
          by blast
    qed
```

3.2.3 Homogeneity Quotient: Simplex

```
fun vote-fraction :: 'a Preference-Relation ⇒ ('a, 'v) Election ⇒ rat where
  vote-fraction r E =
    (if (finite (voters-ℰ E) ∧ voters-ℰ E ≠ {})
      then (Fract (vote-count r E) (card (voters-ℰ E))) else 0)

fun anonymity-homogeneity_ℛ :: ('a, 'v) Election set ⇒ ('a, 'v) Election rel where
  anonymity-homogeneity_ℛ ℰ =
    {(E, E') | E E'. E ∈ ℰ ∧ E' ∈ ℰ
                  ∧ (finite (voters-ℰ E) = finite (voters-ℰ E'))
                  ∧ (∀ r. vote-fraction r E = vote-fraction r E')}

fun anonymity-homogeneity_𝒬 :: 'a set ⇒ ('a, 'v) Election set set where
  anonymity-homogeneity_𝒬 A =
    quotient (elections-𝒜 A) (anonymity-homogeneity_ℛ (elections-𝒜 A))

fun vote-fraction_𝒬 :: 'a Preference-Relation ⇒ ('a, 'v) Election set ⇒ rat where
  vote-fraction_𝒬 p = π_𝒬 (vote-fraction p)

fun anonymity-homogeneity-class :: ('a :: finite, 'v) Election set ⇒
        (rat, 'a Ordered-Preference) vec where
  anonymity-homogeneity-class ℰ = (χ p. vote-fraction_𝒬 (ord2pref p) ℰ)
```

Maps each rational real vector entry to the corresponding rational. If the entry is not rational, the corresponding entry will be undefined.

```
fun rat-vector :: real^'b ⇒ rat^'b where
  rat-vector v = (χ p. the-inv of-rat (v$p))

fun rat-vector-set :: (real^'b) set ⇒ (rat^'b) set where
  rat-vector-set V = rat-vector ‘ {v ∈ V. ∀ i. v$i ∈ ℚ}

definition standard-basis :: (real^'b) set where
  standard-basis ≡ {v. ∃ b. v$b = 1 ∧ (∀ c ≠ b. v$c = 0)}
```

The rational points in the simplex.

```
definition vote-simplex :: (rat^'b) set where
  vote-simplex ≡
    insert 0 (rat-vector-set (convex hull (standard-basis :: (real^'b) set)))
```

Auxiliary Lemmas

```
lemma convex-combination-in-convex-hull:
  fixes
    X :: (real^'b) set and
    x :: real^'b
  assumes ∃ f :: (real^'b) ⇒ real.
           (∑ y ∈ X. f y) = 1 ∧ (∀ x ∈ X. f x ≥ 0)
             ∧ x = (∑ x ∈ X. (f x) *R x)
  shows x ∈ convex hull X
  using assms
proof (induction card X arbitrary: X x)
  case 0
  fix
    X :: (real^'b) set and
    x :: real^'b
  assume
    0 = card X and
    ∃ f. (∑ y ∈ X. f y) = 1 ∧ (∀ x ∈ X. 0 ≤ f x) ∧ x = (∑ x ∈ X. f x *R x)
  hence (∀ f. (∑ y ∈ X. f y) = 0) ∧ (∃ f. (∑ y ∈ X. f y) = 1)
    using card-0-eq empty-iff sum.infinite sum.neutral zero-neq-one
    by metis
  hence ∃ f. (∑ y ∈ X. f y) = 1 ∧ (∑ y ∈ X. f y) = 0
    by metis
  hence False
    using zero-neq-one
    by metis
  thus ?case
    by simp
next
  case (Suc n)
  fix
    X :: (real^'b) set and
    x :: real^'b and
    n :: nat
  assume
    card: Suc n = card X and
    ∃ f. (∑ y ∈ X. f y) = 1 ∧ (∀ x ∈ X. 0 ≤ f x) ∧ x = (∑ x ∈ X. f x *R x)
and
    hyp: ⋀ (X :: (real^'b) set) x. n = card X
           ⟹ ∃ f. (∑ y ∈ X. f y) = 1 ∧ (∀ x ∈ X. 0 ≤ f x) ∧ x =
                  (∑ x ∈ X. f x *R x)
           ⟹ x ∈ convex hull X
  then obtain f :: (real^'b) ⇒ real where
    sum: (∑ y ∈ X. f y) = 1 and
    nonneg: ∀ x ∈ X. 0 ≤ f x and
    x-sum: x = (∑ x ∈ X. f x *R x)
    by blast
  have card X > 0
    using card
    by linarith
  hence fin: finite X
    using card-gt-0-iff
    by blast
  have n = 0 ⟶ card X = 1
    using card
    by presburger
  hence n = 0 ⟶ (∃ y. X = {y} ∧ f y = 1)
    using sum nonneg One-nat-def add.right-neutral card-1-singleton-iff
        empty-iff finite.emptyI sum.insert sum.neutral
    by (metis (no-types, opaque-lifting))
  hence n = 0 ⟶ (∃ y. X = {y} ∧ x = y)
    using x-sum
    by fastforce
  hence n = 0 ⟶ x ∈ X
    by blast
  moreover have n > 0 ⟶ x ∈ convex hull X
  proof (safe)
    assume 0 < n
    hence card-X-gt-one: card X > 1
      using card
      by simp
    have (∀ y ∈ X. f y ≥ 1) ⟶ (∑ y ∈ X. f y) ≥ (∑ x ∈ X. 1)
      using fin sum-mono
      by metis
    moreover have (∑ x ∈ X. 1) = card X
      by force
    ultimately have (∀ y ∈ X. f y ≥ 1) ⟶ card X ≤ (∑ y ∈ X. f y)
      by force
    hence (∀ y ∈ X. f y ≥ 1) ⟶ 1 < (∑ y ∈ X. f y)
      using card-X-gt-one
      by linarith
    then obtain y :: real^'b where
      y-in-X: y ∈ X and
      f-y-lt-one: f y < 1
      using sum
      by auto
    hence 1 − f y ≠ 0 ∧ x = f y *R y + (∑ x ∈ X − {y}. f x *R x)
      using fin sum.remove x-sum
      by simp
    moreover have
      ∀ α ≠ 0. (∑ x ∈ X − {y}. f x *R x) =
          α *R (∑ x ∈ X − {y}. (f x / α) *R x)
      unfolding scaleR-sum-right
      by simp
    ultimately have convex-comb:
      x = f y *R y + (1 − f y) *R (∑ x ∈ X − {y}. (f x / (1 − f y)) *R x)
      by simp
    obtain f' :: real^'b ⇒ real where
      def': f' = (λ x. f x / (1 − f y))
      by simp
```

```
hence ∀ x ∈ X − {y}. f′ x ≥ 0
  using nonneg f-y-lt-one
  by fastforce
moreover have
  (∑ y ∈ X − {y}. f′ y) = (∑ x ∈ X − {y}. f x) / (1 − f y)
  unfolding def′ sum-divide-distrib
  by simp
moreover have
  (∑ x ∈ X − {y}. f x) / (1 − f y) = (1 − f y) / (1 − f y)
  using sum y-in-X
  by (simp add: fin sum.remove)
moreover have (1 − f y) / (1 − f y) = 1
  using f-y-lt-one
  by simp
ultimately have
  (∑ y ∈ X − {y}. f′ y) = 1 ∧ (∀ x ∈ X − {y}. 0 ≤ f′ x)
       ∧ (∑ x ∈ X − {y}. (f x / (1 − f y)) *_R x) =
     (∑ x ∈ X − {y}. f′ x *_R x)
  using def′
  by metis
hence ∃ f′. (∑ y ∈ X − {y}. f′ y) = 1 ∧ (∀ x ∈ X − {y}. 0 ≤ f′ x)
          ∧ (∑ x ∈ X − {y}. (f x / (1 − f y)) *_R x) =
     (∑ x ∈ X − {y}. f′ x *_R x)
  by metis
moreover have card (X − {y}) = n
  using card y-in-X
  by simp
ultimately have
  (∑ x ∈ X − {y}. (f x / (1 − f y)) *_R x) ∈ convex hull (X − {y})
  using hyp
  by blast
hence (∑ x ∈ X − {y}. (f x / (1 − f y)) *_R x) ∈ convex hull X
  using Diff-subset hull-mono in-mono
  by (metis (no-types, lifting))
moreover have f y ≥ 0 ∧ 1 − f y ≥ 0
  using f-y-lt-one nonneg y-in-X
  by simp
moreover have f y + (1 − f y) ≥ 0
  by simp
moreover have y ∈ convex hull X
  using y-in-X
  by (simp add: hull-inc)
moreover have
  ∀ x y. x ∈ convex hull X ∧ y ∈ convex hull X ⟶
    (∀ a ≥ 0. ∀ b ≥ 0. a + b = 1 ⟶ a *_R x + b *_R y ∈ convex hull X)
  using convex-def convex-convex-hull
  by (metis (no-types, opaque-lifting))
ultimately show x ∈ convex hull X
  using convex-comb
```

```
      by simp
  qed
  ultimately show x ∈ convex hull X
    using hull-inc
    by fastforce
qed

lemma standard-simplex-rewrite: convex hull standard-basis =
    {v :: (real^'b). (∀ i. v$i ≥ 0) ∧ (∑ y ∈ UNIV. v$y) = 1}
proof (unfold convex-def hull-def, intro equalityI)
  let ?simplex = {v :: (real^'b). (∀ i. v$i ≥ 0) ∧ (∑ y ∈ UNIV. v$y) = 1}
  have ∀ v :: (real^'b) ∈ standard-basis. ∃ b.
        v$b = 1 ∧ (∀ c. c ≠ b ⟶ v$c = 0)
    unfolding standard-basis-def
    by simp
  then obtain one :: real^'b ⇒ 'b where
    def-map: ∀ v ∈ standard-basis. v$(one v) = 1 ∧ (∀ i ≠ one v. v$i = 0)
    by metis
  hence ∀ v :: (real^'b) ∈ standard-basis. ∀ b. v$b ≥ 0
    using dual-order.refl zero-less-one-class.zero-le-one
    by metis
  moreover have ∀ v :: (real^'b) ∈ standard-basis.
      (∑ z ∈ UNIV. v$z) = (∑ z ∈ UNIV − {one v}. v$z) + v$(one v)
    unfolding def-map
    using add.commute finite insert-UNIV sum.insert-remove
    by metis
  moreover have ∀ v ∈ standard-basis.
        (∑ z ∈ UNIV − {one v}. v$z) + v$(one v) = 1
    using def-map
    by simp
  ultimately have standard-basis ⊆ ?simplex
    by force
  moreover have ∀ x :: (real^'b). ∀ y. (∑ z ∈ UNIV. (x + y)$z) =
            (∑ z ∈ UNIV. x$z) + (∑ z ∈ UNIV. y$z)
    by (simp add: sum.distrib)
  hence ∀ x :: (real^'b). ∀ y. ∀ u v.
    (∑ z ∈ UNIV. (u *R x + v *R y)$z) =
          u *R (∑ z ∈ UNIV. x$z) + v *R (∑ z ∈ UNIV. y$z)
    using scaleR-right.sum sum.cong vector-scaleR-component
    by (metis (no-types))
  hence ∀ x ∈ ?simplex. ∀ y ∈ ?simplex. ∀ u v.
          (∑ z ∈ UNIV. (u *R x + v *R y)$z) = u *R 1 + v *R 1
    by simp
  hence ∀ x ∈ ?simplex. ∀ y ∈ ?simplex. ∀ u ≥ 0. ∀ v ≥ 0.
      u + v = 1 ⟶ u *R x + v *R y ∈ ?simplex
    by simp
  ultimately show
    ⋂ {t. (∀ x ∈ t. ∀ y ∈ t. ∀ u ≥ 0. ∀ v ≥ 0.
                  u + v = 1 ⟶ u *R x + v *R y ∈ t)
          ∧ standard-basis ⊆ t} ⊆ ?simplex
    by blast
next
  show {v. (∀ i. 0 ≤ v$i) ∧ (∑ y ∈ UNIV. v$y) = 1} ⊆
      ⋂ {t. (∀ x ∈ t. ∀ y ∈ t. ∀ u ≥ 0. ∀ v ≥ 0.
                u + v = 1 ⟶ u *R x + v *R y ∈ t)
            ∧ (standard-basis :: (real^'b) set) ⊆ t}
  proof (intro subsetI)
    fix
      x :: real^'b and
      X :: (real^'b) set
    assume convex-comb:
      x ∈ {v. (∀ i. 0 ≤ v$i) ∧ (∑ y ∈ UNIV. v$y) = 1}
    have ∀ v ∈ standard-basis. ∃ b. v$b = 1 ∧ (∀ b' ≠ b. v$b' = 0)
      unfolding standard-basis-def
      by simp
    then obtain ind :: (real^'b) ⇒ 'b where
      ind-eq-one: ∀ v ∈ standard-basis. v$(ind v) = 1 and
      ind-eq-zero: ∀ v ∈ standard-basis. ∀ b ≠ (ind v). v$b = 0
      by metis
    hence ∀ v ∈ standard-basis. ∀ v' ∈ standard-basis.
            ind v = ind v' ⟶ (∀ b. v$b = v'$b)
      by metis
    hence inj-ind:
      ∀ v ∈ standard-basis. ∀ v' ∈ standard-basis.
        ind v = ind v' ⟶ v = v'
      unfolding vec-eq-iff
      by blast
    hence inj-on ind standard-basis
      unfolding inj-on-def
      by blast
    hence bij-ind-std: bij-betw ind standard-basis (ind ` standard-basis)
      unfolding bij-betw-def
      by simp
    obtain ind-inv :: 'b ⇒ (real^'b) where
      char-vec: ind-inv = (λ b. χ i. if i = b then 1 else 0)
      by blast
    hence in-basis: ∀ b. ind-inv b ∈ standard-basis
      unfolding standard-basis-def
      by simp
    moreover from this
    have ind-inv-map: ∀ b. ind (ind-inv b) = b
      using char-vec ind-eq-zero ind-eq-one axis-def axis-nth zero-neq-one
      by metis
    ultimately have ∀ b. ∃ v. v ∈ standard-basis ∧ b = ind v
      by metis
    hence univ: ind ` standard-basis = UNIV
      by blast
    have bij-inv: bij-betw ind-inv UNIV standard-basis
      using ind-inv-map bij-ind-std bij-betw-byWitness[of - ind] in-basis inj-ind
      unfolding image-subset-iff
      by simp
    obtain f :: (real^'b) ⇒ real where
      func: f = (λ v. if v ∈ standard-basis then x$(ind v) else 0)
      by blast
    hence (∑ y ∈ standard-basis. f y) = (∑ v ∈ standard-basis. x$(ind v))
      by simp
    also have ... = (∑ y ∈ ind ‘ standard-basis. x$y)
      using bij-ind-std sum-comp[of ind - - ($) x]
      by simp
    finally have sum-eq-one: (∑ y ∈ standard-basis. f y) = 1
      using univ convex-comb
      by simp
    have nonneg: ∀ v ∈ standard-basis. f v ≥ 0
      using func convex-comb
      by simp
    have ∀ v ∈ standard-basis. (χ i. x$(ind v) ∗ v$i)
         = (χ i. if i = ind v then x$(ind v) else 0)
      using ind-eq-one ind-eq-zero
      by fastforce
    hence
      ∀ v ∈ standard-basis.
         (x$(ind v)) ∗R v = (χ i. if i = ind v then x$(ind v) else 0)
      unfolding scaleR-vec-def
      by simp
    moreover have (∑ x ∈ standard-basis. (f x) ∗R x) =
        (∑ v ∈ standard-basis. (x$(ind v)) ∗R v)
      unfolding func
      by simp
    ultimately have (∑ x ∈ standard-basis. (f x) ∗R x)
         = (∑ v ∈ standard-basis. χ i. if i = ind v then x$(ind v) else 0)
      by force
    also have ... = (∑ b ∈ UNIV. χ i. if i = ind (ind-inv b)
                            then x$(ind (ind-inv b)) else 0)
      using bij-inv sum-comp
      unfolding comp-def
      by blast
    also have ... = (∑ b ∈ UNIV. χ i. if i = b then x$b else 0)
      using ind-inv-map
      by presburger
    finally have (∑ x ∈ standard-basis. (f x) ∗R x) =
        (∑ b ∈ UNIV. χ i. if i = b then x$b else 0)
      by simp
    hence (∑ x ∈ standard-basis. (f x) ∗R x) = x
      unfolding vec-eq-iff
      by simp
    hence ∃ f :: (real^'b) ⇒ real.
              (∑ y ∈ standard-basis. f y) = 1 ∧ (∀ x ∈ standard-basis. f x ≥ 0)
          ∧ x = (∑ x ∈ standard-basis. (f x) *R x)
      using sum-eq-one nonneg
      by blast
    thus x ∈ ⋂ {t. (∀ x ∈ t. ∀ y ∈ t. ∀ u ≥ 0. ∀ v ≥ 0.
                      u + v = 1 ⟶ u *R x + v *R y ∈ t)
               ∧ (standard-basis :: (real^'b) set) ⊆ t}
      using convex-combination-in-convex-hull
      unfolding convex-def hull-def
      by blast
  qed
qed

lemma fract-distr-helper:
  fixes a b c :: int
  assumes c ≠ 0
  shows Fract a c + Fract b c = Fract (a + b) c
  using add-rat assms mult.commute mult-rat-cancel distrib-right
  by metis

lemma anonymity-homogeneity-is-equivalence:
  fixes X :: ('a, 'v) Election set
  assumes ∀ E ∈ X. finite (voters-ℰ E)
  shows equiv X (anonymity-homogeneityℛ X)
proof (unfold equiv-def, safe)
  show refl-on X (anonymity-homogeneityℛ X)
    unfolding refl-on-def anonymity-homogeneityℛ.simps
    by blast
next
  show sym (anonymity-homogeneityℛ X)
    unfolding sym-def anonymity-homogeneityℛ.simps
    using sup-commute
    by simp
next
  show Relation.trans (anonymity-homogeneityℛ X)
  proof
    fix E E' F :: ('a, 'v) Election
    assume
      rel: (E, E') ∈ anonymity-homogeneityℛ X and
      rel': (E', F) ∈ anonymity-homogeneityℛ X
    hence finite (voters-ℰ E')
      unfolding anonymity-homogeneityℛ.simps
      using assms
      by fastforce
    from rel rel' have eq-frac:
      (∀ r. vote-fraction r E = vote-fraction r E') ∧
        (∀ r. vote-fraction r E' = vote-fraction r F)
      unfolding anonymity-homogeneityℛ.simps
      by blast
    hence ∀ r. vote-fraction r E = vote-fraction r F
```

```
      by metis
    thus (E, F) ∈ anonymity-homogeneity_ℛ X
      using rel rel′ snd-conv
      unfolding anonymity-homogeneity_ℛ.simps
      by blast
  qed
qed

lemma fract-distr:
  fixes
    A :: ′x set and
    f :: ′x ⇒ int and
    b :: int
  assumes
    finite A and
    b ≠ 0
  shows (∑ a ∈ A. Fract (f a) b) = Fract (∑ x ∈ A. f x) b
  using assms
proof (induction card A arbitrary: A f b)
  case 0
  fix
    A :: ′x set and
    f :: ′x ⇒ int and
    b :: int
  assume
    0 = card A and
    finite A and
    b ≠ 0
  hence (∑ a ∈ A. Fract (f a) b) = 0 ∧ (∑ y ∈ A. f y) = 0
    by simp
  thus ?case
    using 0 rat-number-collapse
    by simp
next
  case (Suc n)
  fix
    A :: ′x set and
    f :: ′x ⇒ int and
    b :: int and
    n :: nat
  assume
    card-A: Suc n = card A and
    fin-A: finite A and
    b-non-zero: b ≠ 0 and
    hyp: ⋀ A f b.
           n = card (A :: ′x set) ⟹
           finite A ⟹ b ≠ 0 ⟹ (∑ a ∈ A. Fract (f a) b) = Fract (∑ x ∈ A. f
x) b
  hence A ≠ {}
```

```
    by auto
  then obtain c :: 'x where
    c-in-A: c ∈ A
    by blast
  hence (∑ a ∈ A. Fract (f a) b) =
          (∑ a ∈ A − {c}. Fract (f a) b) + Fract (f c) b
    using fin-A
    by (simp add: sum-diff1)
  also have ... = Fract (∑ x ∈ A − {c}. f x) b + Fract (f c) b
    using hyp card-A fin-A b-non-zero c-in-A Diff-empty card-Diff-singleton
        diff-Suc-1 finite-Diff-insert
    by metis
  also have ... = Fract (∑ x ∈ A. f x) b
    using c-in-A fin-A b-non-zero fract-distr-helper
    by (simp add: sum-diff1)
  finally show (∑ a ∈ A. Fract (f a) b) = Fract (∑ x ∈ A. f x) b
    by blast
qed
```

Simplex Bijection

We assume all our elections to consist of a fixed finite alternative set of size n and finite subsets of an infinite voter universe. Profiles are linear orders on the alternatives. Then we can work on the standard simplex of dimension n! instead of the equivalence classes of the equivalence relation for anonymous + homogeneous voting rules (anon hom): Each dimension corresponds to one possible linear order on the alternative set, i.e., the possible preferences. Each equivalence class of elections corresponds to a vector whose entries denote the fraction of voters per election in that class who vote the respective corresponding preference.

```
theorem anonymity-homogeneity_𝒬-isomorphism:
  assumes infinite (UNIV :: 'v set)
  shows
    bij-betw (anonymity-homogeneity-class :: ('a :: finite, 'v) Election set ⇒
        rat^('a Ordered-Preference)) (anonymity-homogeneity_𝒬 (UNIV :: 'a set))
          (vote-simplex :: (rat^('a Ordered-Preference)) set)
proof (unfold bij-betw-def inj-on-def, intro conjI ballI impI)
  fix X Y :: ('a, 'v) Election set
  assume
    class-X: X ∈ anonymity-homogeneity_𝒬 UNIV and
    class-Y: Y ∈ anonymity-homogeneity_𝒬 UNIV and
    eq-vec: anonymity-homogeneity-class X = anonymity-homogeneity-class Y
  have equiv:
    equiv (elections-𝒜 UNIV) (anonymity-homogeneity_ℛ (elections-𝒜 UNIV))
    using anonymity-homogeneity-is-equivalence CollectD IntD1 inf-commute
    unfolding elections-𝒜.simps
    by (metis (no-types, lifting))
  hence subset:
```

```
  X ≠ {} ∧ X ⊆ elections-𝒜 UNIV ∧ Y ≠ {} ∧ Y ⊆ elections-𝒜 UNIV
  using class-X class-Y in-quotient-imp-non-empty in-quotient-imp-subset
  unfolding anonymity-homogeneity_𝒬.simps
  by blast
then obtain E E' :: ('a, 'v) Election where
  E-in-X: E ∈ X and
  E'-in-Y: E' ∈ Y
  by blast
hence class-X-E: anonymity-homogeneity_ℛ (elections-𝒜 UNIV) `` {E} = X
  using class-X equiv Image-singleton-iff equiv-class-eq quotientE
  unfolding anonymity-homogeneity_𝒬.simps
  by (metis (no-types, opaque-lifting))
hence ∀ F ∈ X. (E, F) ∈ anonymity-homogeneity_ℛ (elections-𝒜 UNIV)
  unfolding Image-def
  by blast
hence ∀ F ∈ X. ∀ p. vote-fraction p F = vote-fraction p E
  unfolding anonymity-homogeneity_ℛ.simps
  by fastforce
hence ∀ p. vote-fraction p ` X = {vote-fraction p E}
  using E-in-X
  by blast
hence ∀ p. vote-fraction_𝒬 p X = vote-fraction p E
  using is-singletonI singleton-set-def-if-card-one the-elem-eq
 unfolding is-singleton-altdef vote-fraction_𝒬.simps π_𝒬.simps singleton-set.simps
  by metis
hence eq-X-E:
  ∀ p. (anonymity-homogeneity-class X)$p = vote-fraction (ord2pref p) E
  unfolding anonymity-homogeneity-class.simps
  using vec-lambda-beta
  by metis
have class-Y-E': anonymity-homogeneity_ℛ (elections-𝒜 UNIV) `` {E'} = Y
  using class-Y equiv E'-in-Y Image-singleton-iff equiv-class-eq quotientE
  unfolding anonymity-homogeneity_𝒬.simps
  by (metis (no-types, opaque-lifting))
hence ∀ F ∈ Y. (E', F) ∈ anonymity-homogeneity_ℛ (elections-𝒜 UNIV)
  unfolding Image-def
  by blast
hence ∀ F ∈ Y. ∀ p. vote-fraction p E' = vote-fraction p F
  unfolding anonymity-homogeneity_ℛ.simps
  by blast
hence ∀ p. vote-fraction p ` Y = {vote-fraction p E'}
  using E'-in-Y
  by fastforce
hence ∀ p. vote-fraction_𝒬 p Y = vote-fraction p E'
  using is-singletonI singleton-set-def-if-card-one the-elem-eq
 unfolding is-singleton-altdef vote-fraction_𝒬.simps π_𝒬.simps singleton-set.simps
  by metis
hence eq-Y-E':
  ∀ p. (anonymity-homogeneity-class Y)$p = vote-fraction (ord2pref p) E'
```

```
    unfolding anonymity-homogeneity-class.simps
    using vec-lambda-beta
    by metis
  hence ∀ p. vote-fraction (ord2pref p) E = vote-fraction (ord2pref p) E'
    using eq-X-E eq-vec
    by metis
  hence eq-ord: ∀ p. linear-order p ⟶ vote-fraction p E = vote-fraction p E'
    using mem-Collect-eq pref2ord-inverse
    by metis
  have (∀ v. v ∈ voters-ℰ E ⟶ linear-order (profile-ℰ E v)) ∧
      (∀ v. v ∈ voters-ℰ E' ⟶ linear-order (profile-ℰ E' v))
    using subset E-in-X E'-in-Y
    unfolding elections-𝒜.simps well-formed-elections-def profile-def
    by fastforce
  hence ∀ p. ¬ linear-order p ⟶ vote-count p E = 0 ∧ vote-count p E' = 0
    unfolding vote-count.simps
    using card.infinite card-0-eq Collect-empty-eq
    by (metis (mono-tags, lifting))
  hence ∀ p. ¬ linear-order p ⟶ vote-fraction p E = 0 ∧ vote-fraction p E' = 0
    using int-ops rat-number-collapse
    by simp
  hence ∀ p. vote-fraction p E = vote-fraction p E'
    using eq-ord
    by metis
  hence (E, E') ∈ anonymity-homogeneity_ℛ (elections-𝒜 UNIV)
    using subset E-in-X E'-in-Y elections-𝒜.simps
    unfolding anonymity-homogeneity_ℛ.simps
    by blast
  thus X = Y
    using class-X-E class-Y-E' equiv equiv-class-eq
    by (metis (no-types, lifting))
next
  show (anonymity-homogeneity-class :: ('a, 'v) Election set
          ⇒ rat^('a Ordered-Preference))
      ` anonymity-homogeneity_𝒬 UNIV = vote-simplex
  proof (unfold vote-simplex-def, safe)
    fix X :: ('a, 'v) Election set
    assume
      quot: X ∈ anonymity-homogeneity_𝒬 UNIV and
      not-simplex:
      anonymity-homogeneity-class X ∉ rat-vector-set (convex hull standard-basis)
    have equiv-rel:
      equiv (elections-𝒜 UNIV) (anonymity-homogeneity_ℛ (elections-𝒜 UNIV))
      using anonymity-homogeneity-is-equivalence elections-𝒜.simps
      by blast
    then obtain E :: ('a, 'v) Election where
      E-in-X: E ∈ X and
      X = anonymity-homogeneity_ℛ (elections-𝒜 UNIV) `` {E}
      using quot anonymity-homogeneity_𝒬.simps equiv-Eps-in proj-Eps
      unfolding proj-def
      by metis
    hence rel: ∀ E' ∈ X. (E, E') ∈ anonymity-homogeneity_ℛ (elections-𝒜 UNIV)
      by simp
    hence ∀ p. ∀ E' ∈ X.
        vote-fraction (ord2pref p) E' = vote-fraction (ord2pref p) E
      unfolding anonymity-homogeneity_ℛ.simps
      by fastforce
    hence ∀ p. vote-fraction (ord2pref p) ` X = {vote-fraction (ord2pref p) E}
      using E-in-X
      by blast
    hence repr: ∀ p. vote-fraction_𝒬 (ord2pref p) X = vote-fraction (ord2pref p) E
      using is-singletonI singleton-set-def-if-card-one the-elem-eq
      unfolding vote-fraction_𝒬.simps π_𝒬.simps is-singleton-altdef
      by metis
    have ∀ p. vote-count (ord2pref p) E ≥ 0
      by simp
    hence ∀ p. card (voters-ℰ E) > 0 ⟶
        Fract (int (vote-count (ord2pref p) E)) (int (card (voters-ℰ E))) ≥ 0
      using zero-le-Fract-iff
      by simp
    hence ∀ p. vote-fraction (ord2pref p) E ≥ 0
      unfolding vote-fraction.simps card-gt-0-iff
      by simp
    hence ∀ p. vote-fraction_𝒬 (ord2pref p) X ≥ 0
      using repr
      by simp
    hence geq-zero: ∀ p. real-of-rat (vote-fraction_𝒬 (ord2pref p) X) ≥ 0
      using zero-le-of-rat-iff
      by blast
    have voters-ℰ E = {} ∨ infinite (voters-ℰ E) ⟶
      (∀ p. real-of-rat (vote-fraction p E) = 0)
      by simp
    hence zero-case:
      voters-ℰ E = {} ∨ infinite (voters-ℰ E) ⟶
      (χ p. real-of-rat (vote-fraction_𝒬 (ord2pref p) X)) = 0
      using repr
      unfolding zero-vec-def
      by simp
    let ?vote-sum = (∑ p ∈ UNIV. vote-count p E)
    have finite (UNIV :: ('a × 'a) set)
      by simp
    hence eq-card: finite (voters-ℰ E) ⟶ card (voters-ℰ E) = ?vote-sum
      using vote-count-sum
      by metis
    hence finite (voters-ℰ E) ∧ voters-ℰ E ≠ {} ⟶
      (∑ p ∈ UNIV. vote-fraction p E) =
        (∑ p ∈ UNIV. Fract (vote-count p E) ?vote-sum)
      unfolding vote-fraction.simps
      by presburger
    moreover have fin-imp-sum-gt-zero:
      finite (voters-ℰ E) ∧ voters-ℰ E ≠ {} ⟶ ?vote-sum > 0
      using eq-card
      by fastforce
    hence finite (voters-ℰ E) ∧ voters-ℰ E ≠ {} ⟶
    (∑ p ∈ UNIV. Fract (vote-count p E) ?vote-sum) = Fract ?vote-sum ?vote-sum
      using fract-distr[of UNIV ?vote-sum] card-0-eq eq-card of-nat-eq-0-iff
            finite-class.finite-UNIV of-nat-sum sum.cong
      by (metis (no-types, lifting))
    moreover have
      finite (voters-ℰ E) ∧ voters-ℰ E ≠ {} ⟶ Fract ?vote-sum ?vote-sum = 1
      using fin-imp-sum-gt-zero Fract-le-one-iff Fract-less-one-iff
            of-nat-0-less-iff order-le-less order-less-irrefl
      by metis
    ultimately have fin-imp-sum-eq-one:
      finite (voters-ℰ E) ∧ voters-ℰ E ≠ {}
      ⟶ (∑ p ∈ UNIV. vote-fraction p E) = 1
      by presburger
    have ∀ p. ¬ linear-order p ⟶ vote-count p E = 0
      using E-in-X rel
      unfolding anonymity-homogeneity_𝒬.simps quotient-def vote-count.simps
                elections-𝒜.simps well-formed-elections-def profile-def
      by fastforce
    hence ∀ p. ¬ linear-order p ⟶ vote-fraction p E = 0
      using rat-number-collapse
      by simp
    moreover have (∑ p ∈ UNIV. vote-fraction p E) =
      (∑ p ∈ {r. linear-order r}. vote-fraction p E) +
      (∑ p ∈ UNIV − {r. linear-order r}. vote-fraction p E)
      using finite CollectD Collect-mono UNIV-I add.commute
            sum.subset-diff top-set-def
      by metis
    ultimately have (∑ p ∈ UNIV. vote-fraction p E) =
      (∑ p ∈ {r. linear-order r}. vote-fraction p E)
      by simp
    moreover have bij-betw ord2pref UNIV {p. linear-order p}
      using inj-def ord2pref-inject range-ord2pref
      unfolding bij-betw-def
      by blast
    ultimately have
      (∑ p ∈ UNIV. vote-fraction p E) =
         (∑ p ∈ UNIV. vote-fraction (ord2pref p) E)
      using comp-def sum-comp
      by auto
    hence finite (voters-ℰ E) ∧ voters-ℰ E ≠ {}
      ⟶ (∑ p ∈ UNIV. real-of-rat (vote-fraction (ord2pref p) E)) = 1
      using fin-imp-sum-eq-one of-rat-1 of-rat-sum
      by metis
```

hence (χ *p. real-of-rat* (*vote-fraction*$_{\mathcal{Q}}$ (*ord2pref p*) *X*)) = *0* ∨
((∀ *p.* (χ *p. real-of-rat* (*vote-fraction*$_{\mathcal{Q}}$ (*ord2pref p*) *X*))$*p* ≥ *0*)
∧ (∑ *x* ∈ *UNIV*. (χ *p. real-of-rat* (*vote-fraction*$_{\mathcal{Q}}$ (*ord2pref p*) *X*))$*x*) = *1*)
using *zero-case repr geq-zero*
by *force*
moreover have
∀ *p.* (χ *p. real-of-rat* (*vote-fraction*$_{\mathcal{Q}}$ (*ord2pref p*) *X*))$*p* ∈ ℚ
by *simp*
ultimately have *simplex-el*:
(χ *p. real-of-rat* (*vote-fraction*$_{\mathcal{Q}}$ (*ord2pref p*) *X*))
∈ {*x* ∈ *insert 0* (*convex hull standard-basis*). ∀ *i. x*$*i* ∈ ℚ}
using *standard-simplex-rewrite*
by *blast*
moreover have
∀ *p.* (*rat-vector* (χ *p. of-rat* (*vote-fraction*$_{\mathcal{Q}}$ (*ord2pref p*) *X*)))$*p* =
the-inv real-of-rat ((χ *p. real-of-rat* (*vote-fraction*$_{\mathcal{Q}}$ (*ord2pref p*) *X*))$*p*)
unfolding *rat-vector.simps*
using *vec-lambda-beta*
by *blast*
moreover have
∀ *p. the-inv real-of-rat*
((χ *p. real-of-rat* (*vote-fraction*$_{\mathcal{Q}}$ (*ord2pref p*) *X*))$*p*) =
the-inv real-of-rat (*real-of-rat* (*vote-fraction*$_{\mathcal{Q}}$ (*ord2pref p*) *X*))
by *simp*
moreover have *inv-of-rat*: ∀ *x* ∈ ℚ. *the-inv of-rat* (*of-rat x*) = *x*
unfolding *Rats-def*
using *the-inv-f-f injI of-rat-eq-iff*
by *metis*
hence ∀ *p. the-inv real-of-rat* (*real-of-rat* (*vote-fraction*$_{\mathcal{Q}}$ (*ord2pref p*) *X*)) =
vote-fraction$_{\mathcal{Q}}$ (*ord2pref p*) *X*
using *Rats-eq-range-nat-to-rat-surj surj-nat-to-rat-surj*
by *blast*
moreover have
∀ *p. vote-fraction*$_{\mathcal{Q}}$ (*ord2pref p*) *X* = (*anonymity-homogeneity-class X*)$*p*
by *simp*
ultimately have
∀ *p.* (*rat-vector* (χ *p. of-rat* (*vote-fraction*$_{\mathcal{Q}}$ (*ord2pref p*) *X*)))$*p* =
(*anonymity-homogeneity-class X*)$*p*
by *metis*
hence *rat-vector* (χ *p. of-rat* (*vote-fraction*$_{\mathcal{Q}}$ (*ord2pref p*) *X*))
= *anonymity-homogeneity-class X*
by *simp*
hence ∃ *x* ∈ {*x* ∈ *insert 0* (*convex hull standard-basis*). ∀ *i. x*$*i* ∈ ℚ}.
rat-vector x = *anonymity-homogeneity-class X*
using *simplex-el*
by *blast*
hence *rat-vector 0* = *anonymity-homogeneity-class X*
using *not-simplex image-iff insertE mem-Collect-eq*
unfolding *rat-vector-set.simps*

```
    by (metis (mono-tags, lifting))
  thus anonymity-homogeneity-class X = 0
    unfolding rat-vector.simps
    using Rats-0 inv-of-rat of-rat-0 vec-lambda-unique zero-index
    by (metis (no-types, lifting))
next
  have ∀ E ∈ (anonymity-homogeneity_𝓡 (elections-𝒜 UNIV))
        `` {(UNIV, {}, (λ v. {}))}. ∀ r. vote-fraction r E = 0
    unfolding anonymity-homogeneity_𝓡.simps
    by force
  moreover have
    ∀ E ∈ (anonymity-homogeneity_𝓡 (elections-𝒜 UNIV))
        `` {(UNIV, {}, (λ v. {}))}. finite (voters-ℰ E)
    unfolding Image-def anonymity-homogeneity_𝓡.simps
    by fastforce
  ultimately have all-zero:
    ∀ r. ∀ E ∈ (anonymity-homogeneity_𝓡 (elections-𝒜 UNIV))
        `` {(UNIV, {}, (λ v. {}))}. vote-fraction r E = 0
    by blast
  moreover have (UNIV, {}, λ v. {})
      ∈ (anonymity-homogeneity_𝓡 (elections-𝒜 UNIV) `` {(UNIV, {}, λ v. {})})
    unfolding anonymity-homogeneity_𝓡.simps Image-def elections-𝒜.simps
            well-formed-elections-def profile-def
    by simp
  ultimately have ∀ r. 0 ∈ vote-fraction r
          ` (anonymity-homogeneity_𝓡 (elections-𝒜 UNIV))
            `` {(UNIV, {}, (λ v. {}))}
    using image-eqI
    by (metis (mono-tags, lifting))
  hence
    ∀ r. vote-fraction r
      ` (anonymity-homogeneity_𝓡 (elections-𝒜 UNIV)
          `` {(UNIV, {}, λ v. {})}) = {0}
    using all-zero
    by blast
  hence ∀ r :: 'a Ordered-Preference. vote-fraction_𝒬 (ord2pref r)
        (anonymity-homogeneity_𝓡 (elections-𝒜 UNIV)
          `` {(UNIV, {}, λ v. {})}) = 0
    using is-singletonI singleton-insert-inj-eq' singleton-set-def-if-card-one
    unfolding vote-fraction_𝒬.simps π_𝒬.simps singleton-set.simps
            is-singleton-altdef singleton-set.simps
    by metis
  hence ∀ r :: 'a Ordered-Preference.
    (anonymity-homogeneity-class (anonymity-homogeneity_𝓡 (elections-𝒜 UNIV)
        `` {(UNIV, {}, λ v. {})}))$r = 0
    unfolding anonymity-homogeneity-class.simps
    using vec-lambda-beta
    by (metis (no-types))
  moreover have ∀ r :: 'a Ordered-Preference. 0$r = 0
```

```
      by simp
    ultimately have anonymity-homogeneity-class
      (anonymity-homogeneity_𝓡 (elections-𝒜 UNIV)
          ‘‘ {(UNIV, {}, λ v. {})}) = (0 :: (rat^('a Ordered-Preference)))
      using vec-eq-iff
      by (metis (no-types))
    moreover have (UNIV, {}, λ v. {}) ∈ elections-𝒜 UNIV
      unfolding elections-𝒜.simps well-formed-elections-def profile-def
      by simp
   hence (anonymity-homogeneity_𝓡 (elections-𝒜 UNIV) ‘‘ {(UNIV, {}, λ v. {})})
          ∈ anonymity-homogeneity_𝒬 UNIV
      unfolding anonymity-homogeneity_𝒬.simps quotient-def
      by blast
   ultimately show 0 ∈ anonymity-homogeneity-class ‘ anonymity-homogeneity_𝒬
UNIV
      using image-eqI
      by (metis (no-types))
  next
    fix x :: rat^('a Ordered-Preference)
    assume x ∈ rat-vector-set (convex hull standard-basis)
    — The following converts a rational vector x to real vector x′.
    then obtain x′ :: real^('a Ordered-Preference) where
      conv: x′ ∈ convex hull standard-basis and
      inv: ∀ p. x$p = the-inv real-of-rat (x′$p) and
      rat: ∀ p. x′$p ∈ ℚ
      unfolding rat-vector-set.simps rat-vector.simps
      by force
    hence convex: (∀ p. 0 ≤ x′$p) ∧ (∑ y ∈ UNIV. x′$y) = 1
      using standard-simplex-rewrite
      by blast
    have map: ∀ p. real-of-rat (x$p) = x′$p
      using inv rat f-the-inv-into-f inj-onCI of-rat-eq-iff
      unfolding Rats-def
      by (metis (no-types))
    have ∀ p. ∃ fract. Fract (fst fract) (snd fract) = x$p ∧ 0 < snd fract
      using quotient-of-unique
      by metis
    then obtain fraction′ :: 'a Ordered-Preference ⇒ (int × int) where
      ∀ p. x$p = Fract (fst (fraction′ p)) (snd (fraction′ p)) and
      pos′: ∀ p. 0 < snd (fraction′ p)
      by metis
    hence fract′: ∀ p. x′$p = (fst (fraction′ p)) / (snd (fraction′ p))
      using map div-by-0 divide-less-cancel of-int-0 of-int-pos of-rat-rat
      by metis
    hence ∀ p. (fst (fraction′ p)) / (snd (fraction′ p)) ≥ 0
      using convex
      by fastforce
    hence ∀ p. fst (fraction′ p) ≥ 0
      using pos′ not-less of-int-0-le-iff of-int-pos zero-le-divide-iff
      by metis
    hence ∀ p. fst (fraction' p) ∈ ℕ ∧ snd (fraction' p) ∈ ℕ
      using pos' nonneg-int-cases of-nat-in-Nats order-less-le
      by metis
    hence ∀ p. ∃ (n :: nat) (m :: nat).
      fst (fraction' p) = n ∧ snd (fraction' p) = m
      using Nats-cases
      by metis
    hence ∀ p. ∃ m :: nat × nat. fst (fraction' p) = int (fst m)
          ∧ snd (fraction' p) = int (snd m)
      by simp
    then obtain fraction :: 'a Ordered-Preference ⇒ (nat × nat) where
      eq: ∀ p. fst (fraction' p) = int (fst (fraction p)) ∧
              snd (fraction' p) = int (snd (fraction p))
      by metis
    hence fract: ∀ p. x'$p = (fst (fraction p)) / (snd (fraction p))
      using fract'
      by simp
    hence pos: ∀ p. 0 < snd (fraction p)
      using eq pos'
      by simp
    let ?prod = ∏ p ∈ UNIV. snd (fraction p)
    have fin: finite (UNIV :: 'a Ordered-Preference set)
      by simp
    hence finite {snd (fraction p) | p. p ∈ UNIV}
      using finite-Atleast-Atmost-nat
      by simp
    have pos-prod: ?prod > 0
      using pos
      by simp
    hence ∀ p. ?prod mod (snd (fraction p)) = 0
      using finite UNIV-I mod-mod-trivial mod-prod-eq mod-self prod-zero
      by (metis (no-types, lifting))
    hence div: ∀ p. (?prod div (snd (fraction p))) ∗ (snd (fraction p)) = ?prod
      using add.commute add-0 div-mult-mod-eq
      by metis
    obtain voter-amount :: 'a Ordered-Preference ⇒ nat where
      def-amount:
        voter-amount = (λ p ∈ UNIV. (fst (fraction p)) ∗ (?prod div (snd (fraction
  p))))
      by blast
    let ?voter-sum = (∑ p ∈ UNIV. (fst (fraction p)) ∗ (?prod div (snd (fraction
  p))))
    have rewrite-div: ∀ p. ?prod div (snd (fraction p)) = ?prod / (snd (fraction p))
      using div less-imp-of-nat-less nonzero-mult-div-cancel-right
          of-nat-less-0-iff of-nat-mult pos
      by metis
   hence ?voter-sum = (∑ p ∈ UNIV. (fst (fraction p)) ∗ (?prod / (snd (fraction
  p))))
```

```
    using def-amount
    by simp
  hence ?voter-sum = ?prod ∗ (∑ p ∈ UNIV. (fst (fraction p)) / (snd (fraction
p)))
    using mult-of-nat-commute sum.cong times-divide-eq-right
        vector-space-over-itself.scale-sum-right
    by (metis (mono-tags, lifting))
  hence rewrite-sum: ?voter-sum = ?prod
    using fract convex mult-cancel-left1 of-nat-eq-iff sum.cong
    by (metis (mono-tags, lifting))
  obtain V :: 'v set where
    fin-V: finite V and
    card-V-eq-sum: card V = ?voter-sum
    using assms infinite-arbitrarily-large
    by blast
  then obtain part :: 'a Ordered-Preference ⇒ 'v set where
    partition: V = ⋃ {part p | p. p ∈ UNIV} and
    disjoint: ∀ p p'. p ≠ p' ⟶ part p ∩ part p' = {} and
    card: ∀ p. card (part p) = voter-amount p
    using def-amount obtain-partition[of V UNIV voter-amount]
    by auto
  hence exactly-one-prof: ∀ v ∈ V. ∃!p. v ∈ part p
    by blast
  then obtain prof' :: 'v ⇒ 'a Ordered-Preference where
    maps-to-prof': ∀ v ∈ V. v ∈ part (prof' v)
    by metis
  then obtain prof :: 'v ⇒ 'a Preference-Relation where
    prof: prof = (λ v. if v ∈ V then ord2pref (prof' v) else {})
    by blast
  hence election: (UNIV, V, prof) ∈ elections-𝒜 UNIV
    unfolding elections-𝒜.simps well-formed-elections-def profile-def
    using fin-V ord2pref
    by auto
  have ∀ p. {v ∈ V. prof' v = p} = {v ∈ V. v ∈ part p}
    using maps-to-prof' exactly-one-prof
    by blast
  hence ∀ p. {v ∈ V. prof' v = p} = part p
    using partition
    by fastforce
  hence ∀ p. card {v ∈ V. prof' v = p} = voter-amount p
    using card
    by presburger
  moreover have
    ∀ p. ∀ v. (v ∈ {v ∈ V. prof' v = p}) = (v ∈ {v ∈ V. prof v = ord2pref p})
    using prof
    by (simp add: ord2pref-inject)
  ultimately have
    ∀ p :: 'a Ordered-Preference.
      vote-fraction (ord2pref p) (UNIV, V, prof) = Fract (voter-amount p) (card
V)
    using rat-number-collapse fin-V
    by simp
  moreover have
    ∀ p. Fract (voter-amount p) (card V) = (voter-amount p) / (card V)
    unfolding Fract-of-int-quotient of-rat-divide
    by simp
  moreover have
    ∀ p. (voter-amount p) / (card V) =
        ((fst (fraction p)) ∗ (?prod div (snd (fraction p)))) / ?prod
    using def-amount card-V-eq-sum rewrite-sum
    by force
  moreover have
    ∀ p. ((fst (fraction p)) ∗ (?prod div (snd (fraction p)))) / ?prod =
        (fst (fraction p)) / (snd (fraction p))
    using rewrite-div pos-prod
    by auto
  — The following are the percentages of voters voting for each linearly ordered
profile in (UNIV, V, prof) that equals the entries of the given vector.
  ultimately have
    ∀ p :: 'a Ordered-Preference.
        vote-fraction (ord2pref p) (UNIV, V, prof) = x'$p
    using fract
    by presburger
  moreover have
    ∀ E ∈ anonymity-homogeneityℛ (elections-𝒜 UNIV) `` {(UNIV, V, prof)}.
      ∀ p. vote-fraction (ord2pref p) E =
        vote-fraction (ord2pref p) (UNIV, V, prof)
    unfolding anonymity-homogeneityℛ.simps
    by fastforce
  ultimately have all-eq-vec:
    ∀ p. ∀ E ∈ anonymity-homogeneityℛ (elections-𝒜 UNIV)
      `` {(UNIV, V, prof)}. vote-fraction (ord2pref p) E = x$p
    using Re-complex-of-real Re-divide-of-real of-rat.rep-eq of-real-of-int-eq
        injI of-rat-eq-iff the-inv-f-f rat inv
    by (metis (mono-tags, opaque-lifting))
  moreover have
   (UNIV, V, prof) ∈ anonymity-homogeneityℛ (elections-𝒜 UNIV) `` {(UNIV,
V, prof)}
    using anonymity-homogeneityℛ.simps election
    by blast
  ultimately have ∀ p. vote-fraction (ord2pref p) `
    anonymity-homogeneityℛ (elections-𝒜 UNIV) `` {(UNIV, V, prof)} ⊇ {x$p}
    using image-insert insert-iff mk-disjoint-insert singletonD subsetI
    by (metis (no-types, lifting))
  hence ∀ p. vote-fraction (ord2pref p) `
   anonymity-homogeneityℛ (elections-𝒜 UNIV) `` {(UNIV, V, prof)} = {x$p}
    using all-eq-vec
    by blast
    hence x = anonymity-homogeneity-class
              (anonymity-homogeneity_ℛ (elections-𝒜 UNIV) `` {(UNIV, V, prof)})
     using is-singletonI singleton-inject singleton-set-def-if-card-one vec-lambda-unique
      unfolding anonymity-homogeneity-class.simps is-singleton-altdef
                vote-fraction_𝒬.simps π_𝒬.simps
      by (metis (no-types, lifting))
    thus x ∈ (anonymity-homogeneity-class
             :: ('a, 'v) Election set ⇒ rat^('a Ordered-Preference))
            ` anonymity-homogeneity_𝒬 UNIV
      unfolding anonymity-homogeneity_𝒬.simps quotient-def
      using election
      by blast
  qed
qed

end
```

Chapter 4

Component Types

4.1 Distance

theory *Distance*
  imports *HOL−Library.Extended-Real*
          Social-Choice-Types/Voting-Symmetry
begin

A general distance on a set X is a mapping d: $X \times X \mapsto R \cup \{+\infty\}$ such that for every x, y, z in X, the following four conditions are satisfied:

- $d(x, y) \geq 0$ (non-negativity);
- $d(x, y) = 0$ if and only if $x = y$ (identity of indiscernibles);
- $d(x, y) = d(y, x)$ (symmetry);
- $d(x, y) \leq d(x, z) + d(z, y)$ (triangle inequality).

  Moreover, a mapping that satisfies all but the second conditions is called a pseudo-distance, whereas a quasi-distance needs to satisfy the first three conditions (and not necessarily the last one).

4.1.1 Definition

type-synonym $'a$ *Distance* $= {'a} \Rightarrow {'a} \Rightarrow$ *ereal*

The un-curried version of a distance is defined on tuples.

fun *tup* :: $'a$ *Distance* $\Rightarrow$ $('a * 'a \Rightarrow$ *ereal*$)$ **where**
  tup d = (λ *pair. d* (*fst pair*) (*snd pair*))

definition *distance* :: $'a$ *set* $\Rightarrow$ $'a$ *Distance* $\Rightarrow$ *bool* **where**
  distance S d $\equiv \forall\ x\ y.\ x \in S \land y \in S \longrightarrow d\ x\ x = 0 \land 0 \leq d\ x\ y$

4.1.2 Conditions

```
definition symmetric :: 'a set ⇒ 'a Distance ⇒ bool where
  symmetric S d ≡ ∀ x y. x ∈ S ∧ y ∈ S ⟶ d x y = d y x

definition triangle-ineq :: 'a set ⇒ 'a Distance ⇒ bool where
  triangle-ineq S d ≡ ∀ x y z. x ∈ S ∧ y ∈ S ∧ z ∈ S ⟶ d x z ≤ d x y + d y z

definition eq-if-zero :: 'a set ⇒ 'a Distance ⇒ bool where
  eq-if-zero S d ≡ ∀ x y. x ∈ S ∧ y ∈ S ⟶ d x y = 0 ⟶ x = y

definition vote-distance :: ('a Vote set ⇒ 'a Vote Distance ⇒ bool) ⇒
        'a Vote Distance ⇒ bool where
  vote-distance π d ≡ π {(A, p). linear-order-on A p ∧ finite A} d

definition election-distance :: (('a, 'v) Election set ⇒
        ('a, 'v) Election Distance ⇒ bool) ⇒
            ('a, 'v) Election Distance ⇒ bool where
  election-distance π d ≡ π {(A, V, p). finite-profile V A p} d
```

4.1.3 Standard-Distance Property

```
definition standard :: ('a, 'v) Election Distance ⇒ bool where
 standard d ≡
   ∀ A A' V V' p p'. A ≠ A' ∨ V ≠ V' ⟶ d (A, V, p) (A', V', p') = ∞
```

4.1.4 Auxiliary Lemmas

```
fun arg-min-set :: ('b ⇒ 'a :: ord) ⇒ 'b set ⇒ 'b set where
  arg-min-set f A = Collect (is-arg-min f (λ a. a ∈ A))

lemma arg-min-subset:
  fixes
    B :: 'b set and
    f :: 'b ⇒ 'a :: ord
  shows arg-min-set f B ⊆ B
  unfolding arg-min-set.simps is-arg-min-def
  by safe

lemma sum-monotone:
  fixes
    A :: 'a set and
    f g :: 'a ⇒ int
  assumes ∀ a ∈ A. f a ≤ g a
  shows (∑ a ∈ A. f a) ≤ (∑ a ∈ A. g a)
  using assms
proof (induction A rule: infinite-finite-induct)
  case (infinite A)
  fix A :: 'a set
  show ?case
```

```
    using infinite
    by simp
next
  case empty
  show ?case
    by simp
next
  case (insert x F)
  fix
    x :: 'a and
    F :: 'a set
  show ?case
    using insert
    by simp
qed

lemma distrib:
  fixes
    A :: 'a set and
    f g :: 'a ⇒ int
  shows (∑ a ∈ A. f a) + (∑ a ∈ A. g a) = (∑ a ∈ A. f a + g a)
  using sum.distrib
  by metis

lemma distrib-ereal:
  fixes
    A :: 'a set and
    f g :: 'a ⇒ int
  shows ereal (real-of-int ((∑ a ∈ A. (f :: 'a ⇒ int) a) + (∑ a ∈ A. g a))) =
    ereal (real-of-int ((∑ a ∈ A. (f a) + (g a))))
  using distrib
  by metis

lemma uneq-ereal:
  fixes x y :: int
  assumes x ≤ y
  shows ereal (real-of-int x) ≤ ereal (real-of-int y)
  using assms
  by simp
```

4.1.5 Swap Distance

```
fun neq-ord :: 'a Preference-Relation ⇒ 'a Preference-Relation ⇒
       'a ⇒ 'a ⇒ bool where
  neq-ord r s a b = ((a ≼r b ∧ b ≼s a) ∨ (b ≼r a ∧ a ≼s b))

fun pairwise-disagreements :: 'a set ⇒ 'a Preference-Relation ⇒
       'a Preference-Relation ⇒ ('a × 'a) set where
  pairwise-disagreements A r s = {(a, b) ∈ A × A. a ≠ b ∧ neq-ord r s a b}
```

```
fun pairwise-disagreements' :: 'a set ⇒ 'a Preference-Relation ⇒
        'a Preference-Relation ⇒ ('a × 'a) set where
  pairwise-disagreements' A r s =
      Set.filter (λ (a, b). a ≠ b ∧ neq-ord r s a b) (A × A)

lemma set-eq-filter:
  fixes
    X :: 'a set and
    P :: 'a ⇒ bool
  shows {x ∈ X. P x} = Set.filter P X
  by auto

lemma pairwise-disagreements-eq[code]: pairwise-disagreements = pairwise-disagreements'
  unfolding pairwise-disagreements.simps pairwise-disagreements'.simps
  by fastforce

fun swap :: 'a Vote Distance where
  swap (A, r) (A', r') =
    (if A = A'
    then card (pairwise-disagreements A r r')
    else ∞)

lemma swap-case-infinity:
  fixes x y :: 'a Vote
  assumes alts-𝒱 x ≠ alts-𝒱 y
  shows swap x y = ∞
  using assms
  by (induction rule: swap.induct, simp)

lemma swap-case-fin:
  fixes x y :: 'a Vote
  assumes alts-𝒱 x = alts-𝒱 y
  shows swap x y = card (pairwise-disagreements (alts-𝒱 x) (pref-𝒱 x) (pref-𝒱 y))
  using assms
  by (induction rule: swap.induct, simp)
```

4.1.6 Spearman Distance

```
fun spearman :: 'a Vote Distance where
  spearman (A, x) (A', y) =
    (if A = A'
    then ∑ a ∈ A. abs (int (rank x a) − int (rank y a))
    else ∞)

lemma spearman-case-inf:
  fixes x y :: 'a Vote
  assumes alts-𝒱 x ≠ alts-𝒱 y
  shows spearman x y = ∞
  using assms
  by (induction rule: spearman.induct, simp)

lemma spearman-case-fin:
  fixes x y :: 'a Vote
  assumes alts-𝒱 x = alts-𝒱 y
  shows spearman x y =
    (∑ a ∈ alts-𝒱 x. abs (int (rank (pref-𝒱 x) a) − int (rank (pref-𝒱 y) a)))
  using assms
  by (induction rule: spearman.induct, simp)
```

4.1.7 Properties

Distances that are invariant under specific relations induce symmetry properties in distance rationalized voting rules.

Definitions

```
fun total-invariance_𝒟 :: 'x Distance ⇒ 'x rel ⇒ bool where
  total-invariance_𝒟 d rel = is-symmetry (tup d) (Invariance (product rel))

fun invariance_𝒟 :: 'y Distance ⇒ 'x set ⇒ 'y set ⇒
        ('x, 'y) binary-fun ⇒ bool where
  invariance_𝒟 d X Y φ = is-symmetry (tup d) (Invariance (equivariance X Y φ))

definition distance-anonymity :: ('a, 'v) Election Distance ⇒ bool where
  distance-anonymity d ≡
    ∀ A A' V V' p p' π :: ('v ⇒ 'v).
      (bij π ⟶
        (d (A, V, p) (A', V', p')) =
          (d (rename π (A, V, p))) (rename π (A', V', p')))

fun distance-anonymity' :: ('a, 'v) Election set ⇒
        ('a, 'v) Election Distance ⇒ bool where
  distance-anonymity' X d = invariance_𝒟 d (carrier bijection_𝒱𝒢) X (φ-anon X)

fun distance-neutrality :: ('a, 'v) Election set ⇒
        ('a, 'v) Election Distance ⇒ bool where
  distance-neutrality X d = invariance_𝒟 d (carrier bijection_𝒜𝒢) X (φ-neutral X)

fun distance-reversal-symmetry :: ('a, 'v) Election set ⇒
        ('a, 'v) Election Distance ⇒ bool where
  distance-reversal-symmetry X d =
        invariance_𝒟 d (carrier reversal_𝒢) X (φ-reverse X)

definition distance-homogeneity' :: ('a, 'v :: linorder) Election set ⇒
        ('a, 'v) Election Distance ⇒ bool where
  distance-homogeneity' X d ≡ total-invariance_𝒟 d (homogeneity_ℛ' X)
```

```
definition distance-homogeneity :: ('a, 'v) Election set ⇒
        ('a, 'v) Election Distance ⇒ bool where
  distance-homogeneity X d ≡ total-invariance𝒟 d (homogeneityℛ X)
```

Auxiliary Lemmas

```
lemma rewrite-total-invariance𝒟:
  fixes
    d :: 'x Distance and
    r :: 'x rel
  shows total-invariance𝒟 d r = (∀ (x, y) ∈ r. ∀ (a, b) ∈ r. d a x = d b y)
proof (unfold total-invariance𝒟.simps is-symmetry.simps product.simps, safe)
  fix a b x y :: 'x
  assume
    ∀ x y. (x, y) ∈ {(p, p').
      (fst p, fst p') ∈ r ∧ (snd p, snd p') ∈ r}
        ⟶ tup d x = tup d y and
    (a, b) ∈ r and
    (x, y) ∈ r
  thus d a x = d b y
    unfolding total-invariance𝒟.simps is-symmetry.simps
    by simp
next
  fix a b x y :: 'x
  assume
    ∀ (x, y) ∈ r. ∀ (a, b) ∈ r. d a x = d b y and
    (fst (x, a), fst (y, b)) ∈ r and
    (snd (x, a), snd (y, b)) ∈ r
  hence d x a = d y b
    by auto
  thus tup d (x, a) = tup d (y, b)
    by simp
qed

lemma rewrite-invariance𝒟:
  fixes
    d :: 'y Distance and
    X :: 'x set and
    Y :: 'y set and
    φ :: ('x, 'y) binary-fun
  shows invariance𝒟 d X Y φ =
            (∀ x ∈ X. ∀ y ∈ Y. ∀ z ∈ Y. d y z = d (φ x y) (φ x z))
proof (unfold invariance𝒟.simps is-symmetry.simps equivariance.simps, safe)
  fix
    x :: 'x and
    y z :: 'y
  assume
    x ∈ X and
    y ∈ Y and
    z ∈ Y and
    ∀ x y. (x, y) ∈ {((u, v), x, y). (u, v) ∈ Y × Y
                    ∧ (∃ z ∈ X. x = φ z u ∧ y = φ z v)}
          ⟶ tup d x = tup d y
  thus d y z = d (φ x y) (φ x z)
    by fastforce
next
  fix
    x :: 'x and
    a b :: 'y
  assume
    ∀ x ∈ X. ∀ y ∈ Y. ∀ z ∈ Y. d y z = d (φ x y) (φ x z) and
    x ∈ X and
    a ∈ Y and
    b ∈ Y
  hence d a b = d (φ x a) (φ x b)
    by blast
  thus tup d (a, b) = tup d (φ x a, φ x b)
    by simp
qed

lemma invar-dist-image:
  fixes
    d :: 'y Distance and
    G :: 'x monoid and
    Y Y' :: 'y set and
    φ :: ('x, 'y) binary-fun and
    y :: 'y and
    g :: 'x
  assumes
    invar-d: invariance_𝒟 d (carrier G) Y φ and
    Y'-in-Y: Y' ⊆ Y and
    action-φ: group-action G Y φ and
    g-carrier: g ∈ carrier G and
    y-in-Y: y ∈ Y
  shows d (φ g y) ` (φ g) ` Y' = d y ` Y'
proof (safe)
  fix y' :: 'y
  assume y'-in-Y': y' ∈ Y'
  hence ((y, y'), ((φ g y), (φ g y'))) ∈ equivariance (carrier G) Y φ
    using Y'-in-Y y-in-Y g-carrier
    unfolding equivariance.simps
    by blast
  hence eq-dist: tup d ((φ g y), (φ g y')) = tup d (y, y')
    using invar-d
    unfolding invariance_𝒟.simps
    by fastforce
  thus d (φ g y) (φ g y') ∈ d y ` Y'
    using y'-in-Y'
    by simp
  have φ g y′ ∈ φ g ‘ Y′
    using y′-in-Y′
    by simp
  thus d y y′ ∈ d (φ g y) ‘ φ g ‘ Y′
    using eq-dist
    by (simp add: rev-image-eqI)
qed

lemma swap-neutral: invariance_𝒟 swap (carrier bijection_𝒜𝒢)
                        UNIV (λ π (A, q). (π ‘ A, rel-rename π q))
proof (unfold rewrite-invariance_𝒟, safe)
  fix
    π :: ′a ⇒ ′a and
    A A′ :: ′a set and
    q q′ :: ′a rel
  assume π ∈ carrier bijection_𝒜𝒢
  hence bij-π: bij π
    unfolding bijection_𝒜𝒢-def
    using rewrite-carrier
    by blast
  show swap (A, q) (A′, q′) =
        swap (π ‘ A, rel-rename π q) (π ‘ A′, rel-rename π q′)
  proof (cases A = A′)
    let ?f = (λ (a, b). (π a, π b))
    let ?swap-set = {(a, b) ∈ A × A. a ≠ b ∧ neq-ord q q′ a b}
    let ?swap-set′ =
      {(a, b) ∈ π ‘ A × π ‘ A. a ≠ b
          ∧ neq-ord (rel-rename π q) (rel-rename π q′) a b}
    let ?rel = {(a, b) ∈ A × A. a ≠ b ∧ neq-ord q q′ a b}
    case True
    hence π ‘ A = π ‘ A′
      by simp
    hence swap (π ‘ A, rel-rename π q) (π ‘ A′, rel-rename π q′) = card ?swap-set′
      by simp
    moreover have bij-betw ?f ?swap-set ?swap-set′
    proof (unfold bij-betw-def inj-on-def, intro conjI impI ballI)
      fix x y :: ′a × ′a
      assume
        x ∈ ?swap-set and
        y ∈ ?swap-set and
        ?f x = ?f y
      hence
        π (fst x) = π (fst y) and
        π (snd x) = π (snd y)
        by auto
      hence
        fst x = fst y and
        snd x = snd y
        using bij-π bij-pointE
        by (metis, metis)
      thus x = y
        using prod.expand
        by metis
    next
      show ?f ‘ ?swap-set = ?swap-set′
      proof
        have ∀ a b. (a, b) ∈ A × A ⟶ (π a, π b) ∈ π ‘ A × π ‘ A
          by simp
        moreover have ∀ a b. a ≠ b ⟶  π a ≠ π b
          using bij-π bij-pointE
          by metis
        moreover have
          ∀ a b. neq-ord q q′ a b
            ⟶ neq-ord (rel-rename π q) (rel-rename π q′) (π a) (π b)
          unfolding neq-ord.simps rel-rename.simps
          by auto
        ultimately show ?f ‘ ?swap-set ⊆ ?swap-set′
          by auto
      next
        have ∀ a b. (a, b) ∈ (rel-rename π q) ⟶ (the-inv π a, the-inv π b) ∈ q
          unfolding rel-rename.simps
          using bij-π bij-is-inj the-inv-f-f
          by fastforce
        moreover have
          ∀ a b. (a, b) ∈ (rel-rename π q′) ⟶ (the-inv π a, the-inv π b) ∈ q′
          unfolding rel-rename.simps
          using bij-π bij-is-inj the-inv-f-f
          by fastforce
        ultimately have
          ∀ a b. neq-ord (rel-rename π q) (rel-rename π q′) a b
                  ⟶ neq-ord q q′ (the-inv π a) (the-inv π b)
          by simp
        moreover have
          ∀ a b. (a, b) ∈ π ‘ A × π ‘ A ⟶ (the-inv π a, the-inv π b) ∈ A × A
          using bij-π bij-is-inj f-the-inv-into-f inj-image-mem-iff
          by fastforce
        moreover have ∀ a b. a ≠ b ⟶ the-inv π a ≠ the-inv π b
          using bij-π UNIV-I bij-betw-imp-surj bij-is-inj f-the-inv-into-f
          by metis
        ultimately have
          ∀ a b. (a, b) ∈ ?swap-set′ ⟶ (the-inv π a, the-inv π b) ∈ ?swap-set
          by blast
        moreover have ∀ a b. (a, b) = ?f (the-inv π a, the-inv π b)
          using f-the-inv-into-f-bij-betw bij-π
          by fastforce
        ultimately show ?swap-set′ ⊆ ?f ‘ ?swap-set
          by blast
      qed
    qed
    moreover have card ?swap-set = swap (A, q) (A′, q′)
      using True
      by simp
    ultimately show ?thesis
      by (simp add: bij-betw-same-card)
  next
    case False
    hence π ‘ A ≠ π ‘ A′
      using bij-π bij-is-inj inj-image-eq-iff
      by metis
    thus ?thesis
      using False
      by simp
  qed
qed

end
```

4.2 Votewise Distance

```
theory Votewise-Distance
  imports Social-Choice-Types/Norm
          Distance
begin
```

Votewise distances are a natural class of distances on elections which depend on the submitted votes in a simple and transparent manner. They are formed by using any distance d on individual orders and combining the components with a norm on $\mathbb{R}^n$.

4.2.1 Definition

```
fun votewise-distance :: ′a Vote Distance ⇒ Norm ⇒
        (′a, ′v :: linorder) Election Distance where
  votewise-distance d n (A, V, p) (A′, V′, p′) =
    (if (finite V) ∧ V = V′ ∧ (V ≠ {} ∨ A = A′)
    then n (map2 (λ q q′. d (A, q) (A′, q′)) (to-list V p) (to-list V′ p′))
    else ∞)
```

4.2.2 Inference Rules

```
lemma symmetric-norm-inv-under-map-permute:
  fixes
    d :: ′a Vote Distance and
    n :: Norm and
    A A′ :: ′a set and
    φ :: nat ⇒ nat and
    p p′ :: (′a Preference-Relation) list
  assumes
    perm: φ permutes {0 ..< length p} and
    len-eq: length p = length p′ and
    sym-n: symmetry n
  shows n (map2 (λ q q′. d (A, q) (A′, q′)) p p′) =
      n (map2 (λ q q′. d (A, q) (A′, q′)) (permute-list φ p) (permute-list φ p′))
proof −
  have length (map2 (λ x y. d (A, x) (A′, y)) p p′) = length p
    using len-eq
    by simp
  hence n (map2 (λ q q′. d (A, q) (A′, q′)) p p′) =
      n (permute-list φ (map2 (λ x y. d (A, x) (A′, y)) p p′))
    using perm sym-n mset-permute-list atLeast-upt
    unfolding symmetry-def
    by fastforce
  thus ?thesis
     using perm len-eq atLeast-upt permute-list-map[of - - λ (q, q′). d (A, q) (A′,
q′)]
    by (simp add: permute-list-zip)
qed

lemma permute-invariant-under-map:
  fixes l l′ :: ′a list
  assumes l <~~> l′
  shows map f l <~~> map f l′
  using assms
  by simp

lemma linorder-rank-injective:
  fixes
    V :: ′v :: linorder set and
    v v′ :: ′v
  assumes
    v-in-V: v ∈ V and
    v′-in-V: v′ ∈ V and
    v′-neq-v: v′ ≠ v and
    fin-V: finite V
  shows card {x ∈ V. x < v} ≠ card {x ∈ V. x < v′}
proof −
  have v < v′ ∨ v′ < v
    using v′-neq-v linorder-less-linear
    by metis
  hence {x ∈ V. x < v} ⊂ {x ∈ V. x < v′} ∨ {x ∈ V. x < v′} ⊂ {x ∈ V. x < v}
    using v-in-V v′-in-V dual-order.strict-trans
    by blast
  thus ?thesis
    using assms sorted-list-of-set-nth-equals-card
    by (metis (full-types))
qed

lemma permute-invariant-under-coinciding-funs:
  fixes
    l :: 'v list and
    π₁ π₂ :: nat ⇒ nat
  assumes ∀ i < length l. π₁ i = π₂ i
  shows permute-list π₁ l = permute-list π₂ l
  using assms
  unfolding permute-list-def
  by simp

lemma symmetric-norm-imp-distance-anonymous:
  fixes
    d :: 'a Vote Distance and
    n :: Norm
  assumes symmetry n
  shows distance-anonymity (votewise-distance d n)
proof (unfold distance-anonymity-def, safe)
  fix
    A A' :: 'a set and
    V V' :: 'v :: linorder set and
    p p' :: ('a, 'v) Profile and
    π :: 'v ⇒ 'v
  let ?rn1 = rename π (A, V, p) and
      ?rn2 = rename π (A', V', p') and
      ?rn-V = π ‘ V and
      ?rn-V' = π ‘ V' and
      ?rn-p = p ∘ (the-inv π) and
      ?rn-p' = p' ∘ (the-inv π) and
      ?len = length (to-list V p) and
      ?sl-V = sorted-list-of-set V
  let ?perm = λ i. card {v ∈ ?rn-V. v < π (?sl-V!i)} and
    — Use a total permutation function in order to apply facts such as mset-permute-list.
      ?perm-total = λ i. if i < ?len
                          then card {v ∈ ?rn-V. v < π (?sl-V!i)}
                          else i
  assume bij-π: bij π
  show votewise-distance d n (A, V, p) (A', V', p') =
          votewise-distance d n ?rn1 ?rn2
  proof (cases finite V ∧ V = V' ∧ (V ≠ {} ∨ A = A'))
    case False
      — Case: Both distances are infinite.
    hence votewise-distance d n (A, V, p) (A', V', p') = ∞
      by auto
    moreover have infinite V ⟶ infinite ?rn-V
```

```
      using bij-π bij-betw-finite bij-betw-subset subset-UNIV
      by metis
    moreover have V ≠ V′ ⟶ ?rn-V ≠ ?rn-V′
      using bij-π inj-image-mem-iff subsetI subset-antisym
      unfolding bij-def
      by metis
    ultimately show votewise-distance d n (A, V, p) (A′, V′, p′) =
            votewise-distance d n ?rn1 ?rn2
      using False
      by auto
  next
    case True
      — Case: Both distances are finite.
    have lengths-eq: ?len = length (to-list V′ p′)
      using True
      by simp
    have rn-V-permutes: to-list V p = permute-list ?perm (to-list ?rn-V ?rn-p)
      using assms to-list-permutes-under-bij bij-π to-list-permutes-under-bij
      unfolding comp-def
      by (metis (no-types))
    hence len-V-rn-V-eq: ?len = length (to-list ?rn-V ?rn-p)
      by simp
    hence permute-list ?perm (to-list ?rn-V ?rn-p) =
            permute-list ?perm-total (to-list ?rn-V ?rn-p)
      using permute-invariant-under-coinciding-funs[of to-list ?rn-V ?rn-p]
      by presburger
    hence rn-list-perm-list-V:
      to-list V p = permute-list ?perm-total (to-list ?rn-V ?rn-p)
      using rn-V-permutes
      by metis
    have to-list V′ p′ = permute-list ?perm (to-list ?rn-V′ ?rn-p′)
      unfolding comp-def
      using True bij-π to-list-permutes-under-bij
      by (metis (no-types))
    hence rn-list-perm-list-V′:
      (to-list V′ p′) = permute-list ?perm-total (to-list ?rn-V′ ?rn-p′)
        using lengths-eq permute-invariant-under-coinciding-funs[of to-list ?rn-V′
?rn-p′]
      by fastforce
    have ?perm-total permutes {0 ..< ?len}
    proof −
      have ∀ i j. i < ?len ∧ j < ?len ∧ i ≠ j
              ⟶ π ((sorted-list-of-set V)!i) ≠ π ((sorted-list-of-set V)!j)
        using True bij-π bij-pointE nth-eq-iff-index-eq length-map
            sorted-list-of-set.distinct-sorted-key-list-of-set to-list.elims
        by (metis (mono-tags, opaque-lifting))
      moreover have in-bnds-imp-img-el:
        ∀ i. i < ?len ⟶ π ((sorted-list-of-set V)!i) ∈ π ‘ V
        using True image-eqI length-map nth-mem to-list.simps
            sorted-list-of-set.set-sorted-key-list-of-set
        by (metis (no-types))
      ultimately have
        ∀ i < ?len. ∀ j < ?len. ?perm-total i = ?perm-total j ⟶ i = j
        using True linorder-rank-injective Collect-cong finite-imageI
        by (metis (no-types, lifting))
      hence inj: inj-on ?perm-total {0 ..< ?len}
        unfolding inj-on-def
        by simp
      have ∀ v' ∈ π ‘ V. card {v ∈ π ‘ V. v < v'} < card (π ‘ V)
        using True card-seteq finite-imageI less-irrefl linorder-not-le
            mem-Collect-eq subsetI
        by (metis (no-types, lifting))
      moreover have ∀ i < ?len. π ((sorted-list-of-set V)!i) ∈ π ‘ V
        using in-bnds-imp-img-el
        by simp
      moreover have card (π ‘ V) = card V
        using bij-π bij-betw-same-card bij-betw-subset top-greatest
        by metis
      moreover have card V = ?len
        by simp
      ultimately have
        ∀ i. i < ?len ⟶ ?perm-total i ∈ {0 ..< ?len}
        using atLeast0LessThan lessThan-iff
        by (metis (full-types))
      hence ?perm-total ‘ {0 ..< ?len} ⊆ {0 ..< ?len}
        by force
      hence bij-betw ?perm-total {0 ..< ?len} {0 ..< ?len}
       using inj card-image card-subset-eq atLeast0LessThan finite-atLeastLessThan
        unfolding bij-betw-def
        by blast
      thus ?thesis
        using atLeast0LessThan bij-imp-permutes
        by fastforce
    qed
    hence votewise-distance d n ?rn1 ?rn2 =
            n (map2 (λ q q'. d (A, q) (A', q'))
                  (permute-list ?perm-total (to-list ?rn-V ?rn-p))
                  (permute-list ?perm-total (to-list ?rn-V' ?rn-p')))
      using symmetric-norm-inv-under-map-permute[of - to-list ?rn-V ?rn-p]
            True assms len-V-rn-V-eq
      by force
    also have ... = n (map2 (λ q q'. d (A, q) (A', q')) (to-list V p) (to-list V' p'))
      using rn-list-perm-list-V rn-list-perm-list-V'
      by presburger
    finally show
     votewise-distance d n (A, V, p) (A', V', p') = votewise-distance d n ?rn1 ?rn2
      using True
      by force
  qed
qed

lemma neutral-dist-imp-neutral-votewise-dist:
  fixes
    d :: 'a Vote Distance and
    n :: Norm
  defines vote-action ≡ λ π (A, q). (π ‘ A, rel-rename π q)
  assumes invariance_𝒟 d (carrier bijection_𝒜𝒢) UNIV vote-action
  shows distance-neutrality well-formed-elections (votewise-distance d n)
proof (unfold distance-neutrality.simps rewrite-invariance_𝒟, safe)
  fix
    A A' :: 'a set and
    V V' :: 'v :: linorder set and
    p p' :: ('a, 'v) Profile and
    π :: 'a ⇒ 'a
  assume
    carrier: π ∈ carrier bijection_𝒜𝒢 and
    valid: (A, V, p) ∈ well-formed-elections and
    valid': (A', V', p') ∈ well-formed-elections
  hence bij-π: bij π
    unfolding bijection_𝒜𝒢-def
    using rewrite-carrier
    by blast
  thus votewise-distance d n (A, V, p) (A', V', p') =
          votewise-distance d n
            (φ-neutral well-formed-elections π (A, V, p))
              (φ-neutral well-formed-elections π (A', V', p'))
  proof (cases finite V ∧ V = V' ∧ (V ≠ {} ∨ A = A'))
    case True
    hence finite V ∧ V = V' ∧ (V ≠ {} ∨ π ‘ A = π ‘ A')
      by metis
    hence votewise-distance d n
            (φ-neutral well-formed-elections π (A, V, p))
                (φ-neutral well-formed-elections π (A', V', p')) =
        n (map2 (λ q q'. d (π ‘ A, q) (π ‘ A', q'))
          (to-list V (rel-rename π ∘ p)) (to-list V' (rel-rename π ∘ p')))
      using valid valid'
      by auto
    also have
      ... =
      n (map2 (λ q q'. d (π ‘ A, q) (π ‘ A', q'))
        (map (rel-rename π) (to-list V p)) (map (rel-rename π) (to-list V' p')))
      using to-list-comp
      by metis
    also have
      ... = n (map2 (λ q q'. d (π ‘ A, rel-rename π q) (π ‘ A', rel-rename π q'))
              (to-list V p) (to-list V' p'))
      unfolding map-helper
      by simp
    also have
      ... = (n (map2 (λ q q'. d (A, q) (A', q')) (to-list V p) (to-list V' p')))
      using rewrite-invariance_D[of d - UNIV vote-action] assms carrier
            UNIV-I case-prod-conv
      unfolding vote-action-def
      by (metis (no-types, lifting))
    finally have votewise-distance d n
        (φ-neutral well-formed-elections π (A, V, p))
              (φ-neutral well-formed-elections π (A', V', p')) =
        n (map2 (λ q q'. d (A, q) (A', q')) (to-list V p) (to-list V' p'))
      by simp
    thus ?thesis
      using True
      by auto
  next
    case False
    hence ¬ (finite V ∧ V = V' ∧ (V ≠ {} ∨ π ` A = π ` A'))
      using bij-π bij-is-inj inj-image-eq-iff
      by metis
    thus ?thesis
      using False valid valid'
      by force
  qed
qed

end
```

4.3 Consensus

```
theory Consensus
  imports Social-Choice-Types/Voting-Symmetry
begin
```

An election consisting of a set of alternatives and preferential votes for each voter (a profile) is a consensus if it has an undisputed winner reflecting a certain concept of fairness in the society.

4.3.1 Definition

```
type-synonym ('a, 'v) Consensus = ('a, 'v) Election ⇒ bool
```

4.3.2 Consensus Conditions

Nonempty alternative set.

```
fun nonempty-set_C :: ('a, 'v) Consensus where
```

```
  nonempty-set𝒞 (A, V, p) = (A ≠ {})
```

Nonempty profile, i.e., nonempty voter set. Note that this is also true if p(v) = holds for all voters v in V.

```
fun nonempty-profile𝒞 :: ('a, 'v) Consensus where
  nonempty-profile𝒞 (A, V, p) = (V ≠ {})
```

Equal top ranked alternatives.

```
fun equal-top𝒞' :: 'a ⇒ ('a, 'v) Consensus where
  equal-top𝒞' a (A, V, p) = (a ∈ A ∧ (∀ v ∈ V. above (p v) a = {a}))

fun equal-top𝒞 :: ('a, 'v) Consensus where
  equal-top𝒞 c = (∃ a. equal-top𝒞' a c)
```

Equal votes.

```
fun equal-vote𝒞' :: 'a Preference-Relation ⇒ ('a, 'v) Consensus where
  equal-vote𝒞' r (A, V, p) = (∀ v ∈ V. (p v) = r)

fun equal-vote𝒞 :: ('a, 'v) Consensus where
  equal-vote𝒞 c = (∃ r. equal-vote𝒞' r c)
```

Unanimity condition.

```
fun unanimity𝒞 :: ('a, 'v) Consensus where
  unanimity𝒞 c = (nonempty-set𝒞 c ∧ nonempty-profile𝒞 c ∧ equal-top𝒞 c)
```

Strong unanimity condition.

```
fun strong-unanimity𝒞 :: ('a, 'v) Consensus where
  strong-unanimity𝒞 c = (nonempty-set𝒞 c ∧ nonempty-profile𝒞 c ∧ equal-vote𝒞 c)
```

4.3.3 Properties

```
definition consensus-anonymity :: ('a, 'v) Consensus ⇒ bool where
  consensus-anonymity c ≡
    (∀ A V p π :: ('v ⇒ 'v).
        bij π ⟶
          (let (A', V', q) = (rename π (A, V, p)) in
            profile V A p ⟶ profile V' A' q
            ⟶ c (A, V, p) ⟶ c (A', V', q)))

fun consensus-neutrality :: ('a, 'v) Election set ⇒ ('a, 'v) Consensus ⇒ bool where
  consensus-neutrality X c = is-symmetry c (Invariance (neutralityℛ X))
```

4.3.4 Auxiliary Lemmas

```
lemma cons-anon-conj:
  fixes c c' :: ('a, 'v) Consensus
  assumes
    consensus-anonymity c and
    consensus-anonymity c′
  shows consensus-anonymity (λ e. c e ∧ c′ e)
proof (unfold consensus-anonymity-def Let-def, clarify)
  fix
    A A′ :: ′a set and
    V V′ :: ′v set and
    p q :: (′a, ′v) Profile and
    π :: ′v ⇒ ′v
  assume
    bij-π: bij π and
    renamed: rename π (A, V, p) = (A′, V′, q) and
    prof: profile V A p
  hence profile V′ A′ q
    using rename-sound fst-conv rename.simps
    by metis
  moreover assume
    c (A, V, p) and
    c′ (A, V, p)
  ultimately show c (A′, V′, q) ∧ c′ (A′, V′, q)
    using bij-π renamed assms prof
    unfolding consensus-anonymity-def
    by auto
qed

theorem cons-conjunction-invariant:
  fixes
    ℭ :: (′a, ′v) Consensus set and
    rel :: (′a, ′v) Election rel
  defines C ≡ λ E. ∀ C′ ∈ ℭ. C′ E
  assumes ∀ C′. C′ ∈ ℭ ⟶ is-symmetry C′ (Invariance rel)
  shows is-symmetry C (Invariance rel)
proof (unfold is-symmetry.simps, intro allI impI)
  fix E E′ :: (′a, ′v) Election
  assume (E, E′) ∈ rel
  hence ∀ C′ ∈ ℭ. C′ E = C′ E′
    using assms
    unfolding is-symmetry.simps
    by blast
  thus C E = C E′
    unfolding C-def
    by blast
qed

lemma cons-anon-invariant:
  fixes
    c :: (′a, ′v) Consensus and
    A A′ :: ′a set and
    V V′ :: ′v set and
    p q :: (′a, ′v) Profile and
    π :: 'v ⇒ 'v
  assumes
    anon: consensus-anonymity c and
    bij-π: bij π and
    prof-p: profile V A p and
    renamed: rename π (A, V, p) = (A', V', q) and
    cond-c: c (A, V, p)
  shows c (A', V', q)
proof −
  have profile V' A' q
    using rename-sound bij-π renamed prof-p
    by fastforce
  thus ?thesis
    using anon cond-c renamed rename-finite bij-π prof-p
    unfolding consensus-anonymity-def Let-def
    by auto
qed

lemma ex-anon-cons-imp-cons-anonymous:
  fixes
    b :: ('a, 'v) Consensus and
    b':: 'b ⇒ ('a, 'v) Consensus
  assumes
    general-cond-b: b = (λ E. ∃ x. b' x E) and
    all-cond-anon: ∀ x. consensus-anonymity (b' x)
  shows consensus-anonymity b
proof (unfold consensus-anonymity-def Let-def, safe)
  fix
    A A' :: 'a set and
    V V' :: 'v set and
    p q :: ('a, 'v) Profile and
    π :: 'v ⇒ 'v
  assume
    bij-π: bij π and
    cond-b: b (A, V, p) and
    prof-p: profile V A p and
    renamed: rename π (A, V, p) = (A', V', q)
  have ∃ x. b' x (A, V, p)
    using cond-b general-cond-b
    by simp
  then obtain x :: 'b where
    b' x (A, V, p)
    by blast
  moreover have consensus-anonymity (b' x)
    using all-cond-anon
    by simp
  moreover have profile V' A' q
    using prof-p renamed bij-π rename-sound
    by fastforce
  ultimately have b' x (A', V', q)
    using all-cond-anon bij-π prof-p renamed
    unfolding consensus-anonymity-def
    by auto
  hence ∃ x. b' x (A', V', q)
    by metis
  thus b (A', V', q)
    using general-cond-b
    by simp
qed
```

4.3.5 Theorems

Anonymity

```
lemma nonempty-set-cons-anonymous: consensus-anonymity nonempty-set_𝒞
  unfolding consensus-anonymity-def
  by simp

lemma nonempty-profile-cons-anonymous: consensus-anonymity nonempty-profile_𝒞
proof (unfold consensus-anonymity-def Let-def, clarify)
  fix
    A A' :: 'a set and
    V V' :: 'v set and
    p q :: ('a, 'v) Profile and
    π :: 'v ⇒ 'v
  assume
    bij-π: bij π and
    renamed: rename π (A, V, p) = (A', V', q)
  hence card V = card V'
    using rename.simps Pair-inject bij-betw-same-card
          bij-betw-subset top-greatest
    by (metis (mono-tags, lifting))
  moreover assume nonempty-profile_𝒞 (A, V, p)
  ultimately show nonempty-profile_𝒞 (A', V', q)
    using length-0-conv renamed
    unfolding nonempty-profile_𝒞.simps
    by auto
qed

lemma equal-top-cons'-anonymous:
  fixes a :: 'a
  shows consensus-anonymity (equal-top_𝒞' a)
proof (unfold consensus-anonymity-def Let-def, clarify)
  fix
    A A' :: 'a set and
    V V' :: 'v set and
    p q :: ('a, 'v) Profile and
    π :: 'v ⇒ 'v
  assume
```

```
    bij-π: bij π and
    prof-p: profile V A p and
    renamed: rename π (A, V, p) = (A′, V′, q) and
    top-cons-a: equal-top𝒞′ a (A, V, p)
  have ∀ v′ ∈ V′. q v′ = p ((the-inv π) v′)
    using renamed
    by auto
  moreover have ∀ v′ ∈ V′. (the-inv π) v′ ∈ V
    using bij-π renamed rename.simps bij-is-inj
          f-the-inv-into-f-bij-betw inj-image-mem-iff
    by fastforce
  moreover have winner: ∀ v ∈ V. above (p v) a = {a}
    using top-cons-a
    by simp
  ultimately have ∀ v′ ∈ V′. above (q v′) a = {a}
    by simp
  moreover have a ∈ A
    using top-cons-a
    by simp
  ultimately show equal-top𝒞′ a (A′, V′, q)
    using renamed
    unfolding equal-top𝒞′.simps
    by simp
qed

lemma eq-top-cons-anon: consensus-anonymity equal-top𝒞
  using equal-top-cons′-anonymous
        ex-anon-cons-imp-cons-anonymous[of equal-top𝒞 equal-top𝒞′]
  by fastforce

lemma eq-vote-cons′-anonymous:
  fixes r :: ′a Preference-Relation
  shows consensus-anonymity (equal-vote𝒞′ r)
proof (unfold consensus-anonymity-def Let-def, clarify)
  fix
    A A′ :: ′a set and
    V V′ :: ′v set and
    p q :: (′a, ′v) Profile and
    π :: ′v ⇒ ′v
  assume
    bij-π: bij π and
    renamed: rename π (A, V, p) = (A′, V′, q)
  have ∀ v′ ∈ V′. (the-inv π) v′ ∈ V
    using bij-π renamed bij-is-inj inj-image-mem-iff
          f-the-inv-into-f-bij-betw
    by fastforce
  moreover assume equal-vote𝒞′ r (A, V, p)
  ultimately show equal-vote𝒞′ r (A′, V′, q)
    using renamed
  by force
qed

lemma eq-vote-cons-anonymous: consensus-anonymity equal-vote_𝒞
  unfolding equal-vote_𝒞.simps
  using eq-vote-cons'-anonymous ex-anon-cons-imp-cons-anonymous
  by blast
```

Neutrality

```
lemma nonempty-set_𝒞-neutral: consensus-neutrality well-formed-elections nonempty-set_𝒞
  unfolding well-formed-elections-def
  by auto

lemma nonempty-profile_𝒞-neutral: consensus-neutrality well-formed-elections nonempty-profile_𝒞
  unfolding well-formed-elections-def
  by auto

lemma equal-vote_𝒞-neutral: consensus-neutrality well-formed-elections equal-vote_𝒞
proof (unfold well-formed-elections-def consensus-neutrality.simps is-symmetry.simps,
       intro allI impI,
       unfold split-paired-all neutrality_ℛ.simps action-induced-rel.simps
       voters-ℰ.simps alternatives-ℰ.simps profile-ℰ.simps φ-neutral.simps
       extensional-continuation.simps equal-vote_𝒞.simps equal-vote_𝒞'.simps
       alternatives-rename.simps case-prod-unfold mem-Collect-eq fst-conv
       snd-conv mem-Sigma-iff conj-assoc If-def simp-thms, safe)
  fix
    A A' :: 'a set and
    V V' :: 'v set and
    p p' :: ('a, 'v) Profile and
    π :: 'a ⇒ 'a and
    r :: 'a rel
  assume
    profile V A p and
    (THE z.
        (profile V A p ⟶ z = (π ` A, V, rel-rename π ∘ p))
        ∧ (¬ profile V A p ⟶ z = undefined)) = (A', V', p')
  hence
    equal-voters: V' = V and
    perm-profile: p' = (λ x. {(π a, π b) | a b. (a, b) ∈ p x})
    unfolding comp-def
    by (simp, simp)
  have
    (∀ v ∈ V. p v = r)
      ⟶ (∃ r'. ∀ v ∈ V. {(π a, π b) | a b. (a, b) ∈ p v} = r')
    by simp
  {
    moreover assume ∀ v' ∈ V. p v' = r
    ultimately show ∃ r. ∀ v ∈ V'. p' v = r
```

```
      using equal-voters perm-profile
      by metis
  }
  assume π ∈ carrier bijection_𝒜𝒢
  hence bij π
    using rewrite-carrier
    unfolding bijection_𝒜𝒢-def
    by blast
  hence ∀ a. the-inv π (π a) = a
    using bij-is-inj the-inv-f-f
    by metis
  moreover have
    (∀ v ∈ V. {(π a, π b) | a b. (a, b) ∈ p v} = r) ⟶
      (∀ v ∈ V. {(the-inv π (π a), the-inv π (π b)) | a b. (a, b) ∈ p v} =
              {(the-inv π a, the-inv π b) | a b. (a, b) ∈ r})
    by fastforce
  ultimately have
    (∀ v ∈ V. {(π a, π b) | a b. (a, b) ∈ p v} = r) ⟶
      (∀ v ∈ V. {(a, b) | a b. (a, b) ∈ p v} =
             {(the-inv π a, the-inv π b) | a b. (a, b) ∈ r})
    by auto
  hence
    (∀ v' ∈ V. {(π a, π b) | a b. (a, b) ∈ p v'} = r)
      ⟶ (∃ r'. ∀ v' ∈ V. p v' = r')
    by simp
  moreover assume ∀ v' ∈ V'. p' v' = r
  ultimately show ∃ r'. ∀ v' ∈ V. p v' = r'
    using equal-voters perm-profile
    by metis
qed

lemma strong-unanimity_𝒞-neutral: consensus-neutrality
        well-formed-elections strong-unanimity_𝒞
  using nonempty-set_𝒞-neutral equal-vote_𝒞-neutral nonempty-profile_𝒞-neutral
        cons-conjunction-invariant[of
          {nonempty-set_𝒞, nonempty-profile_𝒞, equal-vote_𝒞}
          neutrality_ℛ well-formed-elections]
  unfolding strong-unanimity_𝒞.simps
  by fastforce

end
```

4.4 Electoral Module

theory *Electoral-Module*
  imports *Social-Choice-Types/Property-Interpretations*
begin

Electoral modules are the principal component type of the composable modules voting framework, as they are a generalization of voting rules in the sense of social choice functions. These are only the types used for electoral modules. Further restrictions are encompassed by the electoral-module predicate.

An electoral module does not need to make final decisions for all alternatives, but can instead defer the decision for some or all of them to other modules. Hence, electoral modules partition the received (possibly empty) set of alternatives into elected, rejected and deferred alternatives. In particular, any of those sets, e.g., the set of winning (elected) alternatives, may also be left empty, as long as they collectively still hold all the received alternatives. Just like a voting rule, an electoral module also receives a profile which holds the voters preferences, which, unlike a voting rule, consider only the (sub-)set of alternatives that the module receives.

4.4.1 Definition

An electoral module maps an election to a result. To enable currying, the Election type is not used here because that would require tuples.

type-synonym ($'a$, $'v$, $'r$) *Electoral-Module* = $'v$ *set* ⇒ $'a$ *set* ⇒
    ($'a$, $'v$) *Profile* ⇒ $'r$

fun *fun*$_{\mathcal{E}}$ :: ($'v$ *set* ⇒ $'a$ *set* ⇒ ($'a$, $'v$) *Profile* ⇒ $'r$) ⇒
    (($'a$, $'v$) *Election* ⇒ $'r$) **where**
  fun$_{\mathcal{E}}$ *m* = (λ *E*. *m* (*voters-*$\mathcal{E}$ *E*) (*alternatives-*$\mathcal{E}$ *E*) (*profile-*$\mathcal{E}$ *E*))

The next three functions take an electoral module and turn it into a function only outputting the elect, reject, or defer set respectively.

abbreviation *elect* :: ($'a$, $'v$, $'r$ *Result*) *Electoral-Module* ⇒ $'v$ *set* ⇒ $'a$ *set* ⇒
    ($'a$, $'v$) *Profile* ⇒ $'r$ *set* **where**
  elect m V A p ≡ *elect-r* (*m V A p*)

abbreviation *reject* :: ($'a$, $'v$, $'r$ *Result*) *Electoral-Module* ⇒ $'v$ *set* ⇒ $'a$ *set* ⇒
    ($'a$, $'v$) *Profile* ⇒ $'r$ *set* **where**
  reject m V A p ≡ *reject-r* (*m V A p*)

abbreviation *defer* :: ($'a$, $'v$, $'r$ *Result*) *Electoral-Module* ⇒ $'v$ *set* ⇒ $'a$ *set* ⇒
    ($'a$, $'v$) *Profile* ⇒ $'r$ *set* **where**
  defer m V A p ≡ *defer-r* (*m V A p*)

4.4.2 Auxiliary Definitions

Electoral modules partition a given set of alternatives A into a set of elected alternatives e, a set of rejected alternatives r, and a set of deferred alternatives d, using a profile. e, r, and d partition A. Electoral modules can be used as voting rules. They can also be composed in multiple structures to create more complex electoral modules.

```
fun (in result) electoral-module :: ('a, 'v, ('r Result)) Electoral-Module ⇒
        bool where
    electoral-module m = (∀ A V p. profile V A p ⟶ well-formed A (m V A p))
```

```
fun voters-determine-election :: ('a, 'v, ('r Result)) Electoral-Module ⇒ bool where
  voters-determine-election m =
    (∀ A V p p'. (∀ v ∈ V. p v = p' v) ⟶ m V A p = m V A p')
```

```
lemma (in result) electoral-modI:
  fixes m :: ('a, 'v, ('r Result)) Electoral-Module
  assumes ∀ A V p. profile V A p ⟶ well-formed A (m V A p)
  shows electoral-module m
  unfolding electoral-module.simps
  using assms
  by simp
```

4.4.3 Properties

We only require voting rules to behave a specific way on admissible elections, i.e., elections that are valid profiles (= votes are linear orders on the alternatives). Note that we do not assume finiteness of voter or alternative sets by default.

Anonymity

An electoral module is anonymous iff the result is invariant under renamings of voters, i.e., any permutation of the voter set that does not change the preferences leads to an identical result.

```
definition (in result) anonymity :: ('a, 'v, ('r Result)) Electoral-Module ⇒
        bool where
  anonymity m ≡
    electoral-module m ∧
      (∀ A V p π :: ('v ⇒ 'v).
        bij π ⟶ (let (A', V', q) = (rename π (A, V, p)) in
            profile V A p ∧ profile V' A' q ⟶ m V A p = m V' A' q))
```

Anonymity can alternatively be described as invariance under the voter permutation group acting on elections via the rename function.

```
fun anonymity-in :: ('a, 'v) Election set ⇒ ('a, 'v, 'r) Electoral-Module ⇒
```

```
      bool where
  anonymity-in X m = is-symmetry (fun_𝓔 m) (Invariance (anonymity_ℛ X))

fun anonymity' :: ('a, 'v, 'r) Electoral-Module ⇒ bool where
  anonymity' m = anonymity-in well-formed-elections m
```

Homogeneity

A voting rule is homogeneous if copying an election does not change the result. For ordered voter types and finite elections, we use the notion of copying ballot lists to define copying an election. The more general definition of homogeneity for unordered voter types already implies anonymity.

```
fun homogeneity-in :: ('a, 'v) Election set ⇒
      ('a, 'v, 'r Result) Electoral-Module ⇒ bool where
  homogeneity-in X m = is-symmetry (fun_𝓔 m) (Invariance (homogeneity_ℛ X))
```

— This does not require any specific behaviour on infinite voter sets . . . It might make sense to extend the definition to that case somehow.

```
fun homogeneity :: ('a, 'v, 'b Result) Electoral-Module ⇒ bool where
  homogeneity m = homogeneity-in finite-elections-𝒱 m

fun homogeneity'-in :: ('a, 'v :: linorder) Election set ⇒
      ('a, 'v, 'b Result) Electoral-Module ⇒ bool where
  homogeneity'-in X m = is-symmetry (fun_𝓔 m) (Invariance (homogeneity_ℛ' X))

fun homogeneity' :: ('a, 'v :: linorder, 'b Result) Electoral-Module ⇒ bool where
  homogeneity' m = homogeneity'-in finite-elections-𝒱 m


lemma hom-imp-anon:
  fixes
    X :: ('a, 'v) Election set and
    m :: ('a, 'v, ('r Result)) Electoral-Module
  assumes
    homogeneity-in X m and
    ∀ E ∈ X. finite (voters-𝓔 E)
  shows anonymity-in X m
proof (unfold anonymity-in.simps is-symmetry.simps, intro allI impI)
  fix E E' :: ('a, 'v) Election
  assume rel: (E, E') ∈ anonymity_ℛ X
  then obtain π :: 'v ⇒ 'v where
    π ∈ carrier bijection_𝒱𝒢 and
    E' = φ-anon X π E
    unfolding anonymity_ℛ.simps action-induced-rel.simps
    by blast
  moreover from this have bij π
    unfolding bijection_𝒱𝒢-def rewrite-carrier
    by simp
  moreover from this have in-election-set: E ∈ X
```

```
    using rel
    unfolding anonymity_ℛ.simps action-induced-rel.simps
    by blast
  ultimately have finite (voters-ℰ E′)
    using assms rename.simps rename-finite split-pairs
    unfolding φ-anon.simps extensional-continuation.simps voters-ℰ.simps
    by metis
  moreover have fin-E: finite (voters-ℰ E)
    using in-election-set assms
    unfolding anonymity_ℛ.simps action-induced-rel.simps
    by blast
  moreover have ∀ r. vote-count r E = 1 ∗ (vote-count r E′)
    using fin-E anon-rel-vote-count rel mult-1
    by metis
  moreover have alternatives-ℰ E = alternatives-ℰ E′
    using fin-E anon-rel-vote-count rel
    by metis
  ultimately show fun_ℰ m E = fun_ℰ m E′
    using assms in-election-set
    unfolding homogeneity-in.simps is-symmetry.simps homogeneity_ℛ.simps
    by blast
qed
```

Neutrality

Neutrality is equivariance under consistent renaming of candidates in the candidate set and election results.

```
fun (in result-properties) neutrality-in :: ('a, 'v) Election set ⇒
        ('a, 'v, 'b Result) Electoral-Module ⇒ bool where
  neutrality-in X m =
    is-symmetry (fun_ℰ m) (action-induced-equivariance (carrier bijection_𝒜𝒢) X
          (φ-neutral X) (result-action ψ))

fun (in result-properties) neutrality :: ('a, 'v, 'b Result) Electoral-Module ⇒
        bool where
  neutrality m = neutrality-in well-formed-elections m
```

4.4.4 Social-Welfare Properties

Reversal Symmetry

A social welfare rule is reversal symmetric if reversing all voters' preferences reverses the result rankings as well.

```
definition reversal-symmetry-in :: ('a, 'v) Election set ⇒
        ('a, 'v, 'a rel Result) Electoral-Module ⇒ bool where
  reversal-symmetry-in X m ≡
    is-symmetry (fun_ℰ m) (action-induced-equivariance (carrier reversal_𝒢) X
          (φ-reverse X) (result-action ψ-reverse))
```

```
fun reversal-symmetry :: ('a, 'v, 'a rel Result) Electoral-Module ⇒ bool where
  reversal-symmetry m = reversal-symmetry-in well-formed-elections m
```

4.4.5 Social-Choice Modules

The following results require electoral modules to return social choice results, i.e., sets of elected, rejected and deferred alternatives. In order to export code, we use the hack provided by Locale-Code.

"defers n" is true for all electoral modules that defer exactly n alternatives, whenever there are n or more alternatives.

```
definition defers :: nat ⇒ ('a, 'v, 'a Result) Electoral-Module ⇒ bool where
  defers n m ≡
    𝒮𝒞ℱ-result.electoral-module m ∧
      (∀ A V p. (card A ≥ n ∧ finite A ∧ profile V A p)
          ⟶ card (defer m V A p) = n)
```

"rejects n" is true for all electoral modules that reject exactly n alternatives, whenever there are n or more alternatives.

```
definition rejects :: nat ⇒ ('a, 'v, 'a Result) Electoral-Module ⇒ bool where
  rejects n m ≡
    𝒮𝒞ℱ-result.electoral-module m ∧
      (∀ A V p. (card A ≥ n ∧ finite A ∧ profile V A p)
          ⟶ card (reject m V A p) = n)
```

As opposed to "rejects", "eliminates" allows to stop rejecting if no alternatives were to remain.

```
definition eliminates :: nat ⇒ ('a, 'v, 'a Result) Electoral-Module ⇒ bool where
  eliminates n m ≡
    𝒮𝒞ℱ-result.electoral-module m ∧
      (∀ A V p. (card A > n ∧ profile V A p) ⟶ card (reject m V A p) = n)
```

"elects n" is true for all electoral modules that elect exactly n alternatives, whenever there are n or more alternatives.

```
definition elects :: nat ⇒ ('a, 'v, 'a Result) Electoral-Module ⇒ bool where
  elects n m ≡
    𝒮𝒞ℱ-result.electoral-module m ∧
      (∀ A V p. (card A ≥ n ∧ profile V A p) ⟶ card (elect m V A p) = n)
```

An electoral module is independent of an alternative a iff a's ranking does not influence the outcome.

```
definition indep-of-alt :: ('a, 'v, 'a Result) Electoral-Module ⇒ 'v set ⇒
        'a set ⇒ 'a ⇒ bool where
  indep-of-alt m V A a ≡
    𝒮𝒞ℱ-result.electoral-module m
      ∧ (∀ p q. equiv-prof-except-a V A p q a ⟶ m V A p = m V A q)
```

```
definition unique-winner-if-profile-non-empty :: ('a, 'v, 'a Result)
    Electoral-Module ⇒ bool where
  unique-winner-if-profile-non-empty m ≡
    𝒮𝒞ℱ-result.electoral-module m ∧
    (∀ A V p. (A ≠ {} ∧ V ≠ {} ∧ profile V A p) ⟶
        (∃ a ∈ A. m V A p = ({a}, A − {a}, {})))
```

4.4.6 Equivalence Definitions

```
definition prof-contains-result :: ('a, 'v, 'a Result) Electoral-Module ⇒ 'v set ⇒
    'a set ⇒ ('a, 'v) Profile ⇒ ('a, 'v) Profile ⇒ 'a ⇒ bool where
  prof-contains-result m V A p q a ≡
    𝒮𝒞ℱ-result.electoral-module m ∧
    profile V A p ∧ profile V A q ∧ a ∈ A ∧
    (a ∈ elect m V A p ⟶ a ∈ elect m V A q) ∧
    (a ∈ reject m V A p ⟶ a ∈ reject m V A q) ∧
    (a ∈ defer m V A p ⟶ a ∈ defer m V A q)
```

```
definition prof-leq-result :: ('a, 'v, 'a Result) Electoral-Module ⇒ 'v set ⇒
    'a set ⇒ ('a, 'v) Profile ⇒ ('a, 'v) Profile ⇒ 'a ⇒ bool where
  prof-leq-result m V A p q a ≡
    𝒮𝒞ℱ-result.electoral-module m ∧
    profile V A p ∧ profile V A q ∧ a ∈ A ∧
    (a ∈ reject m V A p ⟶ a ∈ reject m V A q) ∧
    (a ∈ defer m V A p ⟶ a ∉ elect m V A q)
```

```
definition prof-geq-result :: ('a, 'v, 'a Result) Electoral-Module ⇒ 'v set ⇒
    'a set ⇒ ('a, 'v) Profile ⇒ ('a, 'v) Profile ⇒ 'a ⇒ bool where
  prof-geq-result m V A p q a ≡
    𝒮𝒞ℱ-result.electoral-module m ∧
    profile V A p ∧ profile V A q ∧ a ∈ A ∧
    (a ∈ elect m V A p ⟶ a ∈ elect m V A q) ∧
    (a ∈ defer m V A p ⟶ a ∉ reject m V A q)
```

```
definition mod-contains-result :: ('a, 'v, 'a Result) Electoral-Module ⇒
    ('a, 'v, 'a Result) Electoral-Module ⇒ 'v set ⇒ 'a set ⇒
    ('a, 'v) Profile ⇒ 'a ⇒ bool where
  mod-contains-result m n V A p a ≡
    𝒮𝒞ℱ-result.electoral-module m ∧
    𝒮𝒞ℱ-result.electoral-module n ∧
    profile V A p ∧ a ∈ A ∧
    (a ∈ elect m V A p ⟶ a ∈ elect n V A p) ∧
    (a ∈ reject m V A p ⟶ a ∈ reject n V A p) ∧
    (a ∈ defer m V A p ⟶ a ∈ defer n V A p)
```

```
definition mod-contains-result-sym :: ('a, 'v, 'a Result) Electoral-Module ⇒
    ('a, 'v, 'a Result) Electoral-Module ⇒ 'v set ⇒ 'a set ⇒
    ('a, 'v) Profile ⇒ 'a ⇒ bool where
```

```
mod-contains-result-sym m n V A p a ≡
  𝒮𝒞ℱ-result.electoral-module m ∧
  𝒮𝒞ℱ-result.electoral-module n ∧
  profile V A p ∧ a ∈ A ∧
  (a ∈ elect m V A p ⟷ a ∈ elect n V A p) ∧
  (a ∈ reject m V A p ⟷ a ∈ reject n V A p) ∧
  (a ∈ defer m V A p ⟷ a ∈ defer n V A p)
```

4.4.7 Auxiliary Lemmas

```
lemma elect-rej-def-combination:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    V :: 'v set and
    A :: 'a set and
    p :: ('a, 'v) Profile and
    e r d :: 'a set
  assumes
    elect m V A p = e and
    reject m V A p = r and
    defer m V A p = d
  shows m V A p = (e, r, d)
  using assms
  by auto

lemma par-comp-result-sound:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    p :: ('a, 'v) Profile
  assumes
    𝒮𝒞ℱ-result.electoral-module m and
    profile V A p
  shows well-formed-𝒮𝒞ℱ A (m V A p)
  using assms
  by simp

lemma result-presv-alts:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assumes
    𝒮𝒞ℱ-result.electoral-module m and
    profile V A p
  shows (elect m V A p) ∪ (reject m V A p) ∪ (defer m V A p) = A
proof (safe)
  fix a :: 'a
  have
    partition-imp-exist:
    ∀ p'. set-equals-partition A p'
      ⟶ (∃ E R D. p' = (E, R, D) ∧ E ∪ R ∪ D = A) and
    partition-A:
    set-equals-partition A (m V A p)
    using assms
    by (simp, simp)
  {
    assume a ∈ elect m V A p
    with partition-imp-exist partition-A
    show a ∈ A
      using UnI1 fstI
      by (metis (no-types))
  }
  {
    assume a ∈ reject m V A p
    with partition-imp-exist partition-A
    show a ∈ A
      using UnI1 fstI sndI subsetD sup-ge2
      by metis
  }
  {
    assume a ∈ defer m V A p
    with partition-imp-exist partition-A
    show a ∈ A
      using sndI subsetD sup-ge2
      by metis
  }
  {
    assume
      a ∈ A and
      a ∉ defer m V A p and
      a ∉ reject m V A p
    with partition-imp-exist partition-A
    show a ∈ elect m V A p
      using fst-conv snd-conv Un-iff
      by metis
  }
qed

lemma result-disj:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    p :: ('a, 'v) Profile and
    V :: 'v set
  assumes
    𝒮𝒞ℱ-result.electoral-module m and
    profile V A p
  shows
    (elect m V A p) ∩ (reject m V A p) = {} ∧
        (elect m V A p) ∩ (defer m V A p) = {} ∧
        (reject m V A p) ∩ (defer m V A p) = {}
proof (safe)
  fix a :: 'a
  have wf: well-formed-𝒮𝒞ℱ A (m V A p)
    using assms
    unfolding 𝒮𝒞ℱ-result.electoral-module.simps
    by metis
  have disj: disjoint3 (m V A p)
    using assms
    by simp
  {
    assume
      a ∈ elect m V A p and
      a ∈ reject m V A p
    with wf disj
    show a ∈ {}
      using prod.exhaust-sel DiffE UnCI result-imp-rej
      by (metis (no-types))
  }
  {
    assume
      elect-a: a ∈ elect m V A p and
      defer-a: a ∈ defer m V A p
    then obtain
      e :: 'a Result ⇒ 'a set and
      r :: 'a Result ⇒ 'a set and
      d :: 'a Result ⇒ 'a set
      where
        m V A p =
        (e (m V A p), r (m V A p), d (m V A p)) ∧
          e (m V A p) ∩ r (m V A p) = {} ∧
          e (m V A p) ∩ d (m V A p) = {} ∧
          r (m V A p) ∩ d (m V A p) = {}
      using IntI emptyE prod.collapse disj disjoint3.simps
      by metis
    hence ((elect m V A p) ∩ (reject m V A p) = {}) ∧
          ((elect m V A p) ∩ (defer m V A p) = {}) ∧
          ((reject m V A p) ∩ (defer m V A p) = {})
      using eq-snd-iff fstI
      by metis
    thus a ∈ {}
      using elect-a defer-a disjoint-iff-not-equal
      by (metis (no-types))
  }
  {
    assume
      a ∈ reject m V A p and
      a ∈ defer m V A p
    with wf disj
    show a ∈ {}
      using prod.exhaust-sel DiffE UnCI result-imp-rej
      by (metis (no-types))
  }
qed

lemma elect-in-alts:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    p :: ('a, 'v) Profile
  assumes
    𝒮𝒞ℱ-result.electoral-module m and
    profile V A p
  shows elect m V A p ⊆ A
  using le-supI1 assms result-presv-alts sup-ge1
  by metis

lemma reject-in-alts:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assumes
    𝒮𝒞ℱ-result.electoral-module m and
    profile V A p
  shows reject m V A p ⊆ A
  using le-supI1 assms result-presv-alts sup-ge2
  by metis

lemma defer-in-alts:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assumes
    𝒮𝒞ℱ-result.electoral-module m and
    profile V A p
  shows defer m V A p ⊆ A
  using assms result-presv-alts
  by fastforce

lemma def-presv-prof:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    p :: ('a, 'v) Profile
  assumes
    SCF-result.electoral-module m and
    profile V A p
  shows let new-A = defer m V A p in profile V new-A (limit-profile new-A p)
  using defer-in-alts limit-profile-sound assms
  by metis
```

An electoral module can never reject, defer or elect more than |A| alternatives.

```
lemma upper-card-bounds-for-result:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assumes
    SCF-result.electoral-module m and
    profile V A p and
    finite A
  shows
    upper-card-bound-for-elect: card (elect m V A p) ≤ card A and
    upper-card-bound-for-reject: card (reject m V A p) ≤ card A and
    upper-card-bound-for-defer: card (defer m V A p) ≤ card A
  using assms card-mono
  by (metis elect-in-alts,
      metis reject-in-alts,
      metis defer-in-alts)

lemma reject-not-elected-or-deferred:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assumes
    SCF-result.electoral-module m and
    profile V A p
  shows reject m V A p = A − (elect m V A p) − (defer m V A p)
proof −
  from assms have (elect m V A p) ∪ (reject m V A p) ∪ (defer m V A p) = A
    using result-presv-alts
    by blast
  with assms show ?thesis
    using result-disj
    by blast
```

```
qed

lemma elec-and-def-not-rej:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assumes
    𝒮𝒞ℱ-result.electoral-module m and
    profile V A p
  shows elect m V A p ∪ defer m V A p = A − (reject m V A p)
proof −
  from assms have (elect m V A p) ∪ (reject m V A p) ∪ (defer m V A p) = A
    using result-presv-alts
    by blast
  with assms show ?thesis
    using result-disj
    by blast
qed

lemma defer-not-elec-or-rej:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    p :: ('a, 'v) Profile
  assumes
    𝒮𝒞ℱ-result.electoral-module m and
    profile V A p
  shows defer m V A p = A − (elect m V A p) − (reject m V A p)
proof −
  from assms have (elect m V A p) ∪ (reject m V A p) ∪ (defer m V A p) = A
    using result-presv-alts
    by simp
  with assms show ?thesis
    using result-disj
    by blast
qed

lemma electoral-mod-defer-elem:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assumes
    𝒮𝒞ℱ-result.electoral-module m and
    profile V A p and
    a ∈ A and
    a ∉ elect m V A p and
    a ∉ reject m V A p
  shows a ∈ defer m V A p
  using DiffI assms reject-not-elected-or-deferred
  by metis

lemma mod-contains-result-comm:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assumes mod-contains-result m n V A p a
  shows mod-contains-result n m V A p a
proof (unfold mod-contains-result-def, safe)
  show
    SCF-result.electoral-module n and
    SCF-result.electoral-module m and
    profile V A p and
    a ∈ A
    using assms
    unfolding mod-contains-result-def
    by safe
next
  show
    a ∈ elect n V A p ⟹ a ∈ elect m V A p and
    a ∈ reject n V A p ⟹ a ∈ reject m V A p and
    a ∈ defer n V A p ⟹ a ∈ defer m V A p
    using assms IntI electoral-mod-defer-elem empty-iff result-disj
    unfolding mod-contains-result-def
    by (metis (mono-tags, lifting),
        metis (mono-tags, lifting),
        metis (mono-tags, lifting))
qed

lemma not-rej-imp-elec-or-defer:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assumes
    SCF-result.electoral-module m and
    profile V A p and
    a ∈ A and
    a ∉ reject m V A p
  shows a ∈ elect m V A p ∨ a ∈ defer m V A p
  using assms electoral-mod-defer-elem
  by metis

lemma single-elim-imp-red-def-set:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assumes
    eliminates 1 m and
    card A > 1 and
    profile V A p
  shows defer m V A p ⊂ A
  using Diff-eq-empty-iff Diff-subset card-eq-0-iff defer-in-alts eliminates-def
      eq-iff not-one-le-zero psubsetI reject-not-elected-or-deferred assms
  by (metis (no-types, lifting))

lemma eq-alts-in-profs-imp-eq-results:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p q :: ('a, 'v) Profile
  assumes
    eq: ∀ a ∈ A. prof-contains-result m V A p q a and
    mod-m: 𝒮𝒞ℱ-result.electoral-module m and
    prof-p: profile V A p and
    prof-q: profile V A q
  shows m V A p = m V A q
proof –
  have
    elected-in-A: elect m V A q ⊆ A and
    rejected-in-A: reject m V A q ⊆ A and
    deferred-in-A: defer m V A q ⊆ A
    using mod-m prof-q
    by (metis elect-in-alts, metis reject-in-alts, metis defer-in-alts)
  have
    ∀ a ∈ elect m V A p. a ∈ elect m V A q and
    ∀ a ∈ reject m V A p. a ∈ reject m V A q and
    ∀ a ∈ defer m V A p. a ∈ defer m V A q
    using eq mod-m prof-p in-mono
    unfolding prof-contains-result-def
    by (metis (no-types, lifting) elect-in-alts,
        metis (no-types, lifting) reject-in-alts,
        metis (no-types, lifting) defer-in-alts)
  moreover have
    ∀ a ∈ elect m V A q. a ∈ elect m V A p and
    ∀ a ∈ reject m V A q. a ∈ reject m V A p and
    ∀ a ∈ defer m V A q. a ∈ defer m V A p
  proof (safe)
    fix a :: 'a
    assume q-elect-a: a ∈ elect m V A q
    hence a ∈ A
      using elected-in-A
      by blast
    moreover have
      a ∉ defer m V A q and
      a ∉ reject m V A q
      using q-elect-a prof-q mod-m result-disj disjoint-iff-not-equal
      by (metis, metis)
    ultimately show a ∈ elect m V A p
      using eq electoral-mod-defer-elem
      unfolding prof-contains-result-def
      by metis
  next
    fix a :: 'a
    assume q-rejects-a: a ∈ reject m V A q
    hence a ∈ A
      using rejected-in-A
      by blast
    moreover have
      a ∉ defer m V A q and
      a ∉ elect m V A q
      using q-rejects-a prof-q mod-m result-disj disjoint-iff-not-equal
      by (metis, metis)
    ultimately show a ∈ reject m V A p
      using eq electoral-mod-defer-elem
      unfolding prof-contains-result-def
      by metis
  next
    fix a :: 'a
    assume q-defers-a: a ∈ defer m V A q
    moreover have a ∈ A
      using q-defers-a deferred-in-A
      by blast
    moreover have
      a ∉ elect m V A q and
      a ∉ reject m V A q
      using q-defers-a prof-q mod-m result-disj disjoint-iff-not-equal
      by (metis, metis)
    ultimately show a ∈ defer m V A p
      using eq electoral-mod-defer-elem
      unfolding prof-contains-result-def
      by metis
  qed
  ultimately show ?thesis
```

```
    using prod.collapse subsetI subset-antisym
    by (metis (no-types))
qed

lemma eq-def-and-elect-imp-eq:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p q :: ('a, 'v) Profile
  assumes
    mod-m: 𝒮𝒞ℱ-result.electoral-module m and
    mod-n: 𝒮𝒞ℱ-result.electoral-module n and
    fin-p: profile V A p and
    fin-q: profile V A q and
    elec-eq: elect m V A p = elect n V A q and
    def-eq: defer m V A p = defer n V A q
  shows m V A p = n V A q
proof −
  have
    reject m V A p = A − ((elect m V A p) ∪ (defer m V A p)) and
    reject n V A q = A − ((elect n V A q) ∪ (defer n V A q))
    using elect-rej-def-combination result-imp-rej mod-m mod-n fin-p fin-q
    unfolding 𝒮𝒞ℱ-result.electoral-module.simps
    by (metis, metis)
  thus ?thesis
    using prod-eqI elec-eq def-eq
    by metis
qed
```

4.4.8 Non-Blocking

An electoral module is non-blocking iff this module never rejects all alternatives.

```
definition non-blocking :: ('a, 'v, 'a Result) Electoral-Module ⇒ bool where
  non-blocking m ≡
    𝒮𝒞ℱ-result.electoral-module m ∧
      (∀ A V p. ((A ≠ {} ∧ finite A ∧ profile V A p) ⟶ reject m V A p ≠ A))
```

4.4.9 Electing

An electoral module is electing iff it always elects at least one alternative.

```
definition electing :: ('a, 'v, 'a Result) Electoral-Module ⇒ bool where
  electing m ≡
    𝒮𝒞ℱ-result.electoral-module m ∧
      (∀ A V p. (A ≠ {} ∧ finite A ∧ profile V A p) ⟶ elect m V A p ≠ {})

lemma electing-for-only-alt:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assumes
    one-alt: card A = 1 and
    electing: electing m and
    prof: profile V A p
  shows elect m V A p = A
proof (intro equalityI)
  show elect-in-A: elect m V A p ⊆ A
    using electing prof elect-in-alts
    unfolding electing-def
    by metis
  show A ⊆ elect m V A p
  proof (intro subsetI)
    fix a :: 'a
    assume a ∈ A
    thus a ∈ elect m V A p
      using one-alt electing prof elect-in-A IntD2 Int-absorb2 card-1-singletonE
            card-gt-0-iff equals0I zero-less-one singletonD
      unfolding electing-def
      by (metis (no-types))
  qed
qed

theorem electing-imp-non-blocking:
  fixes m :: ('a, 'v, 'a Result) Electoral-Module
  assumes electing m
  shows non-blocking m
proof (unfold non-blocking-def, safe)
  from assms
  show 𝒮𝒞ℱ-result.electoral-module m
    unfolding electing-def
    by simp
next
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assume
    profile V A p and
    finite A and
    reject m V A p = A and
    a ∈ A
  moreover have
    𝒮𝒞ℱ-result.electoral-module m ∧
```

```
      (∀ A V q. A ≠ {} ∧ finite A ∧ profile V A q ⟶ elect m V A q ≠ {})
    using assms
    unfolding electing-def
    by metis
  ultimately show a ∈ {}
    using Diff-cancel Un-empty elec-and-def-not-rej
    by metis
qed
```

4.4.10 Properties

An electoral module is non-electing iff it never elects an alternative.

```
definition non-electing :: ('a, 'v, 'a Result) Electoral-Module ⇒ bool where
  non-electing m ≡
    𝒮𝒞ℱ-result.electoral-module m
      ∧ (∀ A V p. profile V A p ⟶ elect m V A p = {})

lemma single-rej-decr-def-card:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assumes
    rejecting: rejects 1 m and
    non-electing: non-electing m and
    f-prof: finite-profile V A p
  shows card (defer m V A p) = card A − 1
proof −
  have no-elect:
    𝒮𝒞ℱ-result.electoral-module m
        ∧ (∀ V A q. profile V A q ⟶ elect m V A q = {})
    using non-electing
    unfolding non-electing-def
    by (metis (no-types))
  hence reject m V A p ⊆ A
    using f-prof reject-in-alts
    by metis
  moreover have A = A − elect m V A p
    using no-elect f-prof
    by blast
  ultimately show ?thesis
    using f-prof no-elect rejecting card-Diff-subset card-gt-0-iff
          defer-not-elec-or-rej less-one order-less-imp-le Suc-leI
          bot.extremum-unique card.empty diff-is-0-eq' One-nat-def
    unfolding rejects-def
    by metis
qed
```

```
lemma single-elim-decr-def-card':
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assumes
    eliminating: eliminates 1 m and
    non-electing: non-electing m and
    not-empty: card A > 1 and
    prof-p: profile V A p
  shows card (defer m V A p) = card A − 1
proof −
  have no-elect:
    𝒮𝒞ℱ-result.electoral-module m
        ∧ (∀ A V q. profile V A q ⟶ elect m V A q = {})
    using non-electing
    unfolding non-electing-def
    by (metis (no-types))
  hence reject m V A p ⊆ A
    using prof-p reject-in-alts
    by metis
  moreover have A = A − elect m V A p
    using no-elect prof-p
    by blast
  ultimately show ?thesis
    using prof-p not-empty no-elect eliminating card-ge-0-finite
        card-Diff-subset defer-not-elec-or-rej zero-less-one
    unfolding eliminates-def
    by (metis (no-types, lifting))
qed
```

An electoral module is defer-deciding iff this module chooses exactly 1 alternative to defer and rejects any other alternative. Note that ‘rejects n-1 m‘ can be omitted due to the well-formedness property.

```
definition defer-deciding :: ('a, 'v, 'a Result) Electoral-Module ⇒ bool where
  defer-deciding m ≡
    𝒮𝒞ℱ-result.electoral-module m ∧ non-electing m ∧ defers 1 m
```

An electoral module decrements iff this module rejects at least one alternative whenever possible (|A| > 1).

```
definition decrementing :: ('a, 'v, 'a Result) Electoral-Module ⇒ bool where
  decrementing m ≡
    𝒮𝒞ℱ-result.electoral-module m ∧
      (∀ A V p. profile V A p ∧ card A > 1 ⟶ card (reject m V A p) ≥ 1)
```

```
definition defer-condorcet-consistency :: ('a, 'v, 'a Result)
        Electoral-Module ⇒ bool where
  defer-condorcet-consistency m ≡
```

```
    𝒮𝒞ℱ-result.electoral-module m ∧
    (∀ A V p a. condorcet-winner V A p a ⟶
      (m V A p = ({}, A − (defer m V A p), {d ∈ A. condorcet-winner V A p d})))

definition condorcet-compatibility :: ('a, 'v, 'a Result)
        Electoral-Module ⇒ bool where
  condorcet-compatibility m ≡
    𝒮𝒞ℱ-result.electoral-module m ∧
    (∀ A V p a. condorcet-winner V A p a ⟶
      (a ∉ reject m V A p ∧
        (∀ b. ¬ condorcet-winner V A p b ⟶ b ∉ elect m V A p) ∧
          (a ∈ elect m V A p ⟶
            (∀ b ∈ A. ¬ condorcet-winner V A p b ⟶ b ∈ reject m V A p))))
```

An electoral module is defer-monotone iff, when a deferred alternative is lifted, this alternative remains deferred.

```
definition defer-monotonicity :: ('a, 'v, 'a Result) Electoral-Module ⇒ bool where
  defer-monotonicity m ≡
    𝒮𝒞ℱ-result.electoral-module m ∧
      (∀ A V p q a.
        (a ∈ defer m V A p ∧ lifted V A p q a) ⟶ a ∈ defer m V A q)
```

An electoral module is defer-lift-invariant iff lifting a deferred alternative does not affect the outcome.

```
definition defer-lift-invariance :: ('a, 'v, 'a Result) Electoral-Module ⇒ bool where
  defer-lift-invariance m ≡
    𝒮𝒞ℱ-result.electoral-module m ∧
      (∀ A V p q a. (a ∈ (defer m V A p) ∧ lifted V A p q a)
                    ⟶ m V A p = m V A q)

fun dli-rel :: ('a, 'v, 'a Result) Electoral-Module ⇒ ('a, 'v) Election rel where
  dli-rel m = {((A, V, p), (A, V, q)) |A V p q. (∃ a ∈ defer m V A p. lifted V A
p q a)}

lemma rewrite-dli-as-invariance:
  fixes m :: ('a, 'v, 'a Result) Electoral-Module
  shows
    defer-lift-invariance m =
      (𝒮𝒞ℱ-result.electoral-module m
          ∧ (is-symmetry (fun_ℰ m) (Invariance (dli-rel m))))
proof (unfold is-symmetry.simps, safe)
  assume defer-lift-invariance m
  thus 𝒮𝒞ℱ-result.electoral-module m
    unfolding defer-lift-invariance-def
    by blast
next
  fix
    A A' :: 'a set and
    V V' :: 'v set and
    p q :: ('a, 'v) Profile
  assume
    invar: defer-lift-invariance m and
    rel: ((A, V, p), (A', V', q)) ∈ dli-rel m
  then obtain a :: 'a where
    a ∈ defer m V A p ∧ lifted V A p q a
    unfolding dli-rel.simps
    by blast
  moreover with rel have A = A' ∧ V = V'
    by simp
  ultimately show fun_ℰ m (A, V, p) = fun_ℰ m (A', V', q)
    using invar fst-eqD snd-eqD profile-ℰ.simps
   unfolding defer-lift-invariance-def fun_ℰ.simps alternatives-ℰ.simps voters-ℰ.simps
    by metis
next
  assume
    𝒮𝒞ℱ-result.electoral-module m and
    ∀ E E'. (E, E') ∈ dli-rel m ⟶ fun_ℰ m E = fun_ℰ m E'
  hence 𝒮𝒞ℱ-result.electoral-module m ∧ (∀ A V p q.
    ((A, V, p), (A, V, q)) ∈ dli-rel m ⟶ m V A p = m V A q)
    unfolding fun_ℰ.simps alternatives-ℰ.simps profile-ℰ.simps voters-ℰ.simps
    using fst-conv snd-conv
    by metis
  moreover have
    ∀ A V p q a. (a ∈ (defer m V A p) ∧ lifted V A p q a) ⟶
      ((A, V, p), (A, V, q)) ∈ dli-rel m
    unfolding dli-rel.simps
    by blast
  ultimately show defer-lift-invariance m
    unfolding defer-lift-invariance-def
    by blast
qed
```

Two electoral modules are disjoint-compatible if they only make decisions over disjoint sets of alternatives. Electoral modules reject alternatives for which they make no decision.

```
definition disjoint-compatibility :: ('a, 'v, 'a Result) Electoral-Module ⇒
        ('a, 'v, 'a Result) Electoral-Module ⇒ bool where
  disjoint-compatibility m n ≡
    𝒮𝒞ℱ-result.electoral-module m ∧ 𝒮𝒞ℱ-result.electoral-module n ∧
        (∀ V.
          (∀ A.
            (∃ B ⊆ A.
              (∀ a ∈ B. indep-of-alt m V A a ∧
                (∀ p. profile V A p ⟶ a ∈ reject m V A p)) ∧
              (∀ a ∈ A − B. indep-of-alt n V A a ∧
                (∀ p. profile V A p ⟶ a ∈ reject n V A p)))))
```

Lifting an elected alternative a from an invariant-monotone electoral module either does not change the elect set, or makes a the only elected alternative.

```
definition invariant-monotonicity :: ('a, 'v, 'a Result)
        Electoral-Module ⇒ bool where
  invariant-monotonicity m ≡
    SCF-result.electoral-module m ∧
        (∀ A V p q a. (a ∈ elect m V A p ∧ lifted V A p q a) ⟶
          (elect m V A q = elect m V A p ∨ elect m V A q = {a}))
```

Lifting a deferred alternative a from a defer-invariant-monotone electoral module either does not change the defer set, or makes a the only deferred alternative.

```
definition defer-invariant-monotonicity :: ('a, 'v, 'a Result)
        Electoral-Module ⇒ bool where
  defer-invariant-monotonicity m ≡
    SCF-result.electoral-module m ∧ non-electing m ∧
        (∀ A V p q a. (a ∈ defer m V A p ∧ lifted V A p q a) ⟶
          (defer m V A q = defer m V A p ∨ defer m V A q = {a}))
```

4.4.11 Inference Rules

```
lemma ccomp-and-dd-imp-def-only-winner:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assumes
    ccomp: condorcet-compatibility m and
    dd: defer-deciding m and
    winner: condorcet-winner V A p a
  shows defer m V A p = {a}
proof (rule ccontr)
  assume defer m V A p ≠ {a}
  moreover have def-one: defers 1 m
    using dd
    unfolding defer-deciding-def
    by metis
  hence c-win: finite-profile V A p ∧  a ∈ A ∧ (∀ b ∈ A − {a}. wins V a p b)
    using winner
    by auto
  ultimately have ∃ b ∈ A. b ≠ a ∧ defer m V A p = {b}
    using Suc-leI card-gt-0-iff def-one equals0D card-1-singletonE
          defer-in-alts insert-subset
    unfolding defer-deciding-def One-nat-def defers-def
    by metis
  hence a ∉ defer m V A p
    by force
  hence a ∈ reject m V A p
```

```
    using ccomp c-win electoral-mod-defer-elem dd equals0D
    unfolding defer-deciding-def non-electing-def condorcet-compatibility-def
    by metis
  moreover have a ∉ reject m V A p
    using ccomp c-win winner
    unfolding condorcet-compatibility-def
    by simp
  ultimately show False
    by simp
qed

theorem ccomp-and-dd-imp-dcc[simp]:
  fixes m :: ('a, 'v, 'a Result) Electoral-Module
  assumes
    ccomp: condorcet-compatibility m and
    dd: defer-deciding m
  shows defer-condorcet-consistency m
proof (unfold defer-condorcet-consistency-def, safe)
  show SCF-result.electoral-module m
    using dd
    unfolding defer-deciding-def
    by metis
next
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assume c-winner: condorcet-winner V A p a
  hence elect m V A p = {}
    using dd
    unfolding defer-deciding-def non-electing-def
    by simp
  moreover have defer m V A p = {a}
    using c-winner dd ccomp ccomp-and-dd-imp-def-only-winner
    by simp
  ultimately have m V A p = ({}, A − defer m V A p, {a})
    using c-winner reject-not-elected-or-deferred
          elect-rej-def-combination Diff-empty dd
    unfolding defer-deciding-def condorcet-winner.simps
    by metis
  moreover have {a} = {c ∈ A. condorcet-winner V A p c}
    using c-winner cond-winner-unique
    by metis
  ultimately show
    m V A p = ({}, A − defer m V A p, {c ∈ A. condorcet-winner V A p c})
    by simp
qed
```

If m and n are disjoint compatible, so are n and m.

```
theorem disj-compat-comm[simp]:
  fixes m n :: ('a, 'v, 'a Result) Electoral-Module
  assumes disjoint-compatibility m n
  shows disjoint-compatibility n m
proof (unfold disjoint-compatibility-def, safe)
  show
    𝒮𝒞ℱ-result.electoral-module m and
    𝒮𝒞ℱ-result.electoral-module n
    using assms
    unfolding disjoint-compatibility-def
    by safe
next
  fix
    A :: 'a set and
    V :: 'v set
  obtain B :: 'a set where
    B ⊆ A ∧
      (∀ a ∈ B.
        indep-of-alt m V A a ∧ (∀ p. profile V A p ⟶ a ∈ reject m V A p)) ∧
      (∀ a ∈ A − B.
        indep-of-alt n V A a ∧ (∀ p. profile V A p ⟶ a ∈ reject n V A p))
    using assms
    unfolding disjoint-compatibility-def
    by metis
  hence
    ∃ B ⊆ A.
      (∀ a ∈ A − B.
        indep-of-alt n V A a ∧ (∀ p. profile V A p ⟶ a ∈ reject n V A p)) ∧
      (∀ a ∈ B.
        indep-of-alt m V A a ∧ (∀ p. profile V A p ⟶ a ∈ reject m V A p))
    by blast
  thus ∃ B ⊆ A.
          (∀ a ∈ B.
            indep-of-alt n V A a ∧ (∀ p. profile V A p ⟶ a ∈ reject n V A p)) ∧
          (∀ a ∈ A − B.
            indep-of-alt m V A a ∧ (∀ p. profile V A p ⟶ a ∈ reject m V A p))
    by fastforce
qed
```

Every electoral module which is defer-lift-invariant is also defer-monotone.

```
theorem dl-inv-imp-def-mono[simp]:
  fixes m :: ('a, 'v, 'a Result) Electoral-Module
  assumes defer-lift-invariance m
  shows defer-monotonicity m
  using assms
  unfolding defer-monotonicity-def defer-lift-invariance-def
  by metis
```

4.4.12 Social-Choice Properties

Condorcet Consistency

```
definition condorcet-consistency :: ('a, 'v, 'a Result) Electoral-Module ⇒
        bool where
  condorcet-consistency m ≡
    𝒮𝒞ℱ-result.electoral-module m ∧
    (∀ A V p a. condorcet-winner V A p a ⟶
      (m V A p = ({e ∈ A. condorcet-winner V A p e}, A − (elect m V A p), {})))

lemma condorcet-consistency-equiv:
  fixes m :: ('a, 'v, 'a Result) Electoral-Module
  shows condorcet-consistency m =
          (𝒮𝒞ℱ-result.electoral-module m ∧
            (∀ A V p a. condorcet-winner V A p a ⟶
              (m V A p = ({a}, A − (elect m V A p), {}))))
proof (safe)
  assume condorcet-consistency m
  thus 𝒮𝒞ℱ-result.electoral-module m
    unfolding condorcet-consistency-def
    by (metis (mono-tags, lifting))
next
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assume
    condorcet-consistency m and
    condorcet-winner V A p a
  thus m V A p = ({a}, A − elect m V A p, {})
    using cond-winner-unique
    unfolding condorcet-consistency-def
    by (metis (mono-tags, lifting))
next
  assume
    𝒮𝒞ℱ-result.electoral-module m and
    ∀ A V p a. condorcet-winner V A p a
        ⟶ m V A p = ({a}, A − elect m V A p, {})
  thus condorcet-consistency m
    using cond-winner-unique
    unfolding condorcet-consistency-def
    by (metis (mono-tags, lifting))
qed

lemma condorcet-consistency-equiv':
  fixes m :: ('a, 'v, 'a Result) Electoral-Module
  shows condorcet-consistency m =
          (𝒮𝒞ℱ-result.electoral-module m ∧
              (∀ A V p a.
                condorcet-winner V A p a ⟶ m V A p = ({a}, A − {a}, {})))
proof (unfold condorcet-consistency-equiv, safe)
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assume condorcet-winner V A p a
  {
    moreover assume
      ∀ A V p a'. condorcet-winner V A p a'
          ⟶ m V A p = ({a'}, A − elect m V A p, {})
    ultimately show m V A p = ({a}, A − {a}, {})
      using fst-conv
      by metis
  }
  {
    moreover assume
      ∀ A V p a'. condorcet-winner V A p a'
          ⟶ m V A p = ({a'}, A −  {a'}, {})
    ultimately show m V A p = ({a}, A −  elect m V A p, {})
      using fst-conv
      by metis
  }
qed
```

(Weak) Monotonicity

An electoral module is monotone iff when an elected alternative is lifted, this alternative remains elected.

```
definition monotonicity :: ('a, 'v, 'a Result) Electoral-Module ⇒ bool where
  monotonicity m ≡
    𝒮𝒞ℱ-result.electoral-module m ∧
      (∀ A V p q a. a ∈ elect m V A p ∧ lifted V A p q a ⟶ a ∈ elect m V A q)

end
```

4.5 Electoral Module on Election Quotients

```
theory Quotient-Module
  imports Quotients/Relation-Quotients
          Electoral-Module
begin

lemma invariance-is-congruence:
  fixes
    m :: ('a, 'v, 'r) Electoral-Module and
    r :: ('a, 'v) Election rel
  shows is-symmetry (fun_ℰ m) (Invariance r) = fun_ℰ m respects r
  unfolding is-symmetry.simps congruent-def
  by blast

lemma invariance-is-congruence':
  fixes
    f :: 'x ⇒ 'y and
    r :: 'x rel
  shows is-symmetry f (Invariance r) = f respects r
  unfolding is-symmetry.simps congruent-def
  by blast

theorem pass-to-election-quotient:
  fixes
    m :: ('a, 'v, 'r) Electoral-Module and
    r :: ('a, 'v) Election rel and
    X :: ('a, 'v) Election set
  assumes
    equiv X r and
    is-symmetry (fun_ℰ m) (Invariance r)
  shows ∀ A ∈ X // r. ∀ E ∈ A. π_𝒬 (fun_ℰ m) A = fun_ℰ m E
  using invariance-is-congruence pass-to-quotient assms
  by blast

end
```

4.6 Evaluation Function

```
theory Evaluation-Function
  imports Social-Choice-Types/Profile
begin
```

This is the evaluation function. From a set of currently eligible alternatives, the evaluation function computes a numerical value that is then to be used for further (s)election, e.g., by the elimination module.

4.6.1 Definition

```
type-synonym ('a, 'v) Evaluation-Function =
  'v set ⇒ 'a ⇒ 'a set ⇒ ('a, 'v) Profile ⇒ enat
```

4.6.2 Property

An Evaluation function is a Condorcet-rating iff the following holds: If a Condorcet Winner w exists, w and only w has the highest value.

```
definition condorcet-rating :: ('a, 'v) Evaluation-Function ⇒ bool where
  condorcet-rating f ≡
    ∀ A V p w . condorcet-winner V A p w ⟶
      (∀ l ∈ A . l ≠ w ⟶ f V l A p < f V w A p)
```

An Evaluation function is dependent only on the participating voters iff it is invariant under profile changes that only impact non-voters.

```
fun voters-determine-evaluation :: ('a, 'v) Evaluation-Function ⇒ bool where
  voters-determine-evaluation f =
    (∀ A V p p'. (∀ v ∈ V. p v = p' v) ⟶ (∀ a ∈ A. f V a A p = f V a A p'))
```

4.6.3 Theorems

If e is Condorcet-rating, the following holds: If a Condorcet winner w exists, w has the maximum evaluation value.

```
theorem cond-winner-imp-max-eval-val:
  fixes
    e :: ('a, 'v) Evaluation-Function and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assumes
    rating: condorcet-rating e and
    f-prof: finite-profile V A p and
    winner: condorcet-winner V A p a
  shows e V a A p = Max {e V b A p | b. b ∈ A}
proof −
  let ?set = {e V b A p | b. b ∈ A} and
      ?eMax = Max {e V b A p | b. b ∈ A} and
      ?eW = e V a A p
  have ?eW ∈ ?set
    using CollectI winner
    unfolding condorcet-winner.simps
    by (metis (mono-tags, lifting))
  moreover have ∀ e ∈ ?set. e ≤ ?eW
  proof (safe)
    fix b :: 'a
    assume b ∈ A
    thus e V b A p ≤ e V a A p
      using less-imp-le rating winner order-refl
      unfolding condorcet-rating-def
      by metis
  qed
  moreover have finite ?set
    using f-prof
    by simp
  moreover have ?set ≠ {}
    using winner
    unfolding condorcet-winner.simps
    by fastforce
  ultimately show ?thesis
    using Max-eq-iff
    by (metis (no-types, lifting))
qed
```

If e is Condorcet-rating, the following holds: If a Condorcet Winner w exists, a non-Condorcet winner has a value lower than the maximum evaluation value.

```
theorem non-cond-winner-not-max-eval:
  fixes
    e :: ('a, 'v) Evaluation-Function and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a b :: 'a
  assumes
    rating: condorcet-rating e and
    f-prof: finite-profile V A p and
    winner: condorcet-winner V A p a and
    lin-A: b ∈ A and
    loser: a ≠ b
  shows e V b A p < Max {e V c A p | c. c ∈ A}
proof −
  have e V b A p < e V a A p
    using lin-A loser rating winner
    unfolding condorcet-rating-def
    by metis
  also have ... = Max {e V c A p | c. c ∈ A}
    using cond-winner-imp-max-eval-val f-prof rating winner
    by fastforce
  finally show ?thesis
    by simp
qed

end
```

4.7 Elimination Module

```
theory Elimination-Module
  imports Evaluation-Function
          Electoral-Module
begin
```

This is the elimination module. It rejects a set of alternatives only if these are not all alternatives. The alternatives potentially to be rejected are put in a so-called elimination set. These are all alternatives that score below a preset threshold value that depends on the specific voting rule.

4.7.1 General Definitions

```
type-synonym Threshold-Value = enat

type-synonym Threshold-Relation = enat ⇒ enat ⇒ bool

type-synonym ('a, 'v) Electoral-Set = 'v set ⇒ 'a set ⇒ ('a, 'v) Profile ⇒ 'a set

fun elimination-set :: ('a, 'v) Evaluation-Function ⇒ Threshold-Value ⇒
        Threshold-Relation ⇒ ('a, 'v) Electoral-Set where
 elimination-set e t r V A p = (if finite A then {a ∈ A . r (e V a A p) t} else {})

fun average :: ('a, 'v) Evaluation-Function ⇒ 'v set ⇒ 'a set ⇒ ('a, 'v) Profile ⇒
        Threshold-Value where
  average e V A p = (let sum-eval = (∑ x ∈ A. e V x A p) in
                    (if sum-eval = ∞ then ∞ else the-enat sum-eval div card A))
```

4.7.2 Social-Choice Definitions

```
fun elimination-module :: ('a, 'v) Evaluation-Function ⇒ Threshold-Value ⇒
        Threshold-Relation ⇒ ('a, 'v, 'a Result) Electoral-Module where
  elimination-module e t r V A p =
      (if (elimination-set e t r V A p) ≠ A
        then ({}, (elimination-set e t r V A p), A − (elimination-set e t r V A p))
        else ({}, {}, A))
```

4.7.3 Social-Choice Eliminators

```
fun less-eliminator :: ('a, 'v) Evaluation-Function ⇒ Threshold-Value ⇒
        ('a, 'v, 'a Result) Electoral-Module where
  less-eliminator e t V A p = elimination-module e t (<) V A p

fun max-eliminator :: ('a, 'v) Evaluation-Function ⇒
        ('a, 'v, 'a Result) Electoral-Module where
  max-eliminator e V A p =
    less-eliminator e (Max {e V x A p | x. x ∈ A}) V A p
```

```
fun leq-eliminator :: ('a, 'v) Evaluation-Function ⇒ Threshold-Value ⇒
      ('a, 'v, 'a Result) Electoral-Module where
  leq-eliminator e t V A p = elimination-module e t (≤) V A p

fun min-eliminator :: ('a, 'v) Evaluation-Function ⇒
      ('a, 'v, 'a Result) Electoral-Module where
  min-eliminator e V A p =
    leq-eliminator e (Min {e V x A p | x. x ∈ A}) V A p

fun less-average-eliminator :: ('a, 'v) Evaluation-Function ⇒
      ('a, 'v, 'a Result) Electoral-Module where
  less-average-eliminator e V A p = less-eliminator e (average e V A p) V A p

fun leq-average-eliminator :: ('a, 'v) Evaluation-Function ⇒
      ('a, 'v, 'a Result) Electoral-Module where
  leq-average-eliminator e V A p = leq-eliminator e (average e V A p) V A p
```

4.7.4 Soundness

```
lemma elim-mod-sound[simp]:
  fixes
    e :: ('a, 'v) Evaluation-Function and
    t :: Threshold-Value and
    r :: Threshold-Relation
  shows 𝒮𝒞ℱ-result.electoral-module (elimination-module e t r)
  unfolding 𝒮𝒞ℱ-result.electoral-module.simps
  by auto

lemma less-elim-sound[simp]:
  fixes
    e :: ('a, 'v) Evaluation-Function and
    t :: Threshold-Value
  shows 𝒮𝒞ℱ-result.electoral-module (less-eliminator e t)
  unfolding 𝒮𝒞ℱ-result.electoral-module.simps
  by auto

lemma leq-elim-sound[simp]:
  fixes
    e :: ('a, 'v) Evaluation-Function and
    t :: Threshold-Value
  shows 𝒮𝒞ℱ-result.electoral-module (leq-eliminator e t)
  unfolding 𝒮𝒞ℱ-result.electoral-module.simps
  by auto

lemma max-elim-sound[simp]:
  fixes e :: ('a, 'v) Evaluation-Function
  shows 𝒮𝒞ℱ-result.electoral-module (max-eliminator e)
  unfolding 𝒮𝒞ℱ-result.electoral-module.simps
  by auto
```

```
lemma min-elim-sound[simp]:
  fixes e :: ('a, 'v) Evaluation-Function
  shows SCF-result.electoral-module (min-eliminator e)
  unfolding SCF-result.electoral-module.simps
  by auto

lemma less-avg-elim-sound[simp]:
  fixes e :: ('a, 'v) Evaluation-Function
  shows SCF-result.electoral-module (less-average-eliminator e)
  unfolding SCF-result.electoral-module.simps
  by auto

lemma leq-avg-elim-sound[simp]:
  fixes e :: ('a, 'v) Evaluation-Function
  shows SCF-result.electoral-module (leq-average-eliminator e)
  unfolding SCF-result.electoral-module.simps
  by auto
```

4.7.5 Independence of Non-Voters

```
lemma voters-determine-elim-mod[simp]:
  fixes
    e :: ('a, 'v) Evaluation-Function and
    t :: Threshold-Value and
    r :: Threshold-Relation
  assumes voters-determine-evaluation e
  shows voters-determine-election (elimination-module e t r)
proof (unfold voters-determine-election.simps elimination-module.simps, safe)
  fix
    A :: 'a set and
    V :: 'v set and
    p p' :: ('a, 'v) Profile
  assume ∀ v ∈ V. p v = p' v
  hence ∀ a ∈ A. (e V a A p) = (e V a A p')
    using assms
    unfolding voters-determine-election.simps
    by simp
  hence {a ∈ A. r (e V a A p) t} = {a ∈ A. r (e V a A p') t}
    by metis
  hence elimination-set e t r V A p = elimination-set e t r V A p'
    unfolding elimination-set.simps
    by presburger
  thus (if elimination-set e t r V A p ≠ A
        then ({}, elimination-set e t r V A p, A − elimination-set e t r V A p)
        else ({}, {}, A)) =
     (if elimination-set e t r V A p' ≠ A
        then ({}, elimination-set e t r V A p', A − elimination-set e t r V A p')
        else ({}, {}, A))
  by presburger
qed

lemma voters-determine-less-elim[simp]:
  fixes
    e :: ('a, 'v) Evaluation-Function and
    t :: Threshold-Value
  assumes voters-determine-evaluation e
  shows voters-determine-election (less-eliminator e t)
  using assms voters-determine-elim-mod
  unfolding less-eliminator.simps voters-determine-election.simps
  by (metis (full-types))

lemma voters-determine-leq-elim[simp]:
  fixes
    e :: ('a, 'v) Evaluation-Function and
    t :: Threshold-Value
  assumes voters-determine-evaluation e
  shows voters-determine-election (leq-eliminator e t)
  using assms voters-determine-elim-mod
  unfolding leq-eliminator.simps voters-determine-election.simps
  by (metis (full-types))

lemma voters-determine-max-elim[simp]:
  fixes e :: ('a, 'v) Evaluation-Function
  assumes voters-determine-evaluation e
  shows voters-determine-election (max-eliminator e)
proof (unfold max-eliminator.simps voters-determine-election.simps, safe)
  fix
    A :: 'a set and
    V :: 'v set and
    p p' :: ('a, 'v) Profile
  assume coinciding: ∀ v ∈ V. p v = p' v
  hence ∀ x ∈ A. e V x A p = e V x A p'
    using assms
    unfolding voters-determine-evaluation.simps
    by simp
  hence Max {e V x A p | x. x ∈ A} = Max {e V x A p' | x. x ∈ A}
    by metis
  thus less-eliminator e (Max {e V x A p | x. x ∈ A}) V A p =
      less-eliminator e (Max {e V x A p' | x. x ∈ A}) V A p'
    using coinciding assms voters-determine-less-elim
    unfolding voters-determine-election.simps
    by (metis (no-types, lifting))
qed

lemma voters-determine-min-elim[simp]:
  fixes e :: ('a, 'v) Evaluation-Function
  assumes voters-determine-evaluation e
  shows voters-determine-election (min-eliminator e)
proof (unfold min-eliminator.simps voters-determine-election.simps, safe)
  fix
    A :: 'a set and
    V :: 'v set and
    p p' :: ('a, 'v) Profile
  assume coinciding: ∀ v ∈ V. p v = p' v
  hence ∀ x ∈ A. e V x A p = e V x A p'
    using assms
    unfolding voters-determine-election.simps
    by simp
  hence Min {e V x A p | x. x ∈ A} = Min {e V x A p' | x. x ∈ A}
    by metis
  thus leq-eliminator e (Min {e V x A p | x. x ∈ A}) V A p =
      leq-eliminator e (Min {e V x A p' | x. x ∈ A}) V A p'
    using coinciding assms voters-determine-leq-elim
    unfolding voters-determine-election.simps
    by (metis (no-types, lifting))
qed

lemma voters-determine-less-avg-elim[simp]:
  fixes e :: ('a, 'v) Evaluation-Function
  assumes voters-determine-evaluation e
  shows voters-determine-election (less-average-eliminator e)
proof (unfold less-average-eliminator.simps voters-determine-election.simps, safe)
  fix
    A :: 'a set and
    V :: 'v set and
    p p' :: ('a, 'v) Profile
  assume coinciding: ∀ v ∈ V. p v = p' v
  hence ∀ x ∈ A. e V x A p = e V x A p'
    using assms
    unfolding voters-determine-election.simps
    by simp
  hence average e V A p = average e V A p'
    unfolding average.simps
    by auto
  thus less-eliminator e (average e V A p) V A p =
      less-eliminator e (average e V A p') V A p'
    using coinciding assms voters-determine-less-elim
    unfolding voters-determine-election.simps
    by (metis (no-types, lifting))
qed

lemma voters-determine-leq-avg-elim[simp]:
  fixes e :: ('a, 'v) Evaluation-Function
  assumes voters-determine-evaluation e
  shows voters-determine-election (leq-average-eliminator e)
proof (unfold leq-average-eliminator.simps voters-determine-election.simps, safe)
```

```
  fix
    A :: 'a set and
    V :: 'v set and
    p p' :: ('a, 'v) Profile
  assume coinciding: ∀ v ∈ V. p v = p' v
  hence ∀ x ∈ A. e V x A p = e V x A p'
    using assms
    unfolding voters-determine-election.simps
    by simp
  hence average e V A p = average e V A p'
    unfolding average.simps
    by auto
  thus leq-eliminator e (average e V A p) V A p =
      leq-eliminator e (average e V A p') V A p'
    using coinciding assms voters-determine-leq-elim
    unfolding voters-determine-election.simps
    by (metis (no-types, lifting))
qed
```

4.7.6 Non-Blocking

```
lemma elim-mod-non-blocking:
  fixes
    e :: ('a, 'v) Evaluation-Function and
    t :: Threshold-Value and
    r :: Threshold-Relation
  shows non-blocking (elimination-module e t r)
  unfolding non-blocking-def
  by auto

lemma less-elim-non-blocking:
  fixes
    e :: ('a, 'v) Evaluation-Function and
    t :: Threshold-Value
  shows non-blocking (less-eliminator e t)
  unfolding less-eliminator.simps
  using elim-mod-non-blocking
  by auto

lemma leq-elim-non-blocking:
  fixes
    e :: ('a, 'v) Evaluation-Function and
    t :: Threshold-Value
  shows non-blocking (leq-eliminator e t)
  unfolding leq-eliminator.simps
  using elim-mod-non-blocking
  by auto

lemma max-elim-non-blocking:
```

fixes *e* :: (*'a*, *'v*) *Evaluation-Function*
shows *non-blocking* (*max-eliminator e*)
unfolding *non-blocking-def*
using $\mathcal{SCF}$*-result.electoral-module.simps*
by *auto*

lemma *min-elim-non-blocking*:
fixes *e* :: (*'a*, *'v*) *Evaluation-Function*
shows *non-blocking* (*min-eliminator e*)
unfolding *non-blocking-def*
using $\mathcal{SCF}$*-result.electoral-module.simps*
by *auto*

lemma *less-avg-elim-non-blocking*:
fixes *e* :: (*'a*, *'v*) *Evaluation-Function*
shows *non-blocking* (*less-average-eliminator e*)
unfolding *non-blocking-def*
using $\mathcal{SCF}$*-result.electoral-module.simps*
by *auto*

lemma *leq-avg-elim-non-blocking*:
fixes *e* :: (*'a*, *'v*) *Evaluation-Function*
shows *non-blocking* (*leq-average-eliminator e*)
unfolding *non-blocking-def*
using $\mathcal{SCF}$*-result.electoral-module.simps*
by *auto*

4.7.7 Non-Electing

lemma *elim-mod-non-electing*:
fixes
e :: (*'a*, *'v*) *Evaluation-Function* **and**
t :: *Threshold-Value* **and**
r :: *Threshold-Relation*
shows *non-electing* (*elimination-module e t r*)
unfolding *non-electing-def*
by *force*

lemma *less-elim-non-electing*:
fixes
e :: (*'a*, *'v*) *Evaluation-Function* **and**
t :: *Threshold-Value*
shows *non-electing* (*less-eliminator e t*)
using *elim-mod-non-electing less-elim-sound*
unfolding *non-electing-def*
by *force*

lemma *leq-elim-non-electing*:
fixes

```
    e :: ('a, 'v) Evaluation-Function and
    t :: Threshold-Value
  shows non-electing (leq-eliminator e t)
  unfolding non-electing-def
  by force

lemma max-elim-non-electing:
  fixes e :: ('a, 'v) Evaluation-Function
  shows non-electing (max-eliminator e)
  unfolding non-electing-def
  by force

lemma min-elim-non-electing:
  fixes e :: ('a, 'v) Evaluation-Function
  shows non-electing (min-eliminator e)
  unfolding non-electing-def
  by force

lemma less-avg-elim-non-electing:
  fixes e :: ('a, 'v) Evaluation-Function
  shows non-electing (less-average-eliminator e)
  unfolding non-electing-def
  by auto

lemma leq-avg-elim-non-electing:
  fixes e :: ('a, 'v) Evaluation-Function
  shows non-electing (leq-average-eliminator e)
  unfolding non-electing-def
  by force
```

4.7.8 Inference Rules

If the used evaluation function is Condorcet rating, max-eliminator is Condorcet compatible.

```
theorem cr-eval-imp-ccomp-max-elim[simp]:
  fixes e :: ('a, 'v) Evaluation-Function
  assumes condorcet-rating e
  shows condorcet-compatibility (max-eliminator e)
proof (unfold condorcet-compatibility-def, safe)
  show 𝒮𝒞ℱ-result.electoral-module (max-eliminator e)
    by force
next
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assume
    c-win: condorcet-winner V A p a and
    rej-a: a ∈ reject (max-eliminator e) V A p
  have e V a A p = Max {e V b A p | b. b ∈ A}
    using c-win cond-winner-imp-max-eval-val assms
    by fastforce
  hence a ∉ reject (max-eliminator e) V A p
    by simp
  thus False
    using rej-a
    by linarith
next
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assume a ∈ elect (max-eliminator e) V A p
  moreover have a ∉ elect (max-eliminator e) V A p
    by simp
  ultimately show False
    by linarith
next
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a a' :: 'a
  assume
    condorcet-winner V A p a and
    a ∈ elect (max-eliminator e) V A p
  thus a' ∈ reject (max-eliminator e) V A p
    using empty-iff max-elim-non-electing
    unfolding condorcet-winner.simps non-electing-def
    by metis
qed
```

If the used evaluation function is Condorcet rating, max-eliminator is defer-Condorcet-consistent.

```
theorem cr-eval-imp-dcc-max-elim[simp]:
  fixes e :: ('a, 'v) Evaluation-Function
  assumes condorcet-rating e
  shows defer-condorcet-consistency (max-eliminator e)
proof (unfold defer-condorcet-consistency-def, safe)
  show 𝒮𝒞ℱ-result.electoral-module (max-eliminator e)
    using max-elim-sound
    by metis
next
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assume winner: condorcet-winner V A p a
  hence f-prof: finite-profile V A p
    by simp
  let ?trsh = Max {e V b A p | b. b ∈ A}
  show
    max-eliminator e V A p =
      ({},
        A − defer (max-eliminator e) V A p,
        {b ∈ A. condorcet-winner V A p b})
  proof (cases elimination-set e (?trsh) (<) V A p ≠ A)
    have e V a A p = Max {e V x A p | x. x ∈ A}
      using winner assms cond-winner-imp-max-eval-val
      by fastforce
    hence ∀ b ∈ A. b ≠ a
        ⟷ b ∈ {c ∈ A. e V c A p < Max {e V b A p | b. b ∈ A}}
      using winner assms mem-Collect-eq linorder-neq-iff
      unfolding condorcet-rating-def
      by (metis (mono-tags, lifting))
    hence elim-set: (elimination-set e ?trsh (<) V A p) = A − {a}
      unfolding elimination-set.simps
      using f-prof
      by fastforce
    case True
    hence
      max-eliminator e V A p =
        ({},
          (elimination-set e ?trsh (<) V A p),
          A − (elimination-set e ?trsh (<) V A p))
      by simp
    also have ... = ({}, A − defer (max-eliminator e) V A p, {a})
      using elim-set winner
      by auto
    also have
      ... = ({},
            A − defer (max-eliminator e) V A p,
            {b ∈ A. condorcet-winner V A p b})
      using cond-winner-unique winner Collect-cong
      by (metis (no-types, lifting))
    finally show ?thesis
      using winner
      by metis
  next
    case False
    moreover have ?trsh = e V a A p
      using assms winner cond-winner-imp-max-eval-val
      by fastforce
    ultimately show ?thesis
```

```
   using winner
   by auto
 qed
qed

end
```

4.8 Aggregator

```
theory Aggregator
 imports Social-Choice-Types/Social-Choice-Result
begin
```

An aggregator gets two partitions (results of electoral modules) as input and output another partition. They are used to aggregate results of parallel composed electoral modules. They are commutative, i.e., the order of the aggregated modules does not affect the resulting aggregation. Moreover, they are conservative in the sense that the resulting decisions are subsets of the two given partitions' decisions.

4.8.1 Definition

```
type-synonym 'a Aggregator = 'a set ⇒ 'a Result ⇒ 'a Result ⇒ 'a Result

definition aggregator :: 'a Aggregator ⇒ bool where
 aggregator agg ≡
  ∀ A e e' d d' r r'.
   (well-formed-SCF A (e, r, d) ∧ well-formed-SCF A (e', r', d')) ⟶
   well-formed-SCF A (agg A (e, r, d) (e', r', d'))
```

4.8.2 Properties

```
definition agg-commutative :: 'a Aggregator ⇒ bool where
 agg-commutative agg ≡
  aggregator agg ∧ (∀ A e e' d d' r r'.
   agg A (e, r, d) (e', r', d') = agg A (e', r', d') (e, r, d))

definition agg-conservative :: 'a Aggregator ⇒ bool where
 agg-conservative agg ≡
  aggregator agg ∧
  (∀ A e e' d d' r r'.
   ((well-formed-SCF A (e, r, d) ∧ well-formed-SCF A (e', r', d')) ⟶
    elect-r (agg A (e, r, d) (e', r', d')) ⊆ (e ∪ e') ∧
    reject-r (agg A (e, r, d) (e′, r′, d′)) ⊆ (r ∪ r′) ∧
    defer-r (agg A (e, r, d) (e′, r′, d′)) ⊆ (d ∪ d′)))

end
```

4.9 Maximum Aggregator

```
theory Maximum-Aggregator
  imports Aggregator
begin
```

The max(imum) aggregator takes two partitions of an alternative set A as input. It returns a partition where every alternative receives the maximum result of the two input partitions.

4.9.1 Definition

```
fun max-aggregator :: ′a Aggregator where
  max-aggregator A (e, r, d) (e′, r′, d′) =
    (e ∪ e′,
     A − (e ∪ e′ ∪ d ∪ d′),
     (d ∪ d′) − (e ∪ e′))
```

4.9.2 Auxiliary Lemma

```
lemma max-agg-rej-set:
  fixes
    A e e′ d d′ r r′ :: ′a set and
    a :: ′a
  assumes
    wf-first-mod: well-formed-𝒮𝒞ℱ A (e, r, d) and
    wf-second-mod: well-formed-𝒮𝒞ℱ A (e′, r′, d′)
  shows reject-r (max-aggregator A (e, r, d) (e′, r′, d′)) = r ∩ r′
proof −
  have A − (e ∪ d) = r
    using wf-first-mod result-imp-rej
    by metis
  moreover have A − (e′ ∪ d′) = r′
    using wf-second-mod result-imp-rej
    by metis
  ultimately have A − (e ∪ e′ ∪ d ∪ d′) = r ∩ r′
    by blast
  moreover have {l ∈ A. l ∉ e ∪ e′ ∪ d ∪ d′} = A − (e ∪ e′ ∪ d ∪ d′)
    unfolding set-diff-eq
    by simp
  ultimately show reject-r (max-aggregator A (e, r, d) (e', r', d')) = r ∩ r'
    by simp
qed
```

4.9.3 Soundness

```
theorem max-agg-sound[simp]: aggregator max-aggregator
proof (unfold aggregator-def max-aggregator.simps well-formed-SCF.simps disjoint3.simps
            set-equals-partition.simps, safe)
  fix
    A e e' d d' r r' :: 'a set and
    a :: 'a
  assume
    e' ∪ r' ∪ d' = e ∪ r ∪ d and
    a ∉ d and
    a ∉ r and
    a ∈ e'
  thus a ∈ e
    by auto
next
  fix
    A e e' d d' r r' :: 'a set and
    a :: 'a
  assume
    e' ∪ r' ∪ d' = e ∪ r ∪ d and
    a ∉ d and
    a ∉ r and
    a ∈ d'
  thus a ∈ e
    by auto
qed
```

4.9.4 Properties

The max-aggregator is conservative.

```
theorem max-agg-consv[simp]: agg-conservative max-aggregator
proof (unfold agg-conservative-def, safe)
  show aggregator max-aggregator
    using max-agg-sound
    by metis
next
  fix
    A e e' d d' r r' :: 'a set and
    a :: 'a
  assume
    elect-a: a ∈ elect-r (max-aggregator A (e, r, d) (e', r', d')) and
    a-not-in-e': a ∉ e'
  have a ∈ e ∪ e'
    using elect-a
    by simp
  thus a ∈ e
    using a-not-in-e′
    by simp
next
  fix
    A e e′ d d′ r r′ :: ′a set and
    a :: ′a
  assume
    wf-result: well-formed-𝒮𝒞ℱ A (e′, r′, d′) and
    reject-a: a ∈ reject-r (max-aggregator A (e, r, d) (e′, r′, d′)) and
    a-not-in-r′: a ∉ r′
  have a ∈ r ∪ r′
    using wf-result reject-a
    by force
  thus a ∈ r
    using a-not-in-r′
    by simp
next
  fix
    A e e′ d d′ r r′ :: ′a set and
    a :: ′a
  assume
    defer-a: a ∈ defer-r (max-aggregator A (e, r, d) (e′, r′, d′)) and
    a-not-in-d′: a ∉ d′
  have a ∈ d ∪ d′
    using defer-a
    by force
  thus a ∈ d
    using a-not-in-d′
    by simp
qed
```

The max-aggregator is commutative.

```
theorem max-agg-comm[simp]: agg-commutative max-aggregator
  unfolding agg-commutative-def
  by auto

end
```

4.10 Termination Condition

```
theory Termination-Condition
  imports Social-Choice-Types/Result
begin
```

The termination condition is used in loops. It decides whether or not to terminate the loop after each iteration, depending on the current state of the loop.

type-synonym *'r Termination-Condition* = *'r Result* ⇒ *bool*

end

4.11 Defer Equal Condition

theory *Defer-Equal-Condition*
  imports *Termination-Condition*
begin

This is a family of termination conditions. For a natural number n, the according defer-equal condition is true if and only if the given result's defer-set contains exactly n elements.

fun *defer-equal-condition* :: *nat* ⇒ *'a Termination-Condition* **where**
  defer-equal-condition n (*e*, *r*, *d*) = (*card d* = *n*)

end

Chapter 5

Basic Modules

5.1 Defer Module

theory *Defer-Module*
  imports *Component-Types/Electoral-Module*
begin

The defer module is not concerned about the voter's ballots, and simply defers all alternatives. It is primarily used for defining an empty loop.

5.1.1 Definition

fun *defer-module* :: (*'a*, *'v*, *'a Result*) *Electoral-Module* **where**
  defer-module V A p = ({}, {}, *A*)

5.1.2 Soundness

theorem *def-mod-sound*[*simp*]: *$\mathcal{SCF}$-result.electoral-module defer-module*
  unfolding *$\mathcal{SCF}$-result.electoral-module.simps*
  by *simp*

5.1.3 Properties

theorem *def-mod-non-electing*: *non-electing defer-module*
  unfolding *non-electing-def*
  by *simp*

theorem *def-mod-def-lift-inv*: *defer-lift-invariance defer-module*
  unfolding *defer-lift-invariance-def*
  by *simp*

end

5.2 Elect-First Module

```
theory Elect-First-Module
  imports Component-Types/Electoral-Module
begin
```

The elect first module elects the alternative that is most preferred on the first ballot and rejects all other alternatives.

5.2.1 Definition

```
fun least :: 'v :: wellorder set ⇒ 'v where
  least V = (Least (λ v. v ∈ V))

fun elect-first-module :: ('a, 'v :: wellorder, 'a Result) Electoral-Module where
  elect-first-module V A p =
    ({a ∈ A. above (p (least V)) a = {a}},
    {a ∈ A. above (p (least V)) a ≠ {a}},
    {})
```

5.2.2 Soundness

```
theorem elect-first-mod-sound: SCF-result.electoral-module elect-first-module
proof (intro SCF-result.electoral-modI allI impI)
  fix
    A :: 'a set and
    V :: 'v :: wellorder set and
    p :: ('a, 'v) Profile
  have {a ∈ A. above (p (least V)) a = {a}}
        ∪ {a ∈ A. above (p (least V)) a ≠ {a}} = A
    by blast
  hence set-equals-partition A (elect-first-module V A p)
    by simp
  moreover have
    ∀ a ∈ A. (a ∉ {a' ∈ A. above (p (least V)) a' = {a'}} ∨
              a ∉ {a' ∈ A. above (p (least V)) a' ≠ {a'}})
    by simp
  hence {a ∈ A. above (p (least V)) a = {a}}
        ∩ {a ∈ A. above (p (least V)) a ≠ {a}} = {}
    by blast
  hence disjoint3 (elect-first-module V A p)
    by simp
  ultimately show well-formed-SCF A (elect-first-module V A p)
    by simp
qed

end
```

5.3 Consensus Class

```
theory Consensus-Class
  imports Consensus
          ../Defer-Module
          ../Elect-First-Module
begin
```

A consensus class is a pair of a set of elections and a mapping that assigns a unique alternative to each election in that set (of elections). This alternative is then called the consensus alternative (winner). Here, we model the mapping by an electoral module that defers alternatives which are not in the consensus.

5.3.1 Definition

```
type-synonym ('a, 'v, 'r) Consensus-Class =
    ('a, 'v) Consensus × ('a, 'v, 'r) Electoral-Module

fun consensus-𝒦 :: ('a, 'v, 'r) Consensus-Class ⇒ ('a, 'v) Consensus where
  consensus-𝒦 K = fst K

fun rule-𝒦 :: ('a, 'v, 'r) Consensus-Class ⇒ ('a, 'v, 'r) Electoral-Module where
  rule-𝒦 K = snd K
```

5.3.2 Consensus Choice

Returns those consensus elections on a given alternative and voter set from a given consensus that are mapped to the given unique winner by a given consensus rule.

```
fun 𝒦_ℰ :: ('a, 'v, 'r Result) Consensus-Class ⇒ 'r ⇒ ('a, 'v) Election set where
  𝒦_ℰ K w =
    {(A, V, p) | A V p. (consensus-𝒦 K) (A, V, p) ∧ finite-profile V A p
                ∧ elect (rule-𝒦 K) V A p = {w}}

fun elections-𝒦 :: ('a, 'v, 'r Result) Consensus-Class ⇒ ('a, 'v) Election set where
  elections-𝒦 K = ⋃ ((𝒦_ℰ K) ` UNIV)
```

A consensus class is deemed well-formed if the result of its mapping is completely determined by its consensus, the elected set of the electoral module's result.

```
definition well-formed :: ('a, 'v) Consensus ⇒ ('a, 'v, 'r) Electoral-Module ⇒
      bool where
  well-formed c m ≡
    ∀ A V V' p p'.
      profile V A p ∧ profile V' A p' ∧ c (A, V, p) ∧ c (A, V', p')
        ⟶ m V A p = m V' A p'
```

A sensible social choice rule for a given arbitrary consensus and social choice rule r is the one that chooses the result of r for all consensus elections and defers all candidates otherwise.

```
fun consensus-choice :: ('a, 'v) Consensus ⇒
      ('a, 'v, 'a Result) Electoral-Module ⇒
      ('a, 'v, 'a Result) Consensus-Class where
  consensus-choice c m =
    (let
      w = (λ V A p. if c (A, V, p) then m V A p else defer-module V A p)
      in (c, w))
```

5.3.3 Auxiliary Lemmas

```
lemma unanimity'-consensus-imp-elect-fst-mod-well-formed:
  fixes a :: 'a
  shows well-formed
      (λ c. nonempty-set_𝒞 c ∧ nonempty-profile_𝒞 c
            ∧ equal-top_𝒞' a c) elect-first-module
proof (unfold well-formed-def, safe)
  fix
    a :: 'a and
    A :: 'a set and
    V V' :: 'v :: wellorder set and
    p p' :: ('a, 'v) Profile
  let ?cond = λ c. nonempty-set_𝒞 c ∧ nonempty-profile_𝒞 c ∧ equal-top_𝒞' a c
  assume
    prof-p: profile V A p and
    prof-p': profile V' A p' and
    eq-top-p: equal-top_𝒞' a (A, V, p) and
    eq-top-p': equal-top_𝒞' a (A, V', p') and
    not-empty-A: nonempty-set_𝒞 (A, V, p) and
    not-empty-A': nonempty-set_𝒞 (A, V', p') and
    not-empty-p: nonempty-profile_𝒞 (A, V, p) and
    not-empty-p': nonempty-profile_𝒞 (A, V', p')
  hence
    cond-Ap: ?cond (A, V, p) and
    cond-Ap': ?cond (A, V', p')
    by simp-all
  have ∀ a' ∈ A.
    ((above (p (least V)) a' = {a'}) = (above (p' (least V')) a' = {a'}))
  proof
    fix a' :: 'a
    assume a'-in-A: a' ∈ A
    show (above (p (least V)) a' = {a'}) = (above (p' (least V')) a' = {a'})
    proof (cases)
      assume a' = a
      thus ?thesis
      using cond-Ap cond-Ap' Collect-mem-eq LeastI empty-Collect-eq equal-top_𝒞'.simps
            nonempty-profile_𝒞.simps least.simps
      by (metis (no-types, lifting))
    next
      assume a'-neq-a: a' ≠ a
      have non-empty: V ≠ {} ∧ V' ≠ {}
        using not-empty-p not-empty-p'
        by simp
      hence A ≠ {} ∧ linear-order-on A (p (least V))
              ∧ linear-order-on A (p' (least V'))
        using not-empty-A not-empty-A' prof-p prof-p' enumerate-0
              a'-in-A card.remove enumerate-in-set finite-enumerate-in-set
              least.elims all-not-in-conv zero-less-Suc
        unfolding profile-def
        by metis
      hence (a ∈ above (p (least V)) a' ∨ a' ∈ above (p (least V)) a)
          ∧ (a ∈ above (p' (least V')) a' ∨ a' ∈ above (p' (least V')) a)
        using a'-in-A a'-neq-a eq-top-p
        unfolding above-def linear-order-on-def total-on-def
        by auto
      hence
        (above (p (least V)) a = {a} ∧ above (p (least V)) a' = {a'}
            ⟶ a = a')
        ∧ (above (p' (least V')) a = {a} ∧ above (p' (least V')) a' = {a'}
            ⟶ a = a')
        by auto
      thus ?thesis
        using bot-nat-0.not-eq-extremum card-0-eq cond-Ap cond-Ap'
              enumerate-0 enumerate-in-set equal-top_𝒞'.simps
              finite-enumerate-in-set non-empty least.simps
        by metis
    qed
  qed
  thus elect-first-module V A p = elect-first-module V' A p'
    by auto
qed

lemma strong-unanimity'consensus-imp-elect-fst-mod-completely-determined:
  fixes r :: 'a Preference-Relation
  shows well-formed
      (λ c. nonempty-set_𝒞 c ∧ nonempty-profile_𝒞 c ∧ equal-vote_𝒞' r c) elect-first-module
proof (unfold well-formed-def, clarify)
 fix
    a :: 'a and
    A :: 'a set and
    V V' :: 'v :: wellorder set and
    p p' :: ('a, 'v) Profile
  let ?cond = λ c. nonempty-set_𝒞 c ∧ nonempty-profile_𝒞 c ∧ equal-vote_𝒞' r c
  assume
    prof-p: profile V A p and
    prof-p': profile V' A p' and
    eq-vote-p: equal-vote_𝒞' r (A, V,  p) and
    eq-vote-p': equal-vote_𝒞' r (A, V', p') and
    not-empty-A: nonempty-set_𝒞 (A, V, p) and
    not-empty-A': nonempty-set_𝒞 (A, V', p') and
    not-empty-p: nonempty-profile_𝒞 (A, V, p) and
    not-empty-p': nonempty-profile_𝒞 (A, V', p')
  hence
    cond-Ap: ?cond (A, V, p) and
    cond-Ap': ?cond (A, V', p')
    by simp-all
  have p (least V) = r ∧ p' (least V') = r
    using eq-vote-p eq-vote-p' not-empty-p not-empty-p'
          bot-nat-0.not-eq-extremum card-0-eq enumerate-0
          enumerate-in-set equal-vote_𝒞'.simps finite-enumerate-in-set
          nonempty-profile_𝒞.simps least.elims
    by (metis (no-types, lifting))
  thus elect-first-module V A p = elect-first-module V' A p'
    by auto
qed

lemma strong-unanimity'consensus-imp-elect-fst-mod-well-formed:
  fixes r :: 'a Preference-Relation
  shows well-formed
      (λ c. nonempty-set_𝒞 c ∧ nonempty-profile_𝒞 c
            ∧ equal-vote_𝒞' r c) elect-first-module
  using strong-unanimity'consensus-imp-elect-fst-mod-completely-determined
  by blast

lemma cons-domain-valid:
  fixes C :: ('a, 'v, 'r Result) Consensus-Class
  shows elections-𝒦 C ⊆ well-formed-elections
proof
  fix E :: ('a, 'v) Election
  assume E ∈ elections-𝒦 C
  hence fun_ℰ profile E
    unfolding 𝒦_ℰ.simps
    by force
  thus E ∈ well-formed-elections
    unfolding well-formed-elections-def
    by simp
qed

lemma cons-domain-finite:
  fixes C :: ('a, 'v, 'r Result) Consensus-Class
  shows
    finite: elections-𝒦 C ⊆ finite-elections and
    finite-voters: elections-𝒦 C ⊆ finite-elections-𝒱
proof −
  have ∀ E ∈ elections-𝒦 C.
```

```
      fun_ℰ profile E ∧ finite (alternatives-ℰ E) ∧ finite (voters-ℰ E)
    unfolding 𝒦_ℰ.simps
    by force
  thus elections-𝒦 C ⊆ finite-elections
    unfolding finite-elections-def fun_ℰ.simps
    by blast
  thus elections-𝒦 C ⊆ finite-elections-𝒱
    unfolding finite-elections-def finite-elections-𝒱-def
    by blast
qed
```

5.3.4 Consensus Rules

```
definition non-empty-set :: ('a, 'v, 'r) Consensus-Class ⇒ bool where
  non-empty-set c ≡ ∃ K. consensus-𝒦 c K
```

Unanimity condition.

```
definition unanimity :: ('a, 'v :: wellorder, 'a Result) Consensus-Class where
  unanimity ≡ consensus-choice unanimity_𝒞 elect-first-module
```

Strong unanimity condition.

```
definition strong-unanimity :: ('a, 'v :: wellorder, 'a Result) Consensus-Class
where
  strong-unanimity ≡ consensus-choice strong-unanimity_𝒞 elect-first-module
```

5.3.5 Properties

```
definition consensus-rule-anonymity :: ('a, 'v, 'r) Consensus-Class ⇒ bool where
  consensus-rule-anonymity c ≡
    (∀ A V p π :: ('v ⇒ 'v).
        bij π ⟶
          (let (A', V', q) = (rename π (A, V, p)) in
            profile V A p ⟶ profile V' A' q
             ⟶ consensus-𝒦 c (A, V, p)
           ⟶ (consensus-𝒦 c (A', V', q) ∧ (rule-𝒦 c V A p = rule-𝒦 c V' A' q))))

fun consensus-rule-anonymity' :: ('a, 'v) Election set ⇒
        ('a, 'v, 'r Result) Consensus-Class ⇒ bool where
  consensus-rule-anonymity' X C =
    is-symmetry (elect-r ∘ fun_ℰ (rule-𝒦 C)) (Invariance (anonymity_ℛ X))

fun (in result-properties) consensus-rule-neutrality :: ('a, 'v) Election set ⇒
        ('a, 'v, 'b Result) Consensus-Class ⇒ bool where
  consensus-rule-neutrality X C =
    is-symmetry (elect-r ∘ fun_ℰ (rule-𝒦 C))
      (action-induced-equivariance (carrier bijection_𝒜𝒢) X (φ-neutral X) (set-action
ψ))

fun consensus-rule-reversal-symmetry :: ('a, 'v) Election set ⇒
```

```
    ('a, 'v, 'a rel Result) Consensus-Class ⇒ bool where
  consensus-rule-reversal-symmetry X C = is-symmetry (elect-r ∘ funℰ (rule-𝒦 C))
    (action-induced-equivariance (carrier reversal𝒢) X (φ-reverse X) (set-action
ψ-reverse))
```

5.3.6 Inference Rules

```
lemma if-else-cons-equivar:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    c :: ('a, 'v) Consensus and
    G :: 'b set and
    X :: ('a, 'v) Election set and
    φ :: ('b, ('a, 'v) Election) binary-fun and
    ψ :: ('b, 'a) binary-fun and
    f :: 'a Result ⇒ 'a set
  defines
    equivar ≡ action-induced-equivariance G X φ (set-action ψ) and
    if-else-cons ≡ (c, (λ V A p. if c (A, V, p) then m V A p else n V A p))
  assumes
    equivar-m: is-symmetry (f ∘ funℰ m) equivar and
    equivar-n: is-symmetry (f ∘ funℰ n) equivar and
    invar-cons: is-symmetry c (Invariance (action-induced-rel G X φ))
  shows is-symmetry (f ∘ funℰ (rule-𝒦 if-else-cons))
            (action-induced-equivariance G X φ (set-action ψ))
proof (unfold rewrite-equivariance, intro ballI impI)
  fix
    E :: ('a, 'v) Election and
    g :: 'b
  assume
    g-in-G: g ∈ G and
    E-in-X: E ∈ X
  show (f ∘ funℰ (rule-𝒦 if-else-cons)) (φ g E) =
          set-action ψ g ((f ∘ funℰ (rule-𝒦 if-else-cons)) E)
  proof (cases c E)
    case True
    hence c (φ g E)
      using invar-cons rewrite-invar-ind-by-act g-in-G E-in-X
      by metis
    hence (f ∘ funℰ (rule-𝒦 if-else-cons)) (φ g E) =
        (f ∘ funℰ m) (φ g E)
      unfolding if-else-cons-def
      by simp
    also have (f ∘ funℰ m) (φ g E) =
      set-action ψ g ((f ∘ funℰ m) E)
      using equivar-m E-in-X g-in-G rewrite-equivariance
      unfolding equivar-def
      by (metis (mono-tags, lifting))
    also have (f ∘ funℰ m) E =
        (f ∘ fun_ℰ (rule-𝒦 if-else-cons)) E
      using True E-in-X g-in-G invar-cons if-else-cons-def
      by simp
    finally show ?thesis
      by simp
  next
    case False
    hence ¬ c (φ g E)
      using invar-cons rewrite-invar-ind-by-act g-in-G E-in-X
      by metis
    hence (f ∘ fun_ℰ (rule-𝒦 if-else-cons)) (φ g E) =
      (f ∘ fun_ℰ n) (φ g E)
      unfolding if-else-cons-def
      by simp
    also have (f ∘ fun_ℰ n) (φ g E) =
      set-action ψ g ((f ∘ fun_ℰ n) E)
      using equivar-n E-in-X g-in-G rewrite-equivariance
      unfolding equivar-def
      by (metis (mono-tags, lifting))
    also have (f ∘ fun_ℰ n) E =
      (f ∘ fun_ℰ (rule-𝒦 if-else-cons)) E
      using False E-in-X g-in-G invar-cons
      unfolding if-else-cons-def
      by simp
    finally show ?thesis
      by simp
  qed
qed

lemma consensus-choice-anonymous:
  fixes
    α β :: ('a, 'v) Consensus and
    m :: ('a, 'v, 'a Result) Electoral-Module and
    β' :: 'b ⇒ ('a, 'v) Consensus
  assumes
    beta-sat: β = (λ E. ∃ a. β' a E) and
    beta'-anon: ∀ x. consensus-anonymity (β' x) and
    anon-cons-cond: consensus-anonymity α and
    conditions-univ: ∀ x. well-formed (λ E. α E ∧ β' x E) m
  shows consensus-rule-anonymity (consensus-choice (λ E. α E ∧ β E) m)
proof (unfold consensus-rule-anonymity-def Let-def, safe)
  fix
    A A' :: 'a set and
    V V' :: 'v set and
    p q :: ('a, 'v) Profile and
    π :: 'v ⇒ 'v
  assume
    bij-π: bij π and
    prof-p: profile V A p and
    prof-q: profile V′ A′ q and
    renamed: rename π (A, V, p) = (A′, V′, q) and
    consensus-cond:
      consensus-𝒦 (consensus-choice (λ E. α E ∧ β E) m) (A, V, p)
  hence (λ E. α E ∧ β E) (A, V, p)
    by simp
  hence
    alpha-Ap: α (A, V, p) and
    beta-Ap: β (A, V, p)
    by simp-all
  have alpha-A-perm-p: α (A′, V′, q)
    using anon-cons-cond alpha-Ap bij-π prof-p prof-q renamed
    unfolding consensus-anonymity-def
    by fastforce
  moreover have β (A′, V′, q)
    using beta′-anon beta-Ap beta-sat
          ex-anon-cons-imp-cons-anonymous[of - β′] bij-π
          prof-p renamed beta′-anon cons-anon-invariant[of β]
    unfolding consensus-anonymity-def
    by blast
  ultimately show em-cond-perm:
    consensus-𝒦 (consensus-choice (λ E. α E ∧ β E) m) (A′, V′, q)
    using beta-Ap beta-sat ex-anon-cons-imp-cons-anonymous bij-π
          prof-p prof-q
    by simp
  have ∃ x. β′ x (A, V, p)
    using beta-Ap beta-sat
    by simp
  then obtain x :: ′b where
    beta′-x-Ap: β′ x (A, V, p)
    by metis
  hence beta′-x-A-perm-p: β′ x (A′, V′, q)
    using beta′-anon bij-π prof-p renamed
          cons-anon-invariant prof-q
    unfolding consensus-anonymity-def
    by blast
  have m V A p = m V′ A′ q
    using alpha-Ap alpha-A-perm-p beta′-x-Ap beta′-x-A-perm-p
          conditions-univ prof-p prof-q rename.simps prod.inject renamed
    unfolding well-formed-def
    by metis
  thus rule-𝒦 (consensus-choice (λ E. α E ∧ β E) m) V A p =
             rule-𝒦 (consensus-choice (λ E. α E ∧ β E) m) V′ A′ q
    using consensus-cond em-cond-perm
    by simp
qed
```

5.3.7 Theorems

Anonymity

```
lemma unanimity-anonymous: consensus-rule-anonymity unanimity
proof (unfold unanimity-def)
  let ?ne-cond = (λ c. nonempty-set_𝒞 c ∧ nonempty-profile_𝒞 c)
  have consensus-anonymity ?ne-cond
   using nonempty-set-cons-anonymous nonempty-profile-cons-anonymous cons-anon-conj
    by auto
  moreover have equal-top_𝒞 = (λ c. ∃ a. equal-top_𝒞' a c)
    by fastforce
  ultimately have consensus-rule-anonymity
     (consensus-choice
      (λ c. nonempty-set_𝒞 c ∧ nonempty-profile_𝒞 c ∧ equal-top_𝒞 c) elect-first-module)
    using consensus-choice-anonymous[of equal-top_𝒞] equal-top-cons'-anonymous
          unanimity'-consensus-imp-elect-fst-mod-well-formed
    by fastforce
  moreover have consensus-choice
    (λ c. nonempty-set_𝒞 c ∧ nonempty-profile_𝒞 c ∧ equal-top_𝒞 c)
      elect-first-module =
        consensus-choice unanimity_𝒞 elect-first-module
    using unanimity_𝒞.simps
    by metis
 ultimately show consensus-rule-anonymity (consensus-choice unanimity_𝒞 elect-first-module)
    by (metis (no-types))
qed

lemma strong-unanimity-anonymous: consensus-rule-anonymity strong-unanimity
proof (unfold strong-unanimity-def)
  have consensus-anonymity (λ c. nonempty-set_𝒞 c ∧ nonempty-profile_𝒞 c)
   using nonempty-set-cons-anonymous nonempty-profile-cons-anonymous cons-anon-conj
    unfolding consensus-anonymity-def
    by simp
  moreover have equal-vote_𝒞 = (λ c. ∃ v. equal-vote_𝒞' v c)
    by fastforce
  ultimately have consensus-rule-anonymity
      (consensus-choice
      (λ c. nonempty-set_𝒞 c ∧ nonempty-profile_𝒞 c ∧ equal-vote_𝒞 c) elect-first-module)
   using consensus-choice-anonymous[of equal-vote_𝒞] nonempty-set-cons-anonymous
         nonempty-profile-cons-anonymous eq-vote-cons'-anonymous
         strong-unanimity'consensus-imp-elect-fst-mod-well-formed
    by fastforce
  moreover have
    consensus-choice (λ c. nonempty-set_𝒞 c ∧ nonempty-profile_𝒞 c ∧ equal-vote_𝒞 c)
           elect-first-module =
             consensus-choice strong-unanimity_𝒞 elect-first-module
    unfolding strong-unanimity_𝒞.simps
    by metis
  ultimately show
```

```
    consensus-rule-anonymity (consensus-choice strong-unanimity_𝒞 elect-first-module)
      by (metis (no-types))
qed
```

Neutrality

```
lemma defer-winners-equivariant:
  fixes
    G :: 'b set and
    E :: ('a, 'v) Election set and
    φ :: ('b, ('a, 'v) Election) binary-fun and
    ψ :: ('b, 'a) binary-fun
  shows is-symmetry (elect-r ∘ fun_ℰ defer-module)
              (action-induced-equivariance G E φ (set-action ψ))
  using rewrite-equivariance
  by fastforce

lemma elect-first-winners-neutral: is-symmetry (elect-r ∘ fun_ℰ elect-first-module)
              (action-induced-equivariance (carrier bijection_𝒜𝒢)
                well-formed-elections (φ-neutral well-formed-elections)
                  (set-action ψ-neutral_c))
proof (unfold rewrite-equivariance, clarify)
  fix
    A :: 'a set and
    V :: 'v :: wellorder set and
    p :: ('a, 'v) Profile and
    π :: 'a ⇒ 'a
  assume
    bij-carrier-π: π ∈ carrier bijection_𝒜𝒢 and
    valid: (A, V, p) ∈ well-formed-elections
  hence bijective-π: bij π
    unfolding bijection_𝒜𝒢-def
    using rewrite-carrier
    by blast
  hence inv: ∀ a. a = π (the-inv π a)
    by (simp add: f-the-inv-into-f-bij-betw)
  from bij-carrier-π valid have
    (elect-r ∘ fun_ℰ elect-first-module)
        (φ-neutral well-formed-elections π (A, V, p)) =
      {a ∈ π ‘ A. above (rel-rename π (p (least V))) a = {a}}
    by simp
  moreover have
    {a ∈ π ‘ A. above (rel-rename π (p (least V))) a = {a}} =
      {a ∈ π ‘ A. {b. (a, b) ∈ {(π a, π b) | a b. (a, b) ∈ p (least V)}} = {a}}
    unfolding above-def
    by simp
  ultimately have elect-simp:
    (elect-r ∘ fun_ℰ elect-first-module)
        (φ-neutral well-formed-elections π (A, V, p)) =
      {a ∈ π ‘ A. {b. (a, b) ∈ {(π a, π b) | a b. (a, b) ∈ p (least V)}} = {a}}
    by simp
  have ∀ a ∈ π ‘ A. {b. (a, b) ∈ {(π x, π y) | x y. (x, y) ∈ p (least V)}} =
    {π b | b. (a, π b) ∈ {(π x, π y) | x y. (x, y) ∈ p (least V)}}
    by blast
  moreover have ∀ a ∈ π ‘ A.
    {π b | b. (a, π b) ∈ {(π x, π y) | x y. (x, y) ∈ p (least V)}} =
    {π b | b. (π (the-inv π a), π b) ∈ {(π x, π y) | x y. (x, y) ∈ p (least V)}}
    using bijective-π
    by (simp add: f-the-inv-into-f-bij-betw)
  moreover have ∀ a ∈ π ‘ A. ∀ b.
    ((π (the-inv π a), π b) ∈ {(π x, π y) | x y. (x, y) ∈ p (least V)}) =
      ((the-inv π a, b) ∈ {(x, y) | x y. (x, y) ∈ p (least V)})
    using bijective-π rel-rename-helper[of π]
    by auto
  moreover have {(x, y) | x y. (x, y) ∈ p (least V)} = p (least V)
    by simp
  ultimately have
    ∀ a ∈ π ‘ A. ({b. (a, b) ∈ {(π a, π b) | a b. (a, b) ∈ p (least V)}} = {a}) =
       ({π b | b. (the-inv π a, b) ∈ p (least V)} = {a})
    by force
  hence {a ∈ π ‘ A.
    {b. (a, b) ∈ {(π a, π b) | a b. (a, b) ∈ p (least V)}} = {a}} =
      {a ∈ π ‘ A. {π b | b. (the-inv π a, b) ∈ p (least V)} = {a}}
    by blast
  hence (elect-r ∘ fun_ℰ elect-first-module)
      (φ-neutral well-formed-elections π (A, V, p)) =
          {a ∈ π ‘ A. {π b | b. (the-inv π a, b) ∈ p (least V)} = {a}}
    using elect-simp
    by simp
  also have ... = π ‘ {a ∈ A. π ‘ {b | b. (a, b) ∈ p (least V)} = π ‘ {a}}
    using bijective-π inv bij-is-inj the-inv-f-f
    by fastforce
  finally have
    (elect-r ∘ fun_ℰ elect-first-module)
      (φ-neutral well-formed-elections π (A, V, p)) =
      π ‘ {a ∈ A. π ‘ (above (p (least V)) a) = π ‘ {a}}
    unfolding above-def
    by simp
  moreover have
    ∀ a. (π ‘ (above (p (least V)) a) = π ‘ {a}) =
      (the-inv π ‘ π ‘ above (p (least V)) a = the-inv π ‘ π ‘ {a})
    using bijective-π bij-betw-the-inv-into bij-def inj-image-eq-iff
    by metis
  moreover have
    ∀ a. (the-inv π ‘ π ‘ above (p (least V)) a = the-inv π ‘ π ‘ {a}) =
      (above (p (least V)) a = {a})
    using bijective-π bij-betw-imp-inj-on bij-betw-the-inv-into inj-image-eq-iff
    by metis
  ultimately have
    (elect-r ∘ fun_ℰ elect-first-module)
       (φ-neutral well-formed-elections π (A, V, p)) =
          π ‘ {a ∈ A. above (p (least V)) a = {a}}
    by presburger
  moreover have
    elect elect-first-module V A p = {a ∈ A. above (p (least V)) a = {a}}
    by simp
  moreover have set-action ψ-neutral_c π
              ((elect-r ∘ fun_ℰ elect-first-module) (A, V, p)) =
     π ‘ (elect elect-first-module V A p)
    by auto
  ultimately show
    (elect-r ∘ fun_ℰ elect-first-module)
       (φ-neutral well-formed-elections π (A, V, p)) =
     set-action ψ-neutral_c π
              ((elect-r ∘ fun_ℰ elect-first-module) (A, V, p))
    by blast
qed

lemma strong-unanimity-neutral:
  defines domain ≡ well-formed-elections ∩ Collect strong-unanimity_𝒞
  — We want to show neutrality on a set as general as possible, as this implies
subset neutrality.
  shows 𝒮𝒞ℱ-properties.consensus-rule-neutrality domain strong-unanimity
proof –
  have coincides:
    ∀ π. ∀ E ∈ domain. φ-neutral domain π E =
       φ-neutral well-formed-elections π E
    unfolding domain-def φ-neutral.simps
    by auto
  hence neutrality_ℛ domain ⊆ neutrality_ℛ well-formed-elections
    unfolding neutrality_ℛ.simps action-induced-rel.simps
    using domain-def
    by auto
  hence consensus-neutrality domain strong-unanimity_𝒞
    using strong-unanimity_𝒞-neutral invar-under-subset-rel
    unfolding consensus-neutrality.simps
    by blast
  hence is-symmetry strong-unanimity_𝒞
     (Invariance (action-induced-rel (carrier bijection_𝒜𝒢)
                 domain (φ-neutral well-formed-elections)))
    unfolding consensus-neutrality.simps neutrality_ℛ.simps
    using coincides coinciding-actions-ind-equal-rel
    by metis
  moreover have is-symmetry (elect-r ∘ fun_ℰ elect-first-module)
              (action-induced-equivariance (carrier bijection_𝒜𝒢)
                 domain (φ-neutral well-formed-elections) (set-action ψ-neutral_c))
    using elect-first-winners-neutral
```

unfolding *domain-def action-induced-equivariance-def*
using *equivar-under-subset*
by *blast*
ultimately have *is-symmetry* (*elect-r* ∘ *fun*$_{\mathcal{E}}$ (*rule-$\mathcal{K}$ strong-unanimity*))
(*action-induced-equivariance* (*carrier bijection*$_{\mathcal{AG}}$) *domain*
(φ*-neutral well-formed-elections*) (*set-action* ψ*-neutral*$_{\mathrm{c}}$))
using *defer-winners-equivariant*[*of* - *domain* - ψ*-neutral*$_{\mathrm{c}}$]
if-else-cons-equivar[*of* - - - *domain* - ψ*-neutral*$_{\mathrm{c}}$ - *strong-unanimity*$_{\mathcal{C}}$]
unfolding *strong-unanimity-def*
by *fastforce*
thus *?thesis*
unfolding *$\mathcal{SCF}$-properties.consensus-rule-neutrality.simps*
using *coincides equivar-ind-by-act-coincide*
by (*metis* (*no-types*, *lifting*))
qed

lemma *strong-unanimity-neutral'*: *$\mathcal{SCF}$-properties.consensus-rule-neutrality*
(*elections-$\mathcal{K}$ strong-unanimity*) *strong-unanimity*
proof −
have *elections-$\mathcal{K}$ strong-unanimity* ⊆ *well-formed-elections* ∩ *Collect strong-unanimity*$_{\mathcal{C}}$
unfolding *well-formed-elections-def* $\mathcal{K}_{\mathcal{E}}$*.simps strong-unanimity-def*
by *force*
moreover from *this* **have** *coincide*:
∀ π. ∀ *E* ∈ *elections-$\mathcal{K}$ strong-unanimity*.
φ*-neutral* (*well-formed-elections* ∩ *Collect strong-unanimity*$_{\mathcal{C}}$) π *E* =
φ*-neutral* (*elections-$\mathcal{K}$ strong-unanimity*) π *E*
unfolding φ*-neutral.simps*
using *extensional-continuation-subset*
by (*metis* (*no-types*, *lifting*))
ultimately have
is-symmetry (*elect-r* ∘ *fun*$_{\mathcal{E}}$ (*rule-$\mathcal{K}$ strong-unanimity*))
(*action-induced-equivariance* (*carrier bijection*$_{\mathcal{AG}}$) (*elections-$\mathcal{K}$ strong-unanimity*)
(φ*-neutral* (*well-formed-elections* ∩ *Collect strong-unanimity*$_{\mathcal{C}}$))
(*set-action* ψ*-neutral*$_{\mathrm{c}}$))
using *strong-unanimity-neutral equivar-under-subset*
unfolding *action-induced-equivariance-def $\mathcal{SCF}$-properties.consensus-rule-neutrality.simps*
by *blast*
thus *?thesis*
unfolding *$\mathcal{SCF}$-properties.consensus-rule-neutrality.simps*
using *coincide equivar-ind-by-act-coincide*
by (*metis* (*no-types*))
qed

lemma *strong-unanimity-closed-under-neutrality*: *closed-restricted-rel*
(*neutrality*$_{\mathcal{R}}$ *well-formed-elections*) *well-formed-elections*
(*elections-$\mathcal{K}$ strong-unanimity*)
proof (*unfold closed-restricted-rel.simps restricted-rel.simps neutrality*$_{\mathcal{R}}$*.simps*
action-induced-rel.simps elections-$\mathcal{K}$.simps, *safe*)
fix

```
  A A' :: 'a set and
  V V' :: 'b set and
  p p' :: ('a, 'b) Profile and
  π :: 'a ⇒ 'a and
  a :: 'a
assume
  prof: (A, V, p) ∈ well-formed-elections and
  cons: (A, V, p) ∈ 𝒦_ℰ strong-unanimity a and
  bij-carrier-π: π ∈ carrier bijection_𝒜𝒢 and
  img: φ-neutral well-formed-elections π (A, V, p) = (A', V', p')
hence fin: (A, V, p) ∈ finite-elections
  unfolding 𝒦_ℰ.simps finite-elections-def
  by simp
hence valid': (A', V', p') ∈ well-formed-elections
  using bij-carrier-π img φ-neutral-action.group-action-axioms
        group-action.element-image prof
  unfolding finite-elections-def
  by (metis (mono-tags, lifting))
moreover have V' = V ∧ A' = π ` A
  using img fin alternatives-rename.elims fstI prof sndI
  unfolding extensional-continuation.simps φ-neutral.simps
            alternatives-ℰ.simps voters-ℰ.simps
  by (metis (no-types, lifting))
ultimately have prof': finite-profile V' A' p'
  using fin bij-carrier-π CollectD finite-imageI fst-eqD snd-eqD
  unfolding finite-elections-def well-formed-elections-def alternatives-ℰ.simps
            voters-ℰ.simps profile-ℰ.simps
  by (metis (no-types, lifting))
let ?domain = well-formed-elections ∩ Collect strong-unanimity_𝒞
have ((A, V, p), (A', V', p')) ∈ neutrality_ℛ well-formed-elections
  using bij-carrier-π img fin valid'
  unfolding neutrality_ℛ.simps action-induced-rel.simps
            finite-elections-def well-formed-elections-def
  by blast
moreover have unanimous: (A, V, p) ∈ ?domain
  using cons fin
  unfolding 𝒦_ℰ.simps strong-unanimity-def well-formed-elections-def
  by simp
ultimately have unanimous': (A', V', p') ∈ ?domain
  using strong-unanimity_𝒞-neutral valid'
  unfolding consensus-neutrality.simps
  by force
have rewrite: ∀ π ∈ carrier bijection_𝒜𝒢.
    φ-neutral ?domain π (A, V, p) ∈ ?domain
      ⟶ (elect-r ∘ fun_ℰ (rule-𝒦 strong-unanimity))
            (φ-neutral ?domain π (A, V, p)) =
        set-action ψ-neutral_c π
          ((elect-r ∘ fun_ℰ (rule-𝒦 strong-unanimity)) (A, V, p))
  using strong-unanimity-neutral unanimous
```

```
      rewrite-equivariance[of elect-r ∘ fun_𝓔 (rule-𝒦 strong-unanimity) carrier
bijection_𝒜𝒢]
    unfolding 𝒮𝒞ℱ-properties.consensus-rule-neutrality.simps
    by metis
  have img': φ-neutral ?domain π (A, V, p) = (A', V', p')
    using img unanimous
    by simp
  hence elect (rule-𝒦 strong-unanimity) V' A' p' =
        (elect-r ∘ fun_𝓔 (rule-𝒦 strong-unanimity))
          (φ-neutral ?domain π (A, V, p))
    by simp
  also have
    ... = set-action ψ-neutral_c π
          ((elect-r ∘ fun_𝓔 (rule-𝒦 strong-unanimity)) (A, V, p))
    using bij-carrier-π img' unanimous' rewrite
    by metis
  also have ... = set-action ψ-neutral_c π {a}
    using cons
    unfolding 𝒦_𝓔.simps
    by simp
  finally have (A', V', p') ∈ elections-𝒦 strong-unanimity
    unfolding 𝒦_𝓔.simps strong-unanimity-def consensus-choice.simps
    using unanimous' prof'
    by simp
  hence ((A, V, p), (A', V', p'))
        ∈ ⋃ (range (𝒦_𝓔 strong-unanimity)) × ⋃ (range (𝒦_𝓔 strong-unanimity))
    unfolding elections-𝒦.simps
    using cons
    by blast
  moreover have
    ∃ π ∈ carrier bijection_𝒜𝒢.
      φ-neutral well-formed-elections π (A, V, p) = (A', V', p')
    using img bij-carrier-π
    unfolding bijection_𝒜𝒢-def
    by blast
  ultimately show (A', V', p') ∈ ⋃ (range (𝒦_𝓔 strong-unanimity))
    by blast
qed

end
```

5.4 Distance Rationalization

```
theory Distance-Rationalization
  imports Social-Choice-Types/Refined-Types/Preference-List
          Consensus-Class
          Distance
```

begin

A distance rationalization of a voting rule is its interpretation as a procedure that elects an uncontroversial winner if there is one, and otherwise elects the alternatives that are as close to becoming an uncontroversial winner as possible. Within general distance rationalization, a voting rule is characterized by a distance on profiles and a consensus class.

5.4.1 Definitions

Returns the distance of an election to the preimage of a unique winner under the given consensus elections and consensus rule.

fun $\mathit{score} :: ('a,\ 'v)\ \mathit{Election\ Distance} \Rightarrow ('a,\ 'v,\ 'r\ \mathit{Result})\ \mathit{Consensus\text{-}Class} \Rightarrow$
$\quad ('a,\ 'v)\ \mathit{Election} \Rightarrow\ 'r \Rightarrow \mathit{ereal}$ **where**
$\mathit{score}\ d\ K\ E\ w = \mathit{Inf}\ (d\ E\ `\ (\mathcal{K}_{\mathcal{E}}\ K\ w))$

fun (**in** *result*) $\mathcal{R}_{\mathcal{W}} :: ('a,\ 'v)\ \mathit{Election\ Distance} \Rightarrow$
$\quad ('a,\ 'v,\ 'r\ \mathit{Result})\ \mathit{Consensus\text{-}Class} \Rightarrow\ 'v\ \mathit{set} \Rightarrow\ 'a\ \mathit{set} \Rightarrow ('a,\ 'v)\ \mathit{Profile} \Rightarrow$
$\quad 'r\ \mathit{set}$ **where**
$\mathcal{R}_{\mathcal{W}}\ d\ K\ V\ A\ p = \mathit{arg\text{-}min\text{-}set}\ (\mathit{score}\ d\ K\ (A,\ V,\ p))\ (\mathit{limit}\ A\ \mathit{UNIV})$

fun (**in** *result*) $\mathit{distance\text{-}}\mathcal{R} :: ('a,\ 'v)\ \mathit{Election\ Distance} \Rightarrow$
$\quad ('a,\ 'v,\ 'r\ \mathit{Result})\ \mathit{Consensus\text{-}Class} \Rightarrow$
$\quad ('a,\ 'v,\ 'r\ \mathit{Result})\ \mathit{Electoral\text{-}Module}$ **where**
$\mathit{distance\text{-}}\mathcal{R}\ d\ K\ V\ A\ p =$
$\quad (\mathcal{R}_{\mathcal{W}}\ d\ K\ V\ A\ p,\ (\mathit{limit}\ A\ \mathit{UNIV}) - \mathcal{R}_{\mathcal{W}}\ d\ K\ V\ A\ p,\ \{\})$

5.4.2 Standard Definitions

definition $\mathit{standard} :: ('a,\ 'v)\ \mathit{Election\ Distance} \Rightarrow \mathit{bool}$ **where**
$\mathit{standard}\ d \equiv$
$\quad \forall\ A\ A'\ V\ V'\ p\ p'.\ (V \neq V' \vee A \neq A') \longrightarrow d\ (A,\ V,\ p)\ (A',\ V',\ p') = \infty$

definition $\mathit{voters\text{-}determine\text{-}distance} :: ('a,\ 'v)\ \mathit{Election\ Distance} \Rightarrow \mathit{bool}$ **where**
$\mathit{voters\text{-}determine\text{-}distance}\ d \equiv$
$\quad \forall\ A\ A'\ V\ V'\ p\ q\ p'.$
$\quad\quad (\forall\ v \in V.\ p\ v = q\ v)$
$\quad\quad\quad \longrightarrow (d\ (A,\ V,\ p)\ (A',\ V',\ p') = d\ (A,\ V,\ q)\ (A',\ V',\ p')$
$\quad\quad\quad\quad \wedge\ (d\ (A',\ V',\ p')\ (A,\ V,\ p) = d\ (A',\ V',\ p')\ (A,\ V,\ q)))$

Creates a set of all possible profiles on a finite alternative set that are empty everywhere outside of a given finite voter set.

fun $\mathit{profiles} ::\ 'v\ \mathit{set} \Rightarrow\ 'a\ \mathit{set} \Rightarrow (('a,\ 'v)\ \mathit{Profile})\ \mathit{set}$ **where**
$\mathit{profiles}\ V\ A =$
$\quad (\mathit{if}\ (\mathit{infinite}\ A \vee \mathit{infinite}\ V)$
$\quad\quad \mathit{then}\ \{\}\ \mathit{else}\ \{p.\ p\ `\ V \subseteq (\mathit{pl\text{-}}\alpha\ `\ \mathit{permutations\text{-}of\text{-}set}\ A)\})$

fun $\mathcal{K}_{\mathcal{E}}\mathit{\text{-}std} :: ('a,\ 'v,\ 'r\ \mathit{Result})\ \mathit{Consensus\text{-}Class} \Rightarrow\ 'r \Rightarrow\ 'a\ \mathit{set} \Rightarrow\ 'v\ \mathit{set} \Rightarrow$

```
        ('a, 'v) Election set where
  𝒦ℰ-std K w A V =
      (λ p. (A, V, p)) ` (Set.filter
          (λ p. (consensus-𝒦 K) (A, V, p) ∧ elect (rule-𝒦 K) V A p = {w})
          (profiles V A))
```

Returns those consensus elections on a given alternative and voter set from a given consensus that are mapped to the given unique winner by a given consensus rule.

```
fun score-std :: ('a, 'v) Election Distance ⇒ ('a, 'v, 'r Result) Consensus-Class ⇒
        ('a, 'v) Election ⇒ 'r ⇒ ereal where
  score-std d K E w =
        (if 𝒦ℰ-std K w (alternatives-ℰ E) (voters-ℰ E) = {}
          then ∞ else Min (d E ` (𝒦ℰ-std K w (alternatives-ℰ E) (voters-ℰ E))))

fun (in result) ℛ𝒲-std :: ('a, 'v) Election Distance ⇒
        ('a, 'v, 'r Result) Consensus-Class ⇒ 'v set ⇒ 'a set ⇒ ('a, 'v) Profile ⇒
        'r set where
  ℛ𝒲-std d K V A p = arg-min-set (score-std d K (A, V, p)) (limit A UNIV)

fun (in result) distance-ℛ-std :: ('a, 'v) Election Distance ⇒
        ('a, 'v, 'r Result) Consensus-Class ⇒
        ('a, 'v, 'r Result) Electoral-Module where
  distance-ℛ-std d K V A p =
    (ℛ𝒲-std d K V A p, (limit A UNIV) − ℛ𝒲-std d K V A p, {})
```

5.4.3 Auxiliary Lemmas

```
lemma fin-𝒦ℰ:
  fixes C :: ('a, 'v, 'r Result) Consensus-Class
  shows elections-𝒦 C ⊆ finite-elections
proof
  fix E :: ('a, 'v) Election
  assume E ∈ elections-𝒦 C
  hence finite-election E
    unfolding 𝒦ℰ.simps
    by force
  thus E ∈ finite-elections
    unfolding finite-elections-def
    by simp
qed

lemma univ-𝒦ℰ:
  fixes C :: ('a, 'v, 'r Result) Consensus-Class
  shows elections-𝒦 C ⊆ UNIV
  by simp

lemma listset-finiteness:
  fixes l :: 'a set list
  assumes ∀ i :: nat. i < length l ⟶ finite (l!i)
  shows finite (listset l)
  using assms
proof (induct l)
  case Nil
  show finite (listset [])
    by simp
next
  case (Cons a l)
  fix
    a :: 'a set and
    l :: 'a set list
  assume ∀ i :: nat < length (a#l). finite ((a#l)!i)
  hence
    finite a and
    ∀ i < length l. finite (l!i)
    by auto
  moreover assume
    ∀ i :: nat < length l. finite (l!i) ⟹ finite (listset l)
  ultimately have finite {a'#l' | a' l'. a' ∈ a ∧ l' ∈ (listset l)}
    using list-cons-presv-finiteness
    by blast
  thus finite (listset (a#l))
    by (simp add: set-Cons-def)
qed

lemma ls-entries-empty-imp-ls-set-empty:
  fixes l :: 'a set list
  assumes
    0 < length l and
    ∀ i :: nat. i < length l ⟶ l!i = {}
  shows listset l = {}
  using assms
proof (induct l)
  case Nil
  thus listset [] = {}
    by simp
next
  case (Cons a l)
  fix
    a :: 'a set and
    l :: 'a set list and
    l' :: 'a list
  assume all-elems-empty: ∀ i :: nat < length (a#l). (a#l)!i = {}
  hence a = {}
    by auto
  moreover from all-elems-empty
  have ∀ i < length l. l!i = {}
    by auto
  ultimately have {a′#l′ | a′ l′. a′ ∈ a ∧ l′ ∈ (listset l)} = {}
    by simp
  thus listset (a#l) = {}
    by (simp add: set-Cons-def)
qed

lemma all-ls-elems-same-len:
  fixes l :: ′a set list
  shows ∀ l′ :: ′a list. l′ ∈ listset l ⟶ length l′ = length l
proof (induct l, safe)
  case Nil
  fix l :: ′a list
  assume l ∈ listset []
  thus length l = length []
    by simp
next
  case (Cons a l)
  fix
    a :: ′a set and
    l :: ′a set list and
    l′ :: ′a list
  assume
    ∀ l′. l′ ∈ listset l ⟶ length l′ = length l and
    l′ ∈ listset (a#l)
  moreover have
    ∀ a′ l′ :: ′a set list.
      listset (a′#l′) = {b#m | b m. b ∈ a′ ∧ m ∈ listset l′}
    by (simp add: set-Cons-def)
  ultimately show length l′ = length (a#l)
    using local.Cons
    by fastforce
qed

lemma fin-all-profs:
  fixes
    A :: ′a set and
    V :: ′v set and
    x :: ′a Preference-Relation
  assumes
    fin-A: finite A and
    fin-V: finite V
  shows finite (profiles V A ∩ {p. ∀ v. v ∉ V ⟶ p v = x})
proof (cases A = {})
  let ?profs = profiles V A ∩ {p. ∀ v. v ∉ V ⟶ p v = x}
  case True
  hence permutations-of-set A = {[]}
    unfolding permutations-of-set-def
    by fastforce
  hence pl-α ‘ permutations-of-set A = {{}}
```

```
    unfolding pl-α-def
    by simp
  hence ∀ p ∈ profiles V A. ∀ v. v ∈ V ⟶ p v = {}
    by (simp add: image-subset-iff)
  hence ∀ p ∈ ?profs. (∀ v. v ∈ V ⟶ p v = {}) ∧ (∀ v. v ∉ V ⟶ p v = x)
    by simp
  hence ∀ p ∈ ?profs. p = (λ v. if v ∈ V then {} else x)
    by (metis (no-types, lifting))
  hence ?profs ⊆ {λ v. if v ∈ V then {} else x}
    by blast
  thus finite ?profs
    using finite.emptyI finite-insert finite-subset
    by (metis (no-types, lifting))
next
  let ?profs = (profiles V A ∩ {p. ∀ v. v ∉ V ⟶ p v = x})
  case False
  from fin-V obtain ord :: 'v rel where
    linear-order-on V ord
    using finite-list lin-ord-equiv lin-order-equiv-list-of-alts
    by metis
  then obtain list-V :: 'v list where
    len: length list-V = card V and
    pl: ord = pl-α list-V and
    perm: list-V ∈ permutations-of-set V
    using lin-order-pl-α fin-V image-iff length-finite-permutations-of-set
    by metis
  let ?map = λ p :: ('a, 'v) Profile. map p list-V
  have ∀ p ∈ profiles V A. ∀ v ∈ V. p v ∈ (pl-α ` permutations-of-set A)
    by (simp add: image-subset-iff)
  hence ∀ p ∈ profiles V A. (∀ v ∈ V. linear-order-on A (p v))
    using pl-α-lin-order fin-A False
    by metis
  moreover have ∀ p ∈ ?profs. ∀ i < length (?map p). (?map p)!i = p (list-V!i)
    by simp
  moreover have ∀ i < length list-V. list-V!i ∈ V
    using perm nth-mem
    unfolding permutations-of-set-def
    by safe
  moreover have lens-eq: ∀ p ∈ ?profs. length (?map p) = length list-V
    by simp
  ultimately have
    ∀ p ∈ ?profs. ∀ i < length (?map p). linear-order-on A ((?map p)!i)
    by simp
  hence subset-map-profs: ?map ` ?profs ⊆ {xs. length xs = card V ∧
                          (∀ i < length xs. linear-order-on A (xs!i))}
    using len lens-eq
    by fastforce
  have ∀ p1 p2.
      p1 ∈ ?profs ∧ p2 ∈ ?profs ∧ p1 ≠ p2 ⟶ (∃ v ∈ V. p1 v ≠ p2 v)
    by fastforce
  hence ∀ p1 p2.
      p1 ∈ ?profs ∧ p2 ∈ ?profs ∧ p1 ≠ p2
        ⟶ (∃ v ∈ set list-V. p1 v ≠ p2 v)
    using perm
    unfolding permutations-of-set-def
    by simp
  hence ∀ p1 p2. p1 ∈ ?profs ∧ p2 ∈ ?profs ∧ p1 ≠ p2 ⟶ ?map p1 ≠ ?map p2
    by simp
  hence inj-on ?map ?profs
    unfolding inj-on-def
    by blast
  moreover have
    finite {xs. length xs = card V ∧ (∀ i < length xs. linear-order-on A (xs!i))}
  proof –
    have finite {r. linear-order-on A r}
      using fin-A
      unfolding linear-order-on-def partial-order-on-def preorder-on-def refl-on-def
      by simp
    hence fin-supset:
      ∀ n. finite {xs. length xs = n ∧ set xs ⊆ {r. linear-order-on A r}}
      using Collect-mono finite-lists-length-eq rev-finite-subset
      by (metis (no-types, lifting))
    have ∀ l ∈ {xs. length xs = card V ∧
                        (∀ i < length xs. linear-order-on A (xs!i))}.
                set l ⊆ {r. linear-order-on A r}
      using in-set-conv-nth mem-Collect-eq subsetI
      by (metis (no-types, lifting))
    hence {xs. length xs = card V ∧
                        (∀ i < length xs. linear-order-on A (xs!i))}
          ⊆ {xs. length xs = card V ∧ set xs ⊆ {r. linear-order-on A r}}
      by blast
    thus ?thesis
      using fin-supset rev-finite-subset
      by blast
  qed
  moreover have ∀ f X Y. inj-on f X ∧ finite Y ∧ f ` X ⊆ Y ⟶ finite X
    using finite-imageD finite-subset
    by metis
  ultimately show finite ?profs
    using subset-map-profs
    by blast
qed

lemma profile-permutation-set:
  fixes
    A :: 'a set and
    V :: 'v set
  shows profiles V A = {p :: ('a, 'v) Profile. finite-profile V A p}
```

```
proof (cases finite A ∧ finite V ∧ A ≠ {})
  case True
  assume finite A ∧ finite V ∧ A ≠ {}
  hence
    fin-A: finite A and
    fin-V: finite V and
    non-empty: A ≠ {}
    by safe
  show profiles V A = {p'. finite-profile V A p'}
  proof (standard, clarify)
    fix p :: 'v ⇒ 'a Preference-Relation
    assume p ∈ profiles V A
    hence ∀ v ∈ V. p v ∈ pl-α ‘ permutations-of-set A
      using fin-A fin-V
      by auto
    hence ∀ v ∈ V. linear-order-on A (p v)
      using fin-A pl-α-lin-order non-empty
      by metis
    thus finite-profile V A p
      unfolding profile-def
      using fin-A fin-V
      by blast
  next
    show {p. finite-profile V A p} ⊆ profiles V A
    proof (standard, clarify)
      fix p :: ('a, 'v) Profile
      assume prof: profile V A p
      have p ∈ {p. p ‘ V ⊆ (pl-α ‘ permutations-of-set A)}
        using fin-A lin-order-pl-α prof
        unfolding profile-def
        by blast
      thus p ∈ profiles V A
        using fin-A fin-V
        unfolding profiles.simps
        by metis
    qed
  qed
next
  case False
  assume not-fin-empty: ¬ (finite A ∧ finite V ∧ A ≠ {})
  have finite A ∧ finite V ∧ A = {} ⟶ permutations-of-set A = {[]}
    unfolding permutations-of-set-def
    by fastforce
  hence pl-empty:
    finite A ∧ finite V ∧ A = {} ⟶ pl-α ‘ permutations-of-set A = {{}}
    unfolding pl-α-def
    by simp
  hence finite A ∧ finite V ∧ A = {} ⟶
    (∀ π ∈ {π. π ‘ V ⊆ (pl-α ‘ permutations-of-set A)}. ∀ v ∈ V. π v = {})
    by fastforce
  hence finite A ∧ finite V ∧ A = {} ⟶
    {π. π ` V ⊆ (pl-α ` permutations-of-set A)} = {π. ∀ v ∈ V. π v = {}}
    using image-subset-iff singletonD singletonI pl-empty
    by fastforce
  moreover have finite A ∧ finite V ∧ A = {}
    ⟶ profiles V A = {π. π ` V ⊆ (pl-α ` permutations-of-set A)}
    by simp
  ultimately have all-prof-eq: finite A ∧ finite V ∧ A = {}
    ⟶ profiles V A = {π. ∀ v ∈ V. π v = {}}
    by simp
  have finite A ∧ finite V ∧ A = {}
    ⟶ (∀ p ∈ {p. finite-profile V A p ∧ (∀ v. v ∉ V ⟶ p v = {})}.
      (∀ v ∈ V. linear-order-on {} (p v)))
    unfolding profile-def
    by simp
  moreover have ∀ r. linear-order-on {} r ⟶ r = {}
    using lin-ord-not-empty
    by metis
  ultimately have non-voters:
    finite A ∧ finite V ∧ A = {}
    ⟶ (∀ p ∈ {p. finite-profile V A p ∧ (∀ v. v ∉ V ⟶ p v = {})}.
      ∀ v. p v = {})
    by blast
  hence (∀ p. profile V {} p ∧ (∀ v. v ∉ V ⟶ p v = {})
          ⟶ (∀ v. p v = {})) ⟶ finite V ⟶ A = {}
    ⟶ {p. profile V {} p} = {p. ∀ v ∈ V. p v = {}}
    unfolding profile-def
    using lin-ord-not-empty
    by auto
  hence finite A ∧ finite V ∧ A = {}
    ⟶ ({p. finite-profile V A p} = {p. ∀ v ∈ V. p v = {}})
    using non-voters
    by blast
  hence finite A ∧ finite V ∧ A = {}
    ⟶ profiles V A = {p. finite-profile V A p}
    using all-prof-eq
    by simp
  moreover have infinite A ∨ infinite V ⟶ profiles V A = {}
    by simp
  moreover have infinite A ∨ infinite V ⟶
    {p. finite-profile V A p ∧ (∀ v. v ∉ V ⟶ p v = {})} = {}
    by auto
  moreover have infinite A ∨ infinite V ∨ A = {}
    using not-fin-empty
    by simp
  ultimately show profiles V A = {p. finite-profile V A p}
    by blast
qed
```

5.4.4 Soundness

```
lemma (in result) ℛ-sound:
  fixes
    K :: ('a, 'v, 'r Result) Consensus-Class and
    d :: ('a, 'v) Election Distance
  shows electoral-module (distance-ℛ d K)
proof (unfold electoral-module.simps, safe)
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  have ℛ𝒲 d K V A p ⊆ (limit A UNIV)
    using ℛ𝒲.simps arg-min-subset
    by metis
  hence set-equals-partition (limit A UNIV) (distance-ℛ d K V A p)
    by auto
  moreover have disjoint3 (distance-ℛ d K V A p)
    by simp
  ultimately show well-formed A (distance-ℛ d K V A p)
    using result-axioms
    unfolding result-def
    by simp
qed
```

5.4.5 Properties

```
fun distance-decisiveness :: ('a, 'v) Election set ⇒ ('a, 'v) Election Distance ⇒
        ('a, 'v, 'r Result) Electoral-Module ⇒ bool where
  distance-decisiveness X d m =
    (∄ E. E ∈ X
    ∧ (∃ δ > 0. ∀ E' ∈ X. d E E' < δ ⟶ card (elect-r (fun𝓔 m E')) > 1))
```

5.4.6 Inference Rules

```
lemma (in result) standard-distance-imp-equal-score:
  fixes
    d :: ('a, 'v) Election Distance and
    K :: ('a, 'v, 'r Result) Consensus-Class and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    w :: 'r
  assumes
    irr-non-V: voters-determine-distance d and
    std: standard d
  shows score d K (A, V, p) w = score-std d K (A, V, p) w
proof –
  have profile-perm-set:
    profiles V A =
    {p′ :: (′a, ′v) Profile. finite-profile V A p′}
  using profile-permutation-set
  by metis
hence eq-intersect: 𝒦ℰ-std K w A V =
        𝒦ℰ K w ∩ Pair A ‘ Pair V ‘ {p′ :: (′a, ′v) Profile. finite-profile V A p′}
  by force
have (𝒦ℰ K w ∩ Pair A ‘ Pair V ‘ {p′ :: (′a, ′v) Profile. finite-profile V A p′})
        ⊆ (𝒦ℰ K w)
  by simp
hence Inf (d (A, V, p) ‘ (𝒦ℰ K w)) ≤
               Inf (d (A, V, p) ‘ (𝒦ℰ K w ∩
                Pair A ‘ Pair V ‘ {p′ :: (′a, ′v) Profile. finite-profile V A p′}))
  using INF-superset-mono dual-order.refl
  by metis
moreover have Inf (d (A, V, p) ‘ (𝒦ℰ K w)) ≥
               Inf (d (A, V, p) ‘ (𝒦ℰ K w ∩
                Pair A ‘ Pair V ‘ {p′ :: (′a, ′v) Profile. finite-profile V A p′}))
proof (rule INF-greatest)
  let ?inf = Inf (d (A, V, p) ‘
    (𝒦ℰ K w ∩ Pair A ‘ Pair V ‘ {p′. finite-profile V A p′}))
  let ?compl = (𝒦ℰ K w) −
    (𝒦ℰ K w ∩ Pair A ‘ Pair V ‘ {p′. finite-profile V A p′})
  fix i :: (′a, ′v) Election
  assume el: i ∈ 𝒦ℰ K w
  have in-intersect:
    i ∈ (𝒦ℰ K w ∩ Pair A ‘ Pair V ‘ {p′. finite-profile V A p′})
          ⟶ ?inf ≤ d (A, V, p) i
    using Complete-Lattices.complete-lattice-class.INF-lower
    by metis
  have compl-imp-neither-voter-nor-alt-nor-infinite-prof:
       i ∈ ?compl ⟶ (V ≠ fst (snd i)
                        ∨ A ≠ fst i
                        ∨ ¬ finite-profile V A (snd (snd i)))
    by fastforce
  moreover have not-voters-imp-infty: V ≠ fst (snd i) ⟶ d (A, V, p) i = ∞
    using std prod.collapse
    unfolding standard-def
    by metis
  moreover have not-alts-imp-infty: A ≠ fst i ⟶ d (A, V, p) i = ∞
    using std prod.collapse
    unfolding standard-def
    by metis
  moreover have V = fst (snd i) ∧ A = fst i
                  ∧ ¬ finite-profile V A (snd (snd i)) ⟶ False
    using el
    by fastforce
  hence i ∈ ?compl ⟶ d (A, V, p) i = ∞
    using not-alts-imp-infty not-voters-imp-infty
          compl-imp-neither-voter-nor-alt-nor-infinite-prof
    by fastforce
  ultimately have
    i ∈ ?compl
      ⟶ Inf (d (A, V, p)
            ‘ (𝒦ℰ K w ∩ Pair A ‘ Pair V ‘ {p'. finite-profile V A p'}))
          ≤ d (A, V, p) i
    using ereal-less-eq
    by (metis (no-types, lifting))
  thus Inf (d (A, V, p) ‘
          (𝒦ℰ K w ∩
           Pair A ‘ Pair V ‘ {p'. finite-profile V A p'}))
          ≤ d (A, V, p) i
    using in-intersect el
    by blast
qed
ultimately have Inf (d (A, V, p) ‘ 𝒦ℰ K w) =
        Inf (d (A, V, p) ‘
          (𝒦ℰ K w ∩ Pair A ‘ Pair V ‘ {p'. finite-profile V A p'}))
  using order-antisym
  by simp
also have inf-eq-min-for-std-cons: . . . = score-std d K (A, V, p) w
proof (cases 𝒦ℰ-std K w A V = {})
  case True
  hence Inf (d (A, V, p) ‘
        (𝒦ℰ K w ∩ Pair A ‘ Pair V ‘
          {p'. finite-profile V A p'})) = ∞
    using eq-intersect top-ereal-def
    by simp
  also have score-std d K (A, V, p) w = ∞
    using True
    unfolding Let-def
    by simp
  finally show ?thesis
    by simp
next
  case False
  hence fin: finite A ∧ finite V
    using eq-intersect
    by blast
  have 𝒦ℰ-std K w A V =
          (𝒦ℰ K w) ∩ {(A, V, p') | p'. finite-profile V A p'}
    using eq-intersect
    by blast
  hence subset-dist-𝒦ℰ-std:
      d (A, V, p) ‘(𝒦ℰ-std K w A V) ⊆
          d (A, V, p) ‘ {(A, V, p') | p'. finite-profile V A p'}
    by blast
  let ?finite-prof = λ p' v. (if (v ∈ V) then p' v else {})
  have ∀ p'. finite-profile V A p' ⟶
```

```
          finite-profile V A (?finite-prof p′)
  unfolding If-def profile-def
  by simp
moreover have ∀ p′. (∀ v. v ∉ V ⟶ ?finite-prof p′ v = {})
  by simp
ultimately have
  ∀ (A′, V′, p′) ∈ {(A′, V′, p′). A′ = A ∧ V′ = V ∧ finite-profile V A p′}.
      (A′, V′, ?finite-prof p′) ∈ {(A, V, p′) | p′. finite-profile V A p′}
  by force
moreover have
  ∀ p′. d (A, V, p) (A, V, p′) = d (A, V, p) (A, V, ?finite-prof p′)
  using irr-non-V
  unfolding voters-determine-distance-def
  by simp
ultimately have
  ∀ (A′, V′, p′) ∈ {(A, V, p′) | p′. finite-profile V A p′}.
      (∃ (X, Y, z) ∈ {(A, V, p′) | p′.
        finite-profile V A p′ ∧ (∀ v. v ∉ V ⟶ p′ v = {})}.
        d (A, V, p) (A′, V′, p′) = d (A, V, p) (X, Y, z))
  by fastforce
hence
  ∀ (A′, V′, p′)
     ∈ {(A′, V′, p′). A′ = A ∧ V′ = V ∧ finite-profile V A p′}.
        d (A, V, p) (A′, V′, p′) ∈
          d (A, V, p) ‘ {(A, V, p′) | p′.
            finite-profile V A p′ ∧ (∀ v. v ∉ V ⟶ p′ v = {})}
  by fastforce
hence subset-dist-restrict-non-voters:
  d (A, V, p) ‘ {(A, V, p′) | p′. finite-profile V A p′}
      ⊆ d (A, V, p) ‘ {(A, V, p′) | p′.
          finite-profile V A p′ ∧ (∀ v. v ∉ V ⟶ p′ v = {})}
  by fastforce
have ∀ (A′, V′, p′) ∈ {(A, V, p′) | p′.
      finite-profile V A p′ ∧ (∀ v. v ∉ V ⟶ p′ v = {})}.
        (∀ v ∈ V. linear-order-on A (p′ v))
                  ∧ (∀ v. v ∉ V ⟶ p′ v = {})
  using fin
  unfolding profile-def
  by simp
hence subset-lin-ord:
  {(A, V, p′) | p′. finite-profile V A p′ ∧ (∀ v. v ∉ V ⟶ p′ v = {})}
      ⊆ {(A, V, p′) | p′. p′ ∈ {p′.
        (∀ v ∈ V. linear-order-on A (p′ v)) ∧ (∀ v. v ∉ V ⟶ p′ v = {})}}
  by blast
have {p′. (∀ v ∈ V. linear-order-on A (p′ v))
          ∧ (∀ v. v ∉ V ⟶ p′ v = {})}
        ⊆ profiles V A ∩ {p. ∀ v. v ∉ V ⟶ p v = {}}
  using lin-order-pl-α fin
  by fastforce
```

```
    moreover have finite (profiles V A ∩ {p. ∀ v. v ∉ V ⟶ p v = {}})
      using fin fin-all-profs
      by blast
    ultimately have
      finite {p'. (∀ v ∈ V.
          linear-order-on A (p' v)) ∧ (∀ v. v ∉ V ⟶ p' v = {})}
      using rev-finite-subset
      by blast
    hence finite {(A, V, p') | p'. p' ∈ {p'.
            (∀ v ∈ V. linear-order-on A (p' v)) ∧ (∀ v. v ∉ V ⟶ p' v = {})}}
      by simp
    hence finite {(A, V, p') | p'.
              finite-profile V A p' ∧ (∀ v. v ∉ V ⟶ p' v = {})}
      using subset-lin-ord rev-finite-subset
      by simp
    hence finite (d (A, V, p) ` {(A, V, p') | p'.
              finite-profile V A p' ∧ (∀ v. v ∉ V ⟶ p' v = {})})
      by simp
    hence finite (d (A, V, p) ` {(A, V, p') | p'. finite-profile V A p'})
      using subset-dist-restrict-non-voters rev-finite-subset
      by simp
    hence finite (d (A, V, p) `(𝒦_ℰ-std K w A V))
      using subset-dist-𝒦_ℰ-std rev-finite-subset
      by blast
    moreover have d (A, V, p) ` (𝒦_ℰ-std K w A V) ≠ {}
      using False
      by simp
    ultimately have
      Inf (d (A, V, p) ` (𝒦_ℰ-std K w A V)) =
          Min (d (A, V, p) ` (𝒦_ℰ-std K w A V))
      using Min-Inf False
      by metis
    also have ... = score-std d K (A, V, p) w
      using False
      by simp
    also have Inf (d (A, V, p) ` (𝒦_ℰ-std K w A V)) =
      Inf (d (A, V, p) ` (𝒦_ℰ K w ∩
        Pair A ` Pair V ` {p'. finite-profile V A p'}))
      using eq-intersect
      by simp
    ultimately show ?thesis
      by simp
  qed
  finally show score d K (A, V, p) w = score-std d K (A, V, p) w
    by simp
qed

lemma (in result) anonymous-distance-and-consensus-imp-rule-anonymity:
  fixes
```

```
    d :: ('a, 'v) Election Distance and
    K :: ('a, 'v, 'r Result) Consensus-Class
  assumes
    d-anon: distance-anonymity d and
    K-anon: consensus-rule-anonymity K
  shows anonymity (distance-ℛ d K)
proof (unfold anonymity-def Let-def, safe)
  show electoral-module (distance-ℛ d K)
    using ℛ-sound
    by metis
next
  fix
    A A' :: 'a set and
    V V' :: 'v set and
    p q :: ('a, 'v) Profile and
    π :: 'v ⇒ 'v
  assume
    bijective: bij π and
    renamed: rename π (A, V, p) = (A', V', q)
  hence eq-univ: limit A UNIV = limit A' UNIV
    by simp
  have dist-rename-inv:
    ∀ E :: ('a, 'v) Election. d (A, V, p) E = d (A', V', q) (rename π E)
    using d-anon bijective renamed surj-pair
    unfolding distance-anonymity-def
    by metis
  hence ∀ S :: ('a, 'v) Election set.
            (d (A, V, p) ‘ S) ⊆ (d (A', V', q) ‘ (rename π ‘ S))
    by blast
  moreover have
    ∀ S :: ('a, 'v) Election set.
        ((d (A', V', q) ‘ (rename π ‘ S)) ⊆ (d (A, V, p) ‘ S))
  proof (clarify)
    fix
      S :: ('a, 'v) Election set and
      X X' :: 'a set and
      Y Y' :: 'v set and
      z z' :: ('a, 'v) Profile
    assume (X', Y', z') = rename π (X, Y, z)
    hence d (A', V', q) (X', Y', z') = d (A, V, p) (X, Y, z)
      using dist-rename-inv
      by metis
    moreover assume (X, Y, z) ∈ S
    ultimately show d (A', V', q) (X', Y', z') ∈ d (A, V, p) ‘ S
      by simp
  qed
  ultimately have eq-range:
    ∀ S :: ('a, 'v) Election set.
        (d (A, V, p) ‘ S) = (d (A', V', q) ‘ (rename π ‘ S))
    by blast
  have ∀ w. rename π ` (𝒦_ℰ K w) ⊆ (𝒦_ℰ K w)
  proof (clarify)
    fix
      w :: 'r and
      A A' :: 'a set and
      V V' :: 'v set and
      p p' :: ('a, 'v) Profile
    assume (A, V, p) ∈ 𝒦_ℰ K w
    hence cons:
      (consensus-𝒦 K) (A, V, p) ∧ finite-profile V A p
        ∧ elect (rule-𝒦 K) V A p = {w}
      by simp
    moreover assume renamed: (A', V', p') = rename π (A, V, p)
    ultimately have finite-profile V' A' p'
      using bijective fst-conv rename-finite rename-prof
      unfolding rename.simps
      by metis
    moreover from this have cons-img:
      consensus-𝒦 K (A', V', p') ∧ (rule-𝒦 K V A p = rule-𝒦 K V' A' p')
      using K-anon renamed bijective cons
      unfolding consensus-rule-anonymity-def Let-def
      by simp
    ultimately show (A', V', p') ∈ 𝒦_ℰ K w
      using cons
      by simp
  qed
  moreover have ∀ w. (𝒦_ℰ K w) ⊆ rename π ` (𝒦_ℰ K w)
  proof (clarify)
    fix
      w :: 'r and
      A :: 'a set and
      V :: 'v set and
      p :: ('a, 'v) Profile
    assume (A, V, p) ∈ 𝒦_ℰ K w
    hence cons:
      (consensus-𝒦 K) (A, V, p) ∧ finite-profile V A p
            ∧ elect (rule-𝒦 K) V A p = {w}
      by simp
    let ?inv = rename (the-inv π) (A, V, p)
    have inv-inv-id: the-inv (the-inv π) = π
      using the-inv-f-f bijective bij-betw-imp-inj-on bij-betw-imp-surj
            inj-on-the-inv-into surj-imp-inv-eq the-inv-into-onto
      by (metis (no-types, opaque-lifting))
    hence ?inv = (A, ((the-inv π) ` V), p ∘ (the-inv (the-inv π)))
      by simp
    moreover have (p ∘ (the-inv (the-inv π))) ∘ (the-inv π) = p
      using bijective inv-inv-id
      unfolding bij-betw-def comp-def
```

```
      by (simp add: f-the-inv-into-f)
    moreover have π ‘ (the-inv π) ‘ V = V
      using bijective the-inv-f-f image-inv-into-cancel top-greatest
            surj-imp-inv-eq
      unfolding bij-betw-def
      by (metis (no-types, opaque-lifting))
    ultimately have preimg: rename π ?inv = (A, V, p)
      unfolding Let-def
      by simp
    have bij (the-inv π)
      using bijective bij-betw-the-inv-into
      by metis
    moreover from this have fin-preimg:
      finite-profile (fst (snd ?inv)) (fst ?inv) (snd (snd ?inv))
      using rename-prof cons
      by fastforce
    ultimately have
      consensus-𝒦 K ?inv ∧
          (rule-𝒦 K V A p =
              rule-𝒦 K (fst (snd ?inv)) (fst ?inv) (snd (snd ?inv)))
      using K-anon renamed bijective cons
      unfolding consensus-rule-anonymity-def Let-def
      by simp
    moreover from this have
      elect (rule-𝒦 K) (fst (snd ?inv)) (fst ?inv) (snd (snd ?inv)) = {w}
      using cons
      by simp
    ultimately have ?inv ∈ 𝒦_ℰ K w
      using fin-preimg
      by simp
    thus (A, V, p) ∈ rename π ‘ 𝒦_ℰ K w
      using preimg image-eqI
      by metis
  qed
  ultimately have ∀ w. (𝒦_ℰ K w) = rename π ‘ (𝒦_ℰ K w)
    by blast
  hence ∀ w. score d K (A, V, p) w = score d K (A′, V′, q) w
    using eq-range
    by simp
  hence arg-min-set (score d K (A, V, p)) (limit A UNIV) =
            arg-min-set (score d K (A′, V′, q)) (limit A′ UNIV)
    using eq-univ
    by presburger
  hence ℛ_𝒲 d K V A p = ℛ_𝒲 d K V′ A′ q
    by simp
  thus distance-ℛ d K V A p = distance-ℛ d K V′ A′ q
    using eq-univ
    by simp
qed
```

```
end
```

5.5 Votewise Distance Rationalization

```
theory Votewise-Distance-Rationalization
  imports Distance-Rationalization
          Votewise-Distance
begin
```

A votewise distance rationalization of a voting rule is its distance rationalization with a distance function that depends on the submitted votes in a simple and a transparent manner by using a distance on individual orders and combining the components with a norm on R to n.

5.5.1 Common Rationalizations

```
fun swap-ℛ :: ('a, 'v :: linorder, 'a Result) Consensus-Class ⇒
      ('a, 'v, 'a Result) Electoral-Module where
  swap-ℛ K = 𝒮𝒞ℱ-result.distance-ℛ (votewise-distance swap l-one) K
```

5.5.2 Theorems

```
lemma votewise-non-voters-irrelevant:
  fixes
    d :: 'a Vote Distance and
    N :: Norm
  shows voters-determine-distance (votewise-distance d N)
proof (unfold voters-determine-distance-def, clarify)
  fix
    A A' :: 'a set and
    V V' :: 'v :: linorder set and
    p p' q :: ('a, 'v) Profile
  assume coincide: ∀ v ∈ V. p v = q v
  have ∀ i < length (sorted-list-of-set V). (sorted-list-of-set V)!i ∈ V
    using card-eq-0-iff not-less-zero nth-mem
          sorted-list-of-set.length-sorted-key-list-of-set
          sorted-list-of-set.set-sorted-key-list-of-set
    by metis
  hence (to-list V p) = (to-list V q)
    using coincide length-map nth-equalityI to-list.simps
    by auto
  thus votewise-distance d N (A, V, p) (A', V', p') =
           votewise-distance d N (A, V, q) (A', V', p') ∧
        votewise-distance d N (A', V', p') (A, V, p) =
           votewise-distance d N (A', V', p') (A, V, q)
    unfolding votewise-distance.simps
    by presburger
qed

lemma swap-standard: standard (votewise-distance swap l-one)
proof (unfold standard-def, clarify)
  fix
    A A' :: 'a set and
    V V' :: 'v :: linorder set and
    p p' :: ('a, 'v) Profile
  assume assms: V ≠ V' ∨ A ≠ A'
  let ?l = (λ l1 l2. (map2 (λ q q'. swap (A, q) (A', q')) l1 l2))
  have A ≠ A' ∧ V = V' ∧ V ≠ {} ∧ finite V ⟶
    (∀ l1 l2. l1 ≠ [] ∧ l2 ≠ [] ⟶ (∀ i < length (?l l1 l2). (?l l1 l2)!i = ∞))
    by simp
  moreover have
    V = V' ∧ V ≠ {} ∧ finite V ⟶ (to-list V p) ≠ [] ∧ (to-list V' p') ≠ []
    using sorted-list-of-set.sorted-key-list-of-set-eq-Nil-iff
          to-list.simps Nil-is-map-conv
    by (metis (no-types))
  moreover have ∀ l. (∃ i < length l. l!i = ∞) ⟶ l-one l = ∞
  proof (safe)
    fix
      l :: ereal list and
      i :: nat
    assume
      i < length l and
      l ! i = ∞
    hence (∑ j < length l. |l!j|) = ∞
      using sum-Pinfty finite-lessThan lessThan-iff abs-ereal.simps
      by metis
    thus l-one l = ∞
      by auto
  qed
  ultimately have A ≠ A' ∧ V = V' ∧ V ≠ {} ∧ finite V
        ⟶ l-one (?l (to-list V p) (to-list V' p)) = ∞
    using length-greater-0-conv map-is-Nil-conv zip-eq-Nil-iff
    by metis
  hence A ≠ A' ∧ V = V' ∧ V ≠ {} ∧ finite V ⟶
          votewise-distance swap l-one (A, V, p) (A', V', p') = ∞
    by force
  moreover have
    V ≠ V'
      ⟶ votewise-distance swap l-one (A, V, p) (A', V', p') = ∞
    by simp
  moreover have
    A ≠ A' ∧ V = {}
      ⟶ votewise-distance swap l-one (A, V, p) (A', V', p') = ∞
    by simp
  moreover have
    (A ≠ A' ∧ V = V' ∧ V ≠ {} ∧ finite V)
        ∨ infinite V ∨ (A ≠ A' ∧ V = {}) ∨ V ≠ V'
    using assms
    by blast
  ultimately show votewise-distance swap l-one (A, V, p) (A', V', p') = ∞
    by fastforce
qed
```

5.5.3 Equivalence Lemmas

```
type-synonym ('a, 'v) score-type = ('a, 'v) Election Distance ⇒
  ('a, 'v, 'a Result) Consensus-Class ⇒ ('a, 'v) Election ⇒ 'a ⇒ ereal

type-synonym ('a, 'v) dist-rat-type = ('a, 'v) Election Distance ⇒
  ('a, 'v, 'a Result) Consensus-Class ⇒ 'v set ⇒ 'a set ⇒ ('a, 'v) Profile ⇒ 'a set

type-synonym ('a, 'v) dist-rat-std-type = ('a, 'v) Election Distance ⇒
  ('a, 'v, 'a Result) Consensus-Class ⇒ ('a, 'v, 'a Result) Electoral-Module

type-synonym ('a, 'v) dist-type = ('a, 'v) Election Distance ⇒
  ('a, 'v, 'a Result) Consensus-Class ⇒ ('a, 'v, 'a Result) Electoral-Module

lemma equal-score-swap: (score :: ('a, 'v :: linorder) score-type)
        (votewise-distance swap l-one) = score-std (votewise-distance swap l-one)
  using votewise-non-voters-irrelevant swap-standard
        𝒮𝒞ℱ-result.standard-distance-imp-equal-score
  by fast

lemma swap-ℛ-code[code]: swap-ℛ =
        (𝒮𝒞ℱ-result.distance-ℛ-std :: ('a, 'v :: linorder) dist-rat-std-type)
            (votewise-distance swap l-one)
unfolding swap-ℛ.simps 𝒮𝒞ℱ-result.distance-ℛ.simps 𝒮𝒞ℱ-result.distance-ℛ-std.simps
          𝒮𝒞ℱ-result.ℛ_𝒲.simps 𝒮𝒞ℱ-result.ℛ_𝒲-std.simps equal-score-swap
  by safe

end
```

5.6 Symmetry in Distance-Rationalizable Rules

```
theory Distance-Rationalization-Symmetry
  imports Distance-Rationalization
begin
```

5.6.1 Minimizer Function

```
fun distance-infimum :: 'a Distance ⇒ 'a set ⇒ 'a ⇒ ereal where
```

```
  distance-infimum d A a = Inf (d a ‘ A)

fun closest-preimg-distance :: ('a ⇒ 'b) ⇒ 'a set ⇒ 'a Distance ⇒
      'a ⇒ 'b ⇒ ereal where
  closest-preimg-distance f domain_f d a b = distance-infimum d (preimg f domain_f
b) a

fun minimizer :: ('a ⇒ 'b) ⇒ 'a set ⇒ 'a Distance ⇒ 'b set ⇒ 'a ⇒ 'b set where
  minimizer f domain_f d A a = arg-min-set (closest-preimg-distance f domain_f d
a) A
```

Auxiliary Lemmas

```
lemma rewrite-arg-min-set:
  fixes
    f :: 'a ⇒ 'b :: linorder and
    A :: 'a set
  shows arg-min-set f A = ⋃ (preimg f A ‘ {y ∈ f ‘ A. ∀ z ∈ f ‘ A. y ≤ z})
proof (safe)
  fix x :: 'a
  assume x ∈ arg-min-set f A
  thus x ∈ ⋃ (preimg f A ‘ {y ∈ f ‘ A. ∀ z ∈ f ‘ A. y ≤ z})
    by (simp add: is-arg-min-linorder)
next
  fix x y :: 'a
  assume
    x ∈ preimg f A (f y) and
    ∀ z ∈ f ‘ A. f y ≤ z
  thus x ∈ arg-min-set f A
    by (simp add: is-arg-min-linorder)
qed
```

Equivariance

```
abbreviation Restrp :: 'a rel ⇒ 'a set ⇒ 'a rel where
  Restrp r A ≡ r Int (A × UNIV)

lemma restr-induced-rel:
  fixes
    A :: 'a set and
    B B' :: 'b set and
    φ :: ('a, 'b) binary-fun
  assumes B' ⊆ B
  shows Restrp (action-induced-rel A B φ) B' = action-induced-rel A B' φ
  using assms
  by force

theorem group-action-invar-dist-and-equivar-f-imp-equivar-minimizer:
  fixes
    f :: 'a ⇒ 'b and
    domain_f X :: 'a set and
    d :: 'a Distance and
    well-formed-img :: 'a ⇒ 'b set and
    G :: 'c monoid and
    φ :: ('c, 'a) binary-fun and
    ψ :: ('c, 'b) binary-fun
  defines equivar-prop-set-valued ≡
      action-induced-equivariance (carrier G) X φ (set-action ψ)
  assumes
    action-φ: group-action G X φ and
    group-action-res: group-action G UNIV ψ and
    dom-in-X: domain_f ⊆ X and
    closed-domain:
      closed-restricted-rel (action-induced-rel (carrier G) X φ) X domain_f and
    equivar-img: is-symmetry well-formed-img equivar-prop-set-valued and
    invar-d: invariance_𝒟 d (carrier G) X φ and
    equivar-f:
      is-symmetry f (action-induced-equivariance (carrier G) domain_f φ ψ)
 shows is-symmetry (λ x. minimizer f domain_f d (well-formed-img x) x) equivar-prop-set-valued
proof (unfold action-induced-equivariance-def equivar-prop-set-valued-def is-symmetry.simps
           set-action.simps minimizer.simps, clarify)
  fix
    x :: 'a and
    g :: 'c
  assume
    group-elem: g ∈ carrier G and
    x-in-X: x ∈ X
  hence img-X: φ g x ∈ X
    using action-φ group-action.element-image
    by metis
  let ?x' = φ g x
  let ?c = closest-preimg-distance f domain_f d x and
      ?c' = closest-preimg-distance f domain_f d ?x'
  have ∀ y. preimg f domain_f y ⊆ X
    using dom-in-X
    by fastforce
  hence ∀ y. d x ` (preimg f domain_f y) = d ?x' ` (φ g ` (preimg f domain_f y))
    using x-in-X group-elem invar-dist-image invar-d action-φ
    by metis
  hence ∀ y. Inf (d ?x' ` preimg f domain_f (ψ g y)) =
      Inf (d x ` preimg f domain_f y)
    using assms group-action-equivar-f-imp-equivar-preimg[of G] group-elem
    by metis
  hence comp:
    closest-preimg-distance f domain_f d x =
          (closest-preimg-distance f domain_f d ?x') ∘ (ψ g)
    by auto
  hence ∀ Y A. {preimg ?c' (ψ g ` Y) α | α. α ∈ A} =
      {ψ g ` preimg ?c Y α | α. α ∈ A}
```

using *preimg-comp*

by *auto*

moreover have

$\forall\ Y\, A.\ \{\psi\ g\ \text{`}\ \textit{preimg}\ ?c\ Y\ \alpha \mid \alpha.\ \alpha \in A\} = \{\psi\ g\ \text{`}\ \beta \mid \beta.\ \beta \in \textit{preimg}\ ?c\ Y\ \text{`}\ A\}$

by *blast*

moreover have

$\forall\ Y\, A.\ \textit{preimg}\ ?c'\ (\psi\ g\ \text{`}\ Y)\ \text{`}\ A = \{\textit{preimg}\ ?c'\ (\psi\ g\ \text{`}\ Y)\ \alpha \mid \alpha.\ \alpha \in A\}$

by *blast*

ultimately have

$\forall\ Y\, A.\ \bigcup\ (\textit{preimg}\ ?c'\ (\psi\ g\ \text{`}\ Y)\ \text{`}\ A) = \bigcup\ \{\psi\ g\ \text{`}\ \alpha \mid \alpha.\ \alpha \in \textit{preimg}\ ?c\ Y\ \text{`}\ A\}$

by *simp*

moreover have

$\forall\ Y\, A.\ \bigcup\ \{\psi\ g\ \text{`}\ \alpha \mid \alpha.\ \alpha \in \textit{preimg}\ ?c\ Y\ \text{`}\ A\} = \psi\ g\ \text{`} \bigcup\ (\textit{preimg}\ ?c\ Y\ \text{`}\ A)$

by *blast*

ultimately have $\forall\ Y.$ *arg-min-set* (*closest-preimg-distance f* $\textit{domain}_f$ *d ?x'*) (ψ g ` Y) =

(ψ g) ` (*arg-min-set* (*closest-preimg-distance f* $\textit{domain}_f$ *d x*) Y)

using *rewrite-arg-min-set*[*of ?c'*] *rewrite-arg-min-set*[*of ?c*] *comp*

by *auto*

moreover have *well-formed-img* (φ g x) = ψ g ` *well-formed-img x*

using *equivar-img x-in-X group-elem img-X rewrite-equivariance*

unfolding *equivar-prop-set-valued-def set-action.simps*

by *metis*

ultimately show

arg-min-set (*closest-preimg-distance f* $\textit{domain}_f$ *d* (φ *g x*))

(*well-formed-img* (φ *g x*)) =

ψ *g* ` *arg-min-set* (*closest-preimg-distance f* $\textit{domain}_f$ *d x*)

(*well-formed-img x*)

by *presburger*

qed

Invariance

lemma *closest-dist-invar-under-refl-rel-and-tot-invar-dist*:

fixes

$f :: 'a \Rightarrow 'b$ **and**

$\textit{domain}_f :: 'a\ \textit{set}$ **and**

$d :: 'a\ \textit{Distance}$ **and**

$\textit{rel} :: 'a\ \textit{rel}$

assumes

reflp-on' $\textit{domain}_f$ (*Restrp rel* $\textit{domain}_f$) **and**

$\textit{total-invariance}_{\mathcal{D}}$ *d rel*

shows *is-symmetry* (*closest-preimg-distance f* $\textit{domain}_f$ *d*) (*Invariance rel*)

proof (*unfold is-symmetry.simps*, *intro allI impI ext*)

fix

$a\ b :: 'a$ **and**

$y :: 'b$

assume $(a,\ b) \in \textit{rel}$

hence $\forall\ c \in \textit{domain}_f.\ d\ a\ c = d\ b\ c$

```
    using assms
    unfolding reflp-on'-def reflp-on-def rewrite-total-invariance_𝒟
    by blast
  thus closest-preimg-distance f domain_f d a y =
        closest-preimg-distance f domain_f d b y
    by simp
qed

lemma refl-rel-and-tot-invar-dist-imp-invar-minimizer:
 fixes
    f :: 'a ⇒ 'b and
    domain_f :: 'a set and
    d :: 'a Distance and
    rel :: 'a rel and
    img :: 'b set
  assumes
    reflp-on' domain_f (Restrp rel domain_f) and
    total-invariance_𝒟 d rel
  shows is-symmetry (minimizer f domain_f d img) (Invariance rel)
proof −
  have is-symmetry (closest-preimg-distance f domain_f d) (Invariance rel)
    using assms closest-dist-invar-under-refl-rel-and-tot-invar-dist
    by metis
  thus ?thesis
    by simp
qed

theorem group-act-invar-dist-and-invar-f-imp-invar-minimizer:
  fixes
    f :: 'a ⇒ 'b and
    domain_f A :: 'a set and
    d :: 'a Distance and
    img :: 'b set and
    G :: 'c monoid and
    φ :: ('c, 'a) binary-fun
  defines
    rel ≡ action-induced-rel (carrier G) A φ and
    rel' ≡ action-induced-rel (carrier G) domain_f φ
  assumes
    action-φ: group-action G A φ and
    dom-in-A: domain_f ⊆ A and
    closed-domain: closed-restricted-rel rel A domain_f and
    invar-d: invariance_𝒟 d (carrier G) A φ and
    invar-f: is-symmetry f (Invariance rel')
  shows is-symmetry (minimizer f domain_f d img) (Invariance rel)
proof −
  let
    ?ψ = λ g. id and
    ?img = λ x. img
  have is-symmetry f (action-induced-equivariance (carrier G) domain_f φ ?ψ)
    using invar-f rewrite-invar-as-equivar
    unfolding rel′-def
    by blast
  moreover have group-action G UNIV ?ψ
    using const-id-is-group-action action-φ
    unfolding group-action-def group-hom-def
    by blast
  moreover have
   is-symmetry ?img (action-induced-equivariance (carrier G) A φ (set-action ?ψ))
    unfolding action-induced-equivariance-def
    by fastforce
  ultimately have
    is-symmetry (λ x. minimizer f domain_f d (?img x) x)
            (action-induced-equivariance (carrier G) A φ (set-action ?ψ))
    using group-action-invar-dist-and-equivar-f-imp-equivar-minimizer[of
          - - - - - ?img] assms
    by blast
  thus ?thesis
    unfolding rel-def set-action.simps
    using rewrite-invar-as-equivar image-id
    by metis
qed
```

5.6.2 Minimizer Translation

```
lemma 𝒦_ℰ-is-preimg:
  fixes
    d :: ('a, 'v) Election Distance and
    C :: ('a, 'v, 'r Result) Consensus-Class and
    E :: ('a, 'v) Election and
    w :: 'r
  shows preimg (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) {w} = 𝒦_ℰ C w
proof −
  have preimg (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) {w} =
    {E ∈ elections-𝒦 C.
          elect (rule-𝒦 C) (voters-ℰ E) (alternatives-ℰ E) (profile-ℰ E) = {w}}
    by simp
  also have ... =
      elections-𝒦 C
       ∩ {E. elect (rule-𝒦 C) (voters-ℰ E) (alternatives-ℰ E) (profile-ℰ E) = {w}}
    by blast
  finally show preimg (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) {w} = 𝒦_ℰ C w
    by force
qed

lemma score-is-closest-preimg-dist:
  fixes
    d :: ('a, 'v) Election Distance and
    C :: ('a, 'v, 'r Result) Consensus-Class and
    E :: ('a, 'v) Election and
    w :: 'r
  shows score d C E w =
      closest-preimg-distance (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) d E {w}
proof –
  have score d C E w = Inf (d E ‘ (𝒦_ℰ C w))
    by simp
  moreover have 𝒦_ℰ C w = preimg (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C)
{w}
    using 𝒦_ℰ-is-preimg
    by metis
  moreover have
    Inf (d E ‘ (preimg (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) {w})) =
       closest-preimg-distance (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) d E {w}
    by simp
  ultimately show ?thesis
    by simp
qed

lemma (in result) ℛ_𝒲-is-minimizer:
  fixes
    d :: ('a, 'v) Election Distance and
    C :: ('a, 'v, 'r Result) Consensus-Class
  shows fun_ℰ (ℛ_𝒲 d C) =
    (λ E. ⋃ (minimizer (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) d
                    (singleton-set-system (limit (alternatives-ℰ E) UNIV)) E))
proof
  fix E :: ('a, 'v) Election
  let ?min = (minimizer (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) d
                       (singleton-set-system (limit (alternatives-ℰ E) UNIV)) E)
  have ?min =
    arg-min-set
      (closest-preimg-distance (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) d E)
        (singleton-set-system (limit (alternatives-ℰ E) UNIV))
    by simp
  also have
    ... = singleton-set-system
          (arg-min-set (score d C E) (limit (alternatives-ℰ E) UNIV))
  proof (safe)
    fix R :: 'r set
    assume
      min: R ∈ arg-min-set
                (closest-preimg-distance
            (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) d E)
                (singleton-set-system (limit (alternatives-ℰ E) UNIV))
    hence R ∈ singleton-set-system (limit (alternatives-ℰ E) UNIV)
      using arg-min-subset subsetD
      by (metis (no-types, lifting))
    then obtain r :: 'r where
      res-singleton: R = {r} and
      r-in-lim-set: r ∈ limit (alternatives-ℰ E) UNIV
      by auto
    have ∄ R'. R' ∈ singleton-set-system (limit (alternatives-ℰ E) UNIV)
        ∧ closest-preimg-distance
              (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) d E R'
          < closest-preimg-distance
              (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) d E R
      using min arg-min-set.simps is-arg-min-def CollectD
      by (metis (mono-tags, lifting))
    hence ∄ r'. r' ∈ limit (alternatives-ℰ E) UNIV
        ∧ closest-preimg-distance
            (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) d E {r'}
        < closest-preimg-distance
            (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) d E {r}
      using res-singleton
      by auto
    hence r ∈ arg-min-set (score d C E) (limit (alternatives-ℰ E) UNIV)
      using score-is-closest-preimg-dist r-in-lim-set CollectI
          arg-min-set.simps is-arg-min-def
      by metis
    thus R ∈ singleton-set-system
            (arg-min-set (score d C E) (limit (alternatives-ℰ E) UNIV))
      using res-singleton
      by simp
  next
    fix R :: 'r set
    assume
      R ∈ singleton-set-system
            (arg-min-set (score d C E) (limit (alternatives-ℰ E) UNIV))
    then obtain r :: 'r where
      res-singleton: R = {r} and
      r-min-lim-set:
        r ∈ arg-min-set (score d C E) (limit (alternatives-ℰ E) UNIV)
      by auto
    hence ∄ r'. r' ∈ limit (alternatives-ℰ E) UNIV
              ∧ score d C E r' < score d C E r
      using CollectD arg-min-set.simps is-arg-min-def
      by metis
    hence
      ∄ r'. r' ∈ limit (alternatives-ℰ E) UNIV
        ∧ closest-preimg-distance
              (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) d E {r'}
          < closest-preimg-distance
              (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) d E {r}
      using score-is-closest-preimg-dist
      by metis
    moreover have
```

```
      ∀ R′ ∈ singleton-set-system (limit (alternatives-𝓔 E) UNIV).
          ∃ r′ ∈ limit (alternatives-𝓔 E) UNIV. R′ = {r′}
      by auto
    ultimately have
      ∄ R′. R′ ∈ singleton-set-system (limit (alternatives-𝓔 E) UNIV)
        ∧ closest-preimg-distance
            (elect-r ∘ fun𝓔 (rule-𝒦 C)) (elections-𝒦 C) d E R′
          < closest-preimg-distance
            (elect-r ∘ fun𝓔 (rule-𝒦 C)) (elections-𝒦 C) d E R
      using res-singleton
      by auto
    moreover have
      R ∈ singleton-set-system (limit (alternatives-𝓔 E) UNIV)
      using r-min-lim-set res-singleton arg-min-subset
      by fastforce
    ultimately show
      R ∈ arg-min-set
            (closest-preimg-distance
              (elect-r ∘ fun𝓔 (rule-𝒦 C)) (elections-𝒦 C) d E)
            (singleton-set-system (limit (alternatives-𝓔 E) UNIV))
      using arg-min-set.simps is-arg-min-def CollectI
      by (metis (mono-tags, lifting))
  qed
  also have
    (arg-min-set (score d C E) (limit (alternatives-𝓔 E) UNIV)) =
      fun𝓔 (ℛ𝒲 d C) E
    by simp
  finally have ⋃ ?min = ⋃ (singleton-set-system (fun𝓔 (ℛ𝒲 d C) E))
    by presburger
  thus fun𝓔 (ℛ𝒲 d C) E = ⋃ ?min
    using un-left-inv-singleton-set-system
    by auto
qed
```

Invariance

```
theorem (in result) tot-invar-dist-imp-invar-dr-rule:
  fixes
    d :: ('a, 'v) Election Distance and
    C :: ('a, 'v, 'r Result) Consensus-Class and
    rel :: ('a, 'v) Election rel
  assumes
    r-refl: reflp-on′ (elections-𝒦 C) (Restrp rel (elections-𝒦 C)) and
    tot-invar-d: total-invariance𝒟 d rel and
    invar-res:
      is-symmetry (λ E. limit (alternatives-𝓔 E) UNIV) (Invariance rel)
  shows is-symmetry (fun𝓔 (distance-ℛ d C)) (Invariance rel)
proof −
  let ?min =
    λ E. ⋃ ∘ (minimizer (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) d
          (singleton-set-system (limit (alternatives-ℰ E) UNIV)))
  have ∀ E. is-symmetry (?min E) (Invariance rel)
    using r-refl tot-invar-d invar-comp
        refl-rel-and-tot-invar-dist-imp-invar-minimizer
    by blast
  moreover have is-symmetry ?min (Invariance rel)
    using invar-res
    by auto
  ultimately have is-symmetry (λ E. ?min E E) (Invariance rel)
    using invar-parameterized-fun[of ?min]
    by blast
  also have (λ E. ?min E E) = fun_ℰ (ℛ_𝒲 d C)
    using ℛ_𝒲-is-minimizer
    unfolding comp-def fun_ℰ.simps
    by metis
  finally have invar-ℛ_𝒲: is-symmetry (fun_ℰ (ℛ_𝒲 d C)) (Invariance rel)
    by simp
  hence
    is-symmetry (λ E. limit (alternatives-ℰ E) UNIV − fun_ℰ (ℛ_𝒲 d C) E)
          (Invariance rel)
    using invar-res
    by fastforce
  thus is-symmetry (fun_ℰ (distance-ℛ d C)) (Invariance rel)
    using invar-ℛ_𝒲
    by auto
qed

theorem (in result) invar-dist-cons-imp-invar-dr-rule:
  fixes
    d :: ('a, 'v) Election Distance and
    C :: ('a, 'v, 'r Result) Consensus-Class and
    G :: 'b monoid and
    φ :: ('b, ('a, 'v) Election) binary-fun and
    B :: ('a, 'v) Election set
  defines
    rel ≡ action-induced-rel (carrier G) B φ and
    rel' ≡ action-induced-rel (carrier G) (elections-𝒦 C) φ
  assumes
    action-φ: group-action G B φ and
    consensus-C-in-B: elections-𝒦 C ⊆ B and
    closed-domain:
      closed-restricted-rel rel B (elections-𝒦 C) and
    invar-res:
      is-symmetry (λ E. limit (alternatives-ℰ E) UNIV) (Invariance rel) and
    invar-d: invariance_𝒟 d (carrier G) B φ and
    invar-C-winners: is-symmetry (elect-r ∘ fun_ℰ (rule-𝒦 C)) (Invariance rel')
  shows is-symmetry (fun_ℰ (distance-ℛ d C)) (Invariance rel)
proof −
```

```
  let ?min =
    λ E. ⋃ ∘ (minimizer (elect-r ∘ funℰ (rule-𝒦 C)) (elections-𝒦 C) d
                (singleton-set-system (limit (alternatives-ℰ E) UNIV)))
  have ∀ E. is-symmetry (?min E) (Invariance rel)
    using action-φ closed-domain consensus-C-in-B invar-d invar-C-winners
          group-act-invar-dist-and-invar-f-imp-invar-minimizer rel-def
          rel'-def invar-comp
    by (metis (no-types, lifting))
  moreover have is-symmetry ?min (Invariance rel)
    using invar-res
    by auto
  ultimately have is-symmetry (λ E. ?min E E) (Invariance rel)
    using invar-parameterized-fun[of ?min]
    by blast
  also have (λ E. ?min E E) = funℰ (ℛ𝒲 d C)
    using ℛ𝒲-is-minimizer
    unfolding comp-def funℰ.simps
    by metis
  finally have invar-ℛ𝒲:
    is-symmetry (funℰ (ℛ𝒲 d C)) (Invariance rel)
    by simp
  hence is-symmetry (λ E. limit (alternatives-ℰ E) UNIV −
    funℰ (ℛ𝒲 d C) E) (Invariance rel)
    using invar-res
    by fastforce
  thus is-symmetry (funℰ (distance-ℛ d C)) (Invariance rel)
    using invar-ℛ𝒲
    by simp
qed
```

Equivariance

```
theorem (in result) invar-dist-equivar-cons-imp-equivar-dr-rule:
  fixes
    d :: ('a, 'v) Election Distance and
    C :: ('a, 'v, 'r Result) Consensus-Class and
    G :: 'b monoid and
    φ :: ('b, ('a, 'v) Election) binary-fun and
    ψ :: ('b, 'r) binary-fun and
    B :: ('a, 'v) Election set
  defines
    rel ≡ action-induced-rel (carrier G) B φ and
    rel' ≡ action-induced-rel (carrier G) (elections-𝒦 C) φ and
    equivar-prop ≡
      action-induced-equivariance (carrier G) (elections-𝒦 C)
        φ (set-action ψ) and
    equivar-prop-global-set-valued ≡
        action-induced-equivariance (carrier G) B φ (set-action ψ) and
    equivar-prop-global-result-valued ≡
```

```
      action-induced-equivariance (carrier G) B φ (result-action ψ)
  assumes
    action-φ: group-action G B φ and
    group-act-res: group-action G UNIV ψ and
    cons-elect-set: elections-𝒦 C ⊆ B and
    closed-domain: closed-restricted-rel rel B (elections-𝒦 C) and
    equivar-res:
      is-symmetry (λ E. limit (alternatives-ℰ E) UNIV)
          equivar-prop-global-set-valued and
    invar-d: invariance_𝒟 d (carrier G) B φ and
    equivar-C-winners: is-symmetry (elect-r ∘ fun_ℰ (rule-𝒦 C)) equivar-prop
  shows is-symmetry (fun_ℰ (distance-ℛ d C)) equivar-prop-global-result-valued
proof −
  let ?min-E =
    λ E. minimizer (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) d
            (singleton-set-system (limit (alternatives-ℰ E) UNIV)) E
  let ?min =
    λ E. ⋃ ∘ (minimizer (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) d
            (singleton-set-system (limit (alternatives-ℰ E) UNIV)))
  let ?ψ' = set-action (set-action ψ)
  let ?equivar-prop-global-set-valued' =
          action-induced-equivariance (carrier G) B φ ?ψ'
  have ∀ E g. g ∈ carrier G ⟶ E ∈ B ⟶
          singleton-set-system (limit (alternatives-ℰ (φ g E)) UNIV) =
            {{r} | r. r ∈ limit (alternatives-ℰ (φ g E)) UNIV}
    by simp
  moreover have
    ∀ E g. g ∈ carrier G ⟶ E ∈ B ⟶
        limit (alternatives-ℰ (φ g E)) UNIV =
              ψ g ‘ (limit (alternatives-ℰ E) UNIV)
    using equivar-res action-φ group-action.element-image
    unfolding equivar-prop-global-set-valued-def action-induced-equivariance-def
    by fastforce
  ultimately have ∀ E g. g ∈ carrier G ⟶ E ∈ B ⟶
      singleton-set-system (limit (alternatives-ℰ (φ g E)) UNIV) =
        {{r} | r. r ∈ ψ g ‘ (limit (alternatives-ℰ E) UNIV)}
    by simp
  moreover have
    ∀ E g. {{r} | r. r ∈ ψ g ‘ (limit (alternatives-ℰ E) UNIV)} =
          {ψ g ‘ {r} | r. r ∈ limit (alternatives-ℰ E) UNIV}
    by blast
  moreover have
    ∀ E g. {ψ g ‘ {r} | r. r ∈ limit (alternatives-ℰ E) UNIV} =
          ?ψ' g {{r} | r. r ∈ limit (alternatives-ℰ E) UNIV}
    unfolding set-action.simps
    by blast
  ultimately have
    is-symmetry (λ E. singleton-set-system (limit (alternatives-ℰ E) UNIV))
                    ?equivar-prop-global-set-valued'
    using rewrite-equivariance[of
            λ E. singleton-set-system (limit (alternatives-ℰ E) UNIV)
            carrier G B φ ?ψ']
    by force
  moreover have group-action G UNIV (set-action ψ)
    unfolding set-action.simps
    using group-act-induces-set-group-act[of - UNIV] group-act-res
    by simp
  ultimately have is-symmetry ?min-E ?equivar-prop-global-set-valued'
    using action-φ invar-d cons-elect-set closed-domain equivar-C-winners
          group-action-invar-dist-and-equivar-f-imp-equivar-minimizer[of
            G B φ set-action ψ elections-𝒦 C
              λ E. singleton-set-system (limit (alternatives-ℰ E) UNIV)
              d elect-r ∘ fun_ℰ (rule-𝒦 C)]
    unfolding rel'-def rel-def equivar-prop-def
    by metis
  moreover have
    is-symmetry
          ⋃ (action-induced-equivariance
            (carrier G) UNIV ?ψ' (set-action ψ))
    using equivar-union-under-image-action[of - ψ]
    by simp
  ultimately have is-symmetry (⋃ ∘ ?min-E) equivar-prop-global-set-valued
    unfolding equivar-prop-global-set-valued-def
    using equivar-ind-by-action-comp[of - - UNIV]
    by simp
  moreover have (λ E. ?min E E) = ⋃ ∘ ?min-E
    unfolding comp-def
    by simp
  ultimately have
    is-symmetry (λ E. ?min E E) equivar-prop-global-set-valued
    by simp
  moreover have (λ E. ?min E E) = fun_ℰ (ℛ_𝒲 d C)
    using ℛ_𝒲-is-minimizer
    unfolding comp-def fun_ℰ.simps
    by metis
  ultimately have equivar-ℛ_𝒲:
    is-symmetry (fun_ℰ (ℛ_𝒲 d C)) equivar-prop-global-set-valued
    by simp
  moreover have ∀ g ∈ carrier G. bij (ψ g)
    using group-act-res
    unfolding bij-betw-def
    by (simp add: group-action.inj-prop group-action.surj-prop)
  ultimately have
    is-symmetry (λ E. limit (alternatives-ℰ E) UNIV − fun_ℰ (ℛ_𝒲 d C) E)
        equivar-prop-global-set-valued
    using equivar-res equivar-set-minus
    unfolding action-induced-equivariance-def set-action.simps
              equivar-prop-global-set-valued-def
    by blast
  thus is-symmetry (fun_ℰ (distance-ℛ d C)) equivar-prop-global-result-valued
    using equivar-ℛ_𝒲
    unfolding equivar-prop-global-result-valued-def
              equivar-prop-global-set-valued-def
              rewrite-equivariance
    by simp
qed
```

5.6.3 Inference Rules

```
theorem (in result) anon-dist-and-cons-imp-anon-dr:
  fixes
    d :: ('a, 'v) Election Distance and
    C :: ('a, 'v, 'r Result) Consensus-Class
  assumes
    anon-d: distance-anonymity' well-formed-elections d and
    anon-C: consensus-rule-anonymity' (elections-𝒦 C) C and
    closed-C: closed-restricted-rel (anonymity_ℛ well-formed-elections)
                well-formed-elections (elections-𝒦 C)
    shows anonymity' (distance-ℛ d C)
proof –
  have ∀ π. ∀ E ∈ elections-𝒦 C.
      φ-anon (elections-𝒦 C) π E = φ-anon well-formed-elections π E
    using cons-domain-valid extensional-continuation-subset
    unfolding φ-anon.simps
    by metis
  hence action-induced-rel (carrier bijection_𝒱𝒢) (elections-𝒦 C)
          (φ-anon well-formed-elections) =
      action-induced-rel (carrier bijection_𝒱𝒢) (elections-𝒦 C)
        (φ-anon (elections-𝒦 C))
    using coinciding-actions-ind-equal-rel
    by metis
  hence is-symmetry (elect-r ∘ fun_ℰ (rule-𝒦 C))
        (Invariance (action-induced-rel
          (carrier bijection_𝒱𝒢) (elections-𝒦 C) (φ-anon well-formed-elections)))
    using anon-C
    unfolding consensus-rule-anonymity'.simps anonymity_ℛ.simps
    by presburger
  thus ?thesis
   using anon-d closed-C cons-domain-valid anonymous-group-action.group-action-axioms
        anonymity-action-presv-symmetry invar-dist-cons-imp-invar-dr-rule
    unfolding distance-anonymity'.simps anonymity_ℛ.simps anonymity'.simps
              anonymity-in.simps
    by blast
qed

theorem (in result-properties) neutr-dist-and-cons-imp-neutr-dr:
  fixes
```

```
    d :: ('a, 'v) Election Distance and
    C :: ('a, 'v, 'b Result) Consensus-Class
  assumes
    neutral-d: distance-neutrality well-formed-elections d and
    neutral-C: consensus-rule-neutrality (elections-𝒦 C) C and
    closed-C: closed-restricted-rel (neutrality_ℛ well-formed-elections)
                well-formed-elections (elections-𝒦 C)
  shows neutrality (distance-ℛ d C)
proof −
  have ∀ π. ∀ E ∈ elections-𝒦 C.
      φ-neutral well-formed-elections π E = φ-neutral (elections-𝒦 C) π E
    using cons-domain-valid extensional-continuation-subset
    unfolding φ-neutral.simps
    by metis
  hence is-symmetry (elect-r ∘ fun_ℰ (rule-𝒦 C))
        (action-induced-equivariance (carrier bijection_𝒜𝒢) (elections-𝒦 C)
          (φ-neutral well-formed-elections) (set-action ψ))
    using neutral-C equivar-ind-by-act-coincide
    unfolding consensus-rule-neutrality.simps
    by (metis (no-types, lifting))
  thus ?thesis
    using neutral-d closed-C φ-neutral-action.group-action-axioms
        neutrality action-neutral cons-domain-valid[of C]
        invar-dist-equivar-cons-imp-equivar-dr-rule[of
          - - φ-neutral well-formed-elections]
    by simp
qed

theorem reversal-sym-dist-and-cons-imp-reversal-sym-dr:
  fixes
    d :: ('a, 'c) Election Distance and
    C :: ('a, 'c, 'a rel Result) Consensus-Class
  assumes
    reverse-sym-d: distance-reversal-symmetry well-formed-elections d and
    reverse-sym-C: consensus-rule-reversal-symmetry (elections-𝒦 C) C and
    closed-C: closed-restricted-rel (reversal_ℛ well-formed-elections)
                well-formed-elections (elections-𝒦 C)
  shows reversal-symmetry (𝒮𝒲ℱ-result.distance-ℛ d C)
proof −
  have ∀ π. ∀ E ∈ elections-𝒦 C.
      φ-reverse well-formed-elections π E = φ-reverse (elections-𝒦 C) π E
    using cons-domain-valid extensional-continuation-subset
    unfolding φ-reverse.simps
    by metis
  hence is-symmetry (elect-r ∘ fun_ℰ (rule-𝒦 C))
        (action-induced-equivariance (carrier reversal_𝒢) (elections-𝒦 C)
          (φ-reverse well-formed-elections) (set-action ψ-reverse))
    using reverse-sym-C equivar-ind-by-act-coincide
    unfolding consensus-rule-reversal-symmetry.simps
    by (metis (no-types, lifting))
  thus ?thesis
    using reverse-sym-d closed-C reversal-symmetry-action-presv-symmetry
          SWℱ-result.invar-dist-equivar-cons-imp-equivar-dr-rule
          φ-reverse-action.group-action-axioms cons-domain-valid
          ψ-reverse-action.group-action-axioms
    unfolding reversal-symmetry.simps reversal-symmetry-in-def
              reversal_ℛ.simps distance-reversal-symmetry.simps
    by metis
qed

theorem (in result) tot-hom-dist-imp-hom-dr:
  fixes
    d :: ('a, nat) Election Distance and
    C :: ('a, nat, 'r Result) Consensus-Class
  assumes distance-homogeneity finite-elections-𝒱 d
  shows homogeneity (distance-ℛ d C)
proof −
  have Restrp (homogeneity_ℛ finite-elections-𝒱) (elections-𝒦 C) =
        homogeneity_ℛ (elections-𝒦 C)
    using cons-domain-finite
    unfolding homogeneity_ℛ.simps finite-elections-𝒱-def
    by blast
  hence reflp-on' (elections-𝒦 C)
      (Restrp (homogeneity_ℛ finite-elections-𝒱) (elections-𝒦 C))
    using refl-homogeneity_ℛ[of elections-𝒦 C] cons-domain-finite[of C]
    by presburger
  moreover have
    is-symmetry (λ E. limit (alternatives-ℰ E) UNIV)
      (Invariance (homogeneity_ℛ finite-elections-𝒱))
    using homogeneity-action-presv-symmetry
    by simp
  ultimately show ?thesis
    using assms tot-invar-dist-imp-invar-dr-rule
    unfolding distance-homogeneity-def homogeneity.simps homogeneity-in.simps
    by blast
qed

theorem (in result) tot-hom-dist-imp-hom-dr':
  fixes
    d :: ('a, 'v :: linorder) Election Distance and
    C :: ('a, 'v, 'r Result) Consensus-Class
  assumes distance-homogeneity' finite-elections-𝒱 d
  shows homogeneity' (distance-ℛ d C)
proof (unfold homogeneity'.simps homogeneity'-in.simps)
  have Restrp (homogeneity_ℛ' finite-elections-𝒱) (elections-𝒦 C) =
        homogeneity_ℛ' (elections-𝒦 C)
    using cons-domain-finite
    unfolding homogeneity_ℛ'.simps finite-elections-𝒱-def
    by blast
  hence reflp-on′ (elections-𝒦 C)
      (Restrp (homogeneity𝓡′ finite-elections-𝒱) (elections-𝒦 C))
    using refl-homogeneity𝓡′[of elections-𝒦 C] cons-domain-finite[of C]
    by presburger
  moreover have
    is-symmetry (λ E. limit (alternatives-ℰ E) UNIV)
        (Invariance (homogeneity𝓡′ finite-elections-𝒱))
    using homogeneity′-action-presv-symmetry
    by simp
  ultimately show
   is-symmetry (fun𝓔 (distance-𝓡 d C)) (Invariance (homogeneity𝓡′ finite-elections-𝒱))
    using assms tot-invar-dist-imp-invar-dr-rule
    unfolding distance-homogeneity′-def
    by blast
qed

end
```

5.7 Distance Rationalization on Election Quotients

```
theory Quotient-Distance-Rationalization
  imports Quotient-Module
          Distance-Rationalization-Symmetry
begin
```

5.7.1 Distances

```
fun distance𝒬 :: ′x Distance ⇒ ′x set Distance where
  distance𝒬 d A B = (if (A = {} ∧ B = {}) then 0 else
                  (if (A = {} ∨ B = {}) then ∞ else
                    π𝒬 (tup d) (A × B)))

fun relation-paths :: ′x rel ⇒ ′x list set where
  relation-paths r =
      {p. ∃ k. (length p = 2 ∗ k ∧ (∀ i < k. (p!(2 ∗ i), p!(2 ∗ i + 1)) ∈ r))}

fun admissible-paths :: ′x rel ⇒ ′x set ⇒ ′x set ⇒ ′x list set where
  admissible-paths r X Y =
      {x#p@[y] | x y p. x ∈ X ∧ y ∈ Y ∧ p ∈ relation-paths r}

fun path-length :: ′x list ⇒ ′x Distance ⇒ ereal where
  path-length [] d = 0 |
  path-length [x] d = 0 |
  path-length (x#y#xs) d = d x y + path-length xs d

fun quotient-dist :: ′x rel ⇒ ′x Distance ⇒ ′x set Distance where
```

quotient-dist r d A B =
Inf (⋃ {{*path-length p d* | *p*. *p* ∈ *admissible-paths r A B*}})

fun *distance-infimum*$_{\mathcal{Q}}$ :: *'x Distance* ⇒ *'x set Distance* **where**
distance-infimum$_{\mathcal{Q}}$ *d A B* = *Inf* {*d a b* | *a b*. *a* ∈ *A* ∧ *b* ∈ *B*}

fun *simple* :: *'x rel* ⇒ *'x set* ⇒ *'x Distance* ⇒ *bool* **where**
simple r X d =
(∀ *A* ∈ *X* // *r*.
(∃ *a* ∈ *A*. ∀ *B* ∈ *X* // *r*.
distance-infimum$_{\mathcal{Q}}$ *d A B* = *Inf* {*d a b* | *b*. *b* ∈ *B*}))
— We call a distance simple with respect to a relation if for all relation classes, there is an a in A that minimizes the infimum distance between A and all B such that the infimum distance between these sets coincides with the infimum distance over all b in B for a fixed a.

fun *product'* :: *'x rel* ⇒ (*'x* ∗ *'x*) *rel* **where**
product' r = {(p_1, p_2). ((*fst* p_1, *fst* p_2) ∈ *r* ∧ *snd* p_1 = *snd* p_2)
∨ ((*snd* p_1, *snd* p_2) ∈ *r* ∧ *fst* p_1 = *fst* p_2)}

Auxiliary Lemmas

lemma *tot-dist-invariance-is-congruence*:
fixes
d :: *'x Distance* **and**
r :: *'x rel*
shows (*total-invariance*$_{\mathcal{D}}$ *d r*) = (*tup d respects* (*product r*))
unfolding *total-invariance*$_{\mathcal{D}}$*.simps is-symmetry.simps congruent-def*
by *blast*

lemma *product-helper*:
fixes
r :: *'x rel* **and**
X :: *'x set*
shows
trans-imp: *Relation.trans r* ⟶ *Relation.trans* (*product r*) **and**
refl-imp: *refl-on X r* ⟶ *refl-on* (*X* × *X*) (*product r*) **and**
sym: *sym-on X r* ⟶ *sym-on* (*X* × *X*) (*product r*)
unfolding *Relation.trans-def refl-on-def sym-on-def product.simps*
by *auto*

theorem *dist-pass-to-quotient*:
fixes
d :: *'x Distance* **and**
r :: *'x rel* **and**
X :: *'x set*
assumes
equiv-X-r: *equiv X r* **and**
tot-inv-dist-d-r: *total-invariance*$_{\mathcal{D}}$ *d r*

```
  shows ∀ A B. A ∈ X // r ∧ B ∈ X // r
          ⟶ (∀ a b. a ∈ A ∧ b ∈ B ⟶ distance_Q d A B = d a b)
proof (safe)
  fix
    A B :: 'x set and
    a b :: 'x
  assume
    a-in-A: a ∈ A and
    A ∈ X // r
  moreover with equiv-X-r quotient-eq-iff
  have (a, a) ∈ r
    by metis
  moreover with equiv-X-r
  have a-in-X: a ∈ X
    using equiv-class-eq-iff
    by metis
  ultimately have A-eq-r-a: A = r `` {a}
    using equiv-X-r quotient-eq-iff quotientI
    by fast
  assume
    b-in-B: b ∈ B and
    B ∈ X // r
  moreover with equiv-X-r quotient-eq-iff
  have (b, b) ∈ r
    by metis
  moreover with equiv-X-r
  have b-in-X: b ∈ X
    using equiv-class-eq-iff
    by metis
  ultimately have B-eq-r-b: B = r `` {b}
    using equiv-X-r quotient-eq-iff quotientI
    by fast
  from A-eq-r-a B-eq-r-b a-in-X b-in-X
  have A × B ∈ (X × X) // (product r)
    unfolding quotient-def
    by fastforce
  moreover have equiv (X × X) (product r)
    using equiv-X-r product-helper UNIV-Times-UNIV equivE equivI
    by metis
  moreover have tup d respects (product r)
    using tot-inv-dist-d-r tot-dist-invariance-is-congruence
    by metis
  ultimately show distance_Q d A B = d a b
    unfolding distance_Q.simps
    using pass-to-quotient a-in-A b-in-B
    by fastforce
qed

lemma relation-paths-subset:
  fixes
    n :: nat and
    p :: 'x list and
    r :: 'x rel and
    X :: 'x set
  assumes r ⊆ X × X
  shows ∀ p. p ∈ relation-paths r ⟶ (∀ i < length p. p!i ∈ X)
proof (safe)
  fix
    p :: 'x list and
    i :: nat
  assume p ∈ relation-paths r
  then obtain k :: nat where
    len-p: length p = 2 ∗ k and
    rel: ∀ i < k. (p!(2 ∗ i), p!(2 ∗ i + 1)) ∈ r
    by auto
  moreover obtain k' :: nat where
    i-cases: i = 2 ∗ k' ∨ i = 2 ∗ k' + 1
    using diff-Suc-1 even-Suc oddE odd-two-times-div-two-nat
    by metis
  moreover assume i < length p
  ultimately have k' < k
    by linarith
  thus p!i ∈ X
    using assms rel i-cases
    by blast
qed

lemma admissible-path-len:
  fixes
    d :: 'x Distance and
    r :: 'x rel and
    X :: 'x set and
    a b :: 'x and
    p :: 'x list
  assumes refl-on X r
  shows triangle-ineq X d ∧ p ∈ relation-paths r ∧ total-invariance_𝒟 d r
        ∧ a ∈ X ∧ b ∈ X ⟶ path-length (a#p@[b]) d ≥ d a b
proof (clarify, induction p d arbitrary: a b rule: path-length.induct)
  case (1 d)
  show d a b ≤ path-length (a#[]@[b]) d
    by simp
next
  case (2 x d)
  thus d a b ≤ path-length (a#[x]@[b]) d
    by simp
next
  case (3 x y xs d)
  assume
```

```
    ineq: triangle-ineq X d and
    a-in-X: a ∈ X and
    b-in-X: b ∈ X and
    rel: x#y#xs ∈ relation-paths r and
    invar: total-invariance_𝒟 d r and
    hyp:
    ⋀ a b. triangle-ineq X d ⟹ xs ∈ relation-paths r
        ⟹ total-invariance_𝒟 d r ⟹ a ∈ X ⟹ b ∈ X
        ⟹ d a b ≤ path-length (a#xs@[b]) d
  then obtain k :: nat where
    len: length (x#y#xs) = 2 ∗ k
    by auto
  moreover have ∀ i < k − 1. (xs!(2 ∗ i), xs!(2 ∗ i + 1)) =
    ((x#y#xs)!(2 ∗ (i + 1)), (x#y#xs)!(2 ∗ (i + 1) + 1))
    by simp
  ultimately have ∀ i < k − 1. (xs!(2 ∗ i), xs!(2 ∗ i + 1)) ∈ r
    using rel less-diff-conv
    unfolding relation-paths.simps
    by fastforce
  moreover have length xs = 2 ∗ (k − 1)
    using len
    by simp
  ultimately have xs ∈ relation-paths r
    by simp
  hence ∀ x y. x ∈ X ∧ y ∈ X ⟶ d x y ≤ path-length (x#xs@[y]) d
    using ineq invar hyp
    by blast
  moreover have
    path-length (a#(x#y#xs)@[b]) d = d a x + path-length (y#xs@[b]) d
    by simp
  moreover have x-rel-y: (x, y) ∈ r
    using rel
    unfolding relation-paths.simps
    by fastforce
  ultimately have path-length (a#(x#y#xs)@[b]) d ≥ d a x + d y b
    using assms add-left-mono assms refl-onD2 b-in-X
    unfolding refl-on-def
    by metis
  moreover have d a x + d y b = d a x + d x b
    using invar x-rel-y rewrite-total-invariance_𝒟 assms b-in-X
    unfolding refl-on-def
    by fastforce
  moreover have d a x + d x b ≥ d a b
    using a-in-X b-in-X x-rel-y assms ineq
    unfolding refl-on-def triangle-ineq-def
    by auto
  ultimately show d a b ≤ path-length (a#(x#y#xs)@[b]) d
    by simp
qed
```

```
lemma quotient-dist-coincides-with-dist_𝒬:
  fixes
    d :: 'x Distance and
    r :: 'x rel and
    X :: 'x set
  assumes
    equiv: equiv X r and
    tri: triangle-ineq X d and
    invar: total-invariance_𝒟 d r
  shows ∀ A ∈ X // r. ∀ B ∈ X // r. quotient-dist r d A B = distance_𝒬 d A B
proof (clarify)
  fix A B :: 'x set
  assume
    A-in-quot-X: A ∈ X // r and
    B-in-quot-X: B ∈ X // r
  then obtain
    a b :: 'x where
      el: a ∈ A ∧ b ∈ B and
      def-dist: distance_𝒬 d A B = d a b
    using dist-pass-to-quotient assms in-quotient-imp-non-empty ex-in-conv
    by (metis (full-types))
  have b ∈ X
    using B-in-quot-X el equiv quotient-eq-iff equiv equiv-class-eq-iff
    by metis
  hence B = r `` {b}
    using equiv-class-self B-in-quot-X el equiv quotientI quotient-eq-iff
    by metis
  moreover have a ∈ X
    using A-in-quot-X el equiv quotient-eq-iff equiv equiv-class-eq-iff
    by metis
  ultimately have equiv-class: A = r `` {a} ∧ B = r `` {b}
    using A-in-quot-X el equiv quotientI quotient-eq-iff
    by slow
  have ∀ p ∈ admissible-paths r A B.
        (∃ p' x y. x ∈ A ∧ y ∈ B ∧ p' ∈ relation-paths r ∧ p = x#p'@[y])
    unfolding admissible-paths.simps
    by blast
  moreover have ∀ x y. x ∈ A ∧ y ∈ B ⟶ d x y = d a b
    using invar equiv-class
    by auto
  moreover have refl-on X r
    using equiv
    unfolding equiv-def
    by blast
  moreover have r ⊆ X × X ∧ A ⊆ X ∧ B ⊆ X
    using assms A-in-quot-X B-in-quot-X Union-quotient Union-upper
    unfolding equiv-def refl-on-def
    by metis
  ultimately have ∀ p. p ∈ admissible-paths r A B ⟶ path-length p d ≥ d a b
    using admissible-path-len[of X r d] tri el invar in-mono
    by metis
  hence ∀ l. l ∈ ⋃ {{path-length p d | p. p ∈ admissible-paths r A B}}
                 ⟶ l ≥ d a b
    by blast
  hence geq: quotient-dist r d A B ≥ distance_𝒬 d A B
    unfolding quotient-dist.simps le-Inf-iff
    using def-dist
    by simp
  have [a, b] ∈ admissible-paths r A B
    using el
    by simp
  moreover have path-length [a, b] d = d a b
    by simp
  ultimately have quotient-dist r d A B ≤ d a b
    unfolding quotient-dist.simps
    using CollectI Inf-lower ccpo-Sup-singleton
    by (metis (mono-tags, lifting))
  thus quotient-dist r d A B = distance_𝒬 d A B
    using geq def-dist nle-le
    by metis
qed

lemma inf-dist-coincides-with-dist_𝒬:
  fixes
    d :: 'x Distance and
    r :: 'x rel and
    X :: 'x set
  assumes
    equiv-X-r: equiv X r and
    tot-inv-d-r: total-invariance_𝒟 d r
  shows ∀ A ∈ X // r. ∀ B ∈ X // r.
          distance-infimum_𝒬 d A B = distance_𝒬 d A B
proof (clarify)
  fix A B :: 'x set
  assume
    A-in-quot-X: A ∈ X // r and
    B-in-quot-X: B ∈ X // r
  then obtain
    a b :: 'x where
      el: a ∈ A ∧ b ∈ B and
      def-dist: distance_𝒬 d A B = d a b
    using dist-pass-to-quotient equiv-X-r tot-inv-d-r
         in-quotient-imp-non-empty ex-in-conv
    by (metis (full-types))
  from def-dist equiv-X-r tot-inv-d-r
  have ∀ x y. x ∈ A ∧ y ∈ B ⟶ d x y = d a b
    using dist-pass-to-quotient A-in-quot-X B-in-quot-X
    by force
  hence {d x y | x y. x ∈ A ∧ y ∈ B} = {d a b}
    using el
    by blast
  thus distance-infimum_Q d A B = distance_Q d A B
    unfolding distance-infimum_Q.simps
    using def-dist
    by simp
qed

lemma inf-helper:
  fixes
    A B :: 'x set and
    d :: 'x Distance
  shows Inf {d a b | a b. a ∈ A ∧ b ∈ B} =
          Inf {Inf {d a b | b. b ∈ B} | a. a ∈ A}
proof −
  have ∀ a b. a ∈ A ∧ b ∈ B ⟶ Inf {d a b | b. b ∈ B} ≤ d a b
    using INF-lower Setcompr-eq-image
    by metis
  hence ∀ α ∈ {d a b | a b. a ∈ A ∧ b ∈ B}.
          ∃ β ∈ {Inf {d a b | b. b ∈ B} | a. a ∈ A}. β ≤ α
    by blast
  hence Inf {Inf {d a b | b. b ∈ B} | a. a ∈ A}
        ≤ Inf {d a b | a b. a ∈ A ∧ b ∈ B}
    using Inf-mono
    by (metis (no-types, lifting))
  moreover have
    ¬ (Inf {Inf {d a b | b. b ∈ B} | a. a ∈ A}
          < Inf {d a b | a b. a ∈ A ∧ b ∈ B})
  proof (rule ccontr, safe)
    assume Inf {Inf {d a b | b. b ∈ B} | a. a ∈ A}
              < Inf {d a b | a b. a ∈ A ∧ b ∈ B}
    then obtain α :: ereal where
      inf: α ∈ {Inf {d a b | b. b ∈ B} | a. a ∈ A} and
      less: α < Inf {d a b | a b. a ∈ A ∧ b ∈ B}
      using Inf-less-iff
      by (metis (no-types, lifting))
    then obtain a :: 'x where
      a-in-A: a ∈ A and
      α = Inf {d a b | b. b ∈ B}
      by blast
    with less
    have inf-less: Inf {d a b | b. b ∈ B} < Inf {d a b | a b. a ∈ A ∧ b ∈ B}
      by blast
    have {d a b | b. b ∈ B} ⊆ {d a b | a b. a ∈ A ∧ b ∈ B}
      using a-in-A
      by blast
    hence Inf {d a b | a b. a ∈ A ∧ b ∈ B} ≤ Inf {d a b | b. b ∈ B}
```

```
      using Inf-superset-mono
      by (metis (no-types, lifting))
    with inf-less
    show False
      using linorder-not-less
      by simp
  qed
  ultimately show ?thesis
    by simp
qed

lemma invar-dist-simple:
  fixes
    d :: 'y Distance and
    G :: 'x monoid and
    Y :: 'y set and
    φ :: ('x, 'y) binary-fun
  assumes
    action-φ: group-action G Y φ and
    invar: invariance_𝒟 d (carrier G) Y φ
  shows simple (action-induced-rel (carrier G) Y φ) Y d
proof (unfold simple.simps, safe)
  fix A :: 'y set
  assume class_Y: A ∈ Y // action-induced-rel (carrier G) Y φ
  moreover have equiv-rel:
    equiv Y (action-induced-rel (carrier G) Y φ)
    using assms rel-ind-by-group-act-equiv
    by blast
  ultimately obtain a :: 'y where
    a-in-A: a ∈ A
    using equiv-Eps-in
    by blast
  have subset: ∀ B ∈ Y // action-induced-rel (carrier G) Y φ. B ⊆ Y
    using equiv-rel in-quotient-imp-subset
    by blast
  hence ∀ B ∈ Y // action-induced-rel (carrier G) Y φ.
        ∀ B' ∈ Y // action-induced-rel (carrier G) Y φ.
          ∀ b ∈ B. ∀ c ∈ B'. b ∈ Y ∧ c ∈ Y
    using class_Y
    by blast
  hence eq-dist:
    ∀ B ∈ Y // action-induced-rel (carrier G) Y φ.
      ∀ B' ∈ Y // action-induced-rel (carrier G) Y φ.
        ∀ b ∈ B. ∀ c ∈ B'. ∀ g ∈ carrier G.
          d (φ g c) (φ g b) = d c b
    using invar rewrite-invariance_𝒟 class_Y
    by metis
  have ∀ b ∈ Y. ∀ g ∈ carrier G.
        (b, φ g b) ∈ action-induced-rel (carrier G) Y φ
    unfolding action-induced-rel.simps
    using group-action.element-image action-φ
    by fastforce
  hence ∀ b ∈ Y. ∀ g ∈ carrier G.
          φ g b ∈ action-induced-rel (carrier G) Y φ `` {b}
    unfolding Image-def
    by blast
  moreover have equiv-class:
    ∀ B. B ∈ Y // action-induced-rel (carrier G) Y φ ⟶
      (∀ b ∈ B. B = action-induced-rel (carrier G) Y φ `` {b})
    using Image-singleton-iff equiv-class-eq-iff equiv-rel
        quotientI quotient-eq-iff
    by meson
  ultimately have closed-class:
    ∀ B ∈ Y // action-induced-rel (carrier G) Y φ.
        ∀ b ∈ B. ∀ g ∈ carrier G. φ g b ∈ B
    using equiv-rel subset
    by blast
  with eq-dist class_Y
  have a-subset-A:
    ∀ B ∈ Y // action-induced-rel (carrier G) Y φ.
      {d a b | b. b ∈ B} ⊆ {d a b | a b. a ∈ A ∧ b ∈ B}
    using a-in-A
    by blast
  have ∀ a' ∈ A. A = action-induced-rel (carrier G) Y φ `` {a'}
    using class_Y equiv-rel equiv-class
    by presburger
  hence ∀ a' ∈ A. (a', a) ∈ action-induced-rel (carrier G) Y φ
    using a-in-A
    by blast
  hence ∀ a' ∈ A. ∃ g ∈ carrier G. φ g a' = a
    by simp
  hence ∀ B ∈ Y // action-induced-rel (carrier G) Y φ.
      ∀ a' b. a' ∈ A ∧ b ∈ B ⟶ (∃ g ∈ carrier G. d a' b = d a (φ g b))
    using eq-dist class_Y
    by metis
  hence ∀ B ∈ Y // action-induced-rel (carrier G) Y φ.
      ∀ a' b. a' ∈ A ∧ b ∈ B ⟶ d a' b ∈ {d a b | b. b ∈ B}
    using closed-class mem-Collect-eq
    by fastforce
  hence ∀ B ∈ Y // action-induced-rel (carrier G) Y φ.
      {d a b | b. b ∈ B} ⊇ {d a b | a b. a ∈ A ∧ b ∈ B}
    using closed-class
    by blast
  with a-subset-A
  have ∀ B ∈ Y // action-induced-rel (carrier G) Y φ.
        distance-infimum_Q d A B = Inf {d a b | b. b ∈ B}
    unfolding distance-infimum_Q.simps
    by fastforce
  thus ∃ a ∈ A. ∀ B ∈ Y // action-induced-rel (carrier G) Y φ.
      distance-infimum_𝒬 d A B = Inf {d a b | b. b ∈ B}
    using a-in-A
    by blast
qed

lemma tot-invar-dist-simple:
  fixes
    d :: 'x Distance and
    r :: 'x rel and
    X :: 'x set
  assumes
    equiv-on-X: equiv X r and
    invar: total-invariance_𝒟 d r
  shows simple r X d
proof (unfold simple.simps, safe)
  fix A :: 'x set
  assume A-quot-X: A ∈ X // r
  then obtain a :: 'x where
    a-in-A: a ∈ A
    using equiv-on-X equiv-Eps-in
    by blast
  have ∀ a ∈ A. A = r `` {a}
    using A-quot-X Image-singleton-iff equiv-class-eq equiv-on-X quotientE
    by metis
  hence ∀ a a'. a ∈ A ∧ a' ∈ A ⟶ (a, a') ∈ r
    by blast
  moreover have ∀ B ∈ X // r. ∀ b ∈ B. (b, b) ∈ r
    using equiv-on-X quotient-eq-iff
    by metis
  ultimately have
    ∀ B ∈ X // r. ∀ a a' b. a ∈ A ∧ a' ∈ A ∧ b ∈ B ⟶ d a b = d a' b
    using invar rewrite-total-invariance_𝒟
    by simp
  hence ∀ B ∈ X // r.
    {d a b | a b. a ∈ A ∧ b ∈ B} = {d a b | a' b. a' ∈ A ∧ b ∈ B}
    using a-in-A
    by blast
  moreover have
    ∀ B ∈ X // r. {d a b | a' b. a' ∈ A ∧ b ∈ B} =
        {d a b | b. b ∈ B}
    using a-in-A
    by blast
  ultimately have
    ∀ B ∈ X // r. Inf {d a b | a b. a ∈ A ∧ b ∈ B} =
        Inf {d a b | b. b ∈ B}
    by simp
  hence ∀ B ∈ X // r. distance-infimum_𝒬 d A B =
        Inf {d a b | b. b ∈ B}
    by simp
  thus ∃ a ∈ A. ∀ B ∈ X // r.
        distance-infimum_Q d A B = Inf {d a b | b. b ∈ B}
    using a-in-A
    by blast
qed
```

5.7.2 Consensus and Results

```
fun elections-𝒦_Q :: ('a, 'v) Election rel ⇒ ('a, 'v, 'r Result) Consensus-Class ⇒
        ('a, 'v) Election set set where
  elections-𝒦_Q r C = (elections-𝒦 C) // r

fun (in result) limit_Q :: ('a, 'v) Election set ⇒ 'r set ⇒ 'r set where
  limit_Q X res = ⋂ {limit (alternatives-ℰ E) res | E. E ∈ X}
```

Auxiliary Lemmas

```
lemma closed-under-equiv-rel-subset:
   fixes
    X Y Z :: 'x set and
    r :: 'x rel
  assumes
    equiv X r and
    Y ⊆ X and
    Z ⊆ X and
    Z ∈ Y // r and
    closed-restricted-rel r X Y
  shows Z ⊆ Y
proof (safe)
  fix z :: 'x
  assume z ∈ Z
  then obtain y :: 'x where
    y ∈ Y and
    (y, z) ∈ r
    using assms
    unfolding quotient-def Image-def
    by blast
  hence (y, z) ∈ r ∩ Y × X
    using assms
    unfolding equiv-def refl-on-def
    by blast
  hence z ∈ {z. ∃ y ∈ Y. (y, z) ∈ r ∩ Y × X}
    by blast
  thus z ∈ Y
    using assms
    unfolding closed-restricted-rel.simps restricted-rel.simps
    by blast
qed
```

```
lemma (in result) limit-invar:
  fixes
    d :: ('a, 'v) Election Distance and
    r :: ('a, 'v) Election rel and
    C :: ('a, 'v, 'r Result) Consensus-Class and
    X A :: ('a, 'v) Election set
  assumes
    quot-class: A ∈ X // r and
    equiv-rel: equiv X r and
    cons-subset: elections-𝒦 C ⊆ X and
    invar-res: is-symmetry (λ E. limit (alternatives-ℰ E) UNIV) (Invariance r)
  shows ∀ a ∈ A. limit (alternatives-ℰ a) UNIV = limit_𝒬 A UNIV
proof
  fix a :: ('a, 'v) Election
  assume a-in-A: a ∈ A
  hence ∀ b ∈ A. (a, b) ∈ r
    using quot-class equiv-rel quotient-eq-iff
    by metis
  hence ∀ b ∈ A.
    limit (alternatives-ℰ b) UNIV = limit (alternatives-ℰ a) UNIV
    using invar-res
    unfolding is-symmetry.simps
    by (metis (mono-tags, lifting))
  hence limit_𝒬 A UNIV = ⋂ {limit (alternatives-ℰ a) UNIV}
    unfolding limit_𝒬.simps
    using a-in-A
    by blast
  thus limit (alternatives-ℰ a) UNIV = limit_𝒬 A UNIV
    by simp
qed

lemma (in result) preimg-invar:
  fixes
    f :: 'x ⇒ 'y and
    domain_f X :: 'x set and
    d :: 'x Distance and
    r :: 'x rel
  assumes
    equiv-rel: equiv X r and
    cons-subset: domain_f ⊆ X and
    closed-domain: closed-restricted-rel r X domain_f and
    invar-f: is-symmetry f (Invariance (Restr r domain_f))
  shows ∀ y. (preimg f domain_f y) // r = preimg (π_𝒬 f) (domain_f // r) y
proof (safe)
  fix
    A :: 'x set and
    y :: 'y
  assume preimg-quot: A ∈ preimg f domain_f y // r
  hence A-in-dom: A ∈ domain_f // r
```

```
    unfolding preimg.simps quotient-def
    by blast
  obtain x :: 'x where
    x ∈ preimg f domain_f y and
    A-eq-img-singleton-r: A = r `` {x}
    using equiv-rel preimg-quot quotientE
    unfolding quotient-def
    by blast
  hence x-in-dom-and-f-x-y: x ∈ domain_f ∧ f x = y
    unfolding preimg.simps
    by blast
  moreover have r `` {x} ⊆ X
    using equiv-rel equiv-type
    by fastforce
  ultimately have r `` {x} ⊆ domain_f
    using closed-domain A-eq-img-singleton-r A-in-dom
    by fastforce
  hence ∀ x' ∈ r `` {x}. (x, x') ∈ Restr r domain_f
    using x-in-dom-and-f-x-y in-mono
    by blast
  hence ∀ x' ∈ r `` {x}. f x' = y
    using invar-f x-in-dom-and-f-x-y
    unfolding is-symmetry.simps
    by metis
  moreover have x ∈ A
    using equiv-rel cons-subset equiv-class-self in-mono
          A-eq-img-singleton-r x-in-dom-and-f-x-y
    by metis
  ultimately have f ` A = {y}
    using A-eq-img-singleton-r
    by auto
  hence π_Q f A = y
    unfolding π_Q.simps singleton-set.simps
    using insert-absorb insert-iff insert-not-empty singleton-set-def-if-card-one
          is-singletonI is-singleton-altdef singleton-set.simps
    by metis
  thus A ∈ preimg (π_Q f) (domain_f // r) y
    using A-in-dom
    unfolding preimg.simps
    by blast
next
  fix
    A :: 'x set and
    y :: 'y
  assume quot-preimg: A ∈ preimg (π_Q f) (domain_f // r) y
  hence A-in-dom-rel-r: A ∈ domain_f // r
    using cons-subset equiv-rel
    by auto
  hence A ⊆ X
```

```
    using equiv-rel cons-subset Image-subset equiv-type quotientE
    by metis
  hence A-in-dom: A ⊆ domain_f
    using closed-under-equiv-rel-subset
          closed-domain cons-subset A-in-dom-rel-r equiv-rel
    by blast
  moreover obtain x :: 'x where
    x-in-A: x ∈ A and
    A-eq-r-img-single-x: A = r `` {x}
    using A-in-dom-rel-r equiv-rel cons-subset equiv-class-self in-mono quotientE
    by metis
  ultimately have ∀ x' ∈ A. (x, x') ∈ Restr r domain_f
    by blast
  hence ∀ x' ∈ A. f x' = f x
    using invar-f
    by fastforce
  hence f ` A = {f x}
    using x-in-A
    by blast
  hence π_Q f A = f x
    unfolding π_Q.simps singleton-set.simps
    using is-singleton-altdef singleton-set-def-if-card-one
    by fastforce
  also have π_Q f A = y
    using quot-preimg
    unfolding preimg.simps
    by blast
  finally have f x = y
    by simp
  moreover have x ∈ domain_f
    using x-in-A A-in-dom
    by blast
  ultimately have x ∈ preimg f domain_f y
    by simp
  thus A ∈ preimg f domain_f y // r
    using A-eq-r-img-single-x
    unfolding quotient-def
    by blast
qed

lemma minimizer-helper:
  fixes
    f :: 'x ⇒ 'y and
    domain_f :: 'x set and
    d :: 'x Distance and
    Y :: 'y set and
    x :: 'x and
    y :: 'y
  shows y ∈ minimizer f domain_f d Y x =
      (y ∈ Y ∧ (∀ y' ∈ Y.
          Inf (d x ` (preimg f domain_f y)) ≤ Inf (d x ` (preimg f domain_f y'))))
  unfolding is-arg-min-def minimizer.simps arg-min-set.simps
  by auto

lemma rewr-singleton-set-system-union:
  fixes
    Y :: 'x set set and
    X :: 'x set
  assumes Y ⊆ singleton-set-system X
  shows
    singleton-set-union: x ∈ ⋃ Y ⟷ {x} ∈ Y and
    obtain-singleton: A ∈ singleton-set-system X ⟷ (∃ x ∈ X. A = {x})
  unfolding singleton-set-system.simps
  using assms
  by auto

lemma union-inf:
  fixes X :: ereal set set
  shows Inf {Inf A | A. A ∈ X} = Inf (⋃ X)
proof –
  let ?inf = Inf {Inf A | A. A ∈ X}
  have ∀ A ∈ X. ∀ x ∈ A. ?inf ≤ x
    using INF-lower2 Inf-lower Setcompr-eq-image
    by metis
  hence ∀ x ∈ ⋃ X. ?inf ≤ x
    by simp
  hence le: ?inf ≤ Inf (⋃ X)
    using Inf-greatest
    by blast
  have ∀ A ∈ X. Inf (⋃ X) ≤ Inf A
    using Inf-superset-mono Union-upper
    by metis
  hence Inf (⋃ X) ≤ Inf {Inf A | A. A ∈ X}
    using le-Inf-iff
    by auto
  thus ?thesis
    using le
    by simp
qed
```

5.7.3 Distance Rationalization

```
fun (in result) ℛ_𝒬 :: ('a, 'v) Election rel ⇒ ('a, 'v) Election Distance ⇒
        ('a, 'v, 'r Result) Consensus-Class ⇒ ('a, 'v) Election set ⇒ 'r set where
  ℛ_𝒬 r d C A =
    ⋃ (minimizer (π_𝒬 (elect-r ∘ fun_ℰ (rule-𝒦 C))) (elections-𝒦_𝒬 r C)
        (distance-infimum_𝒬 d) (singleton-set-system (limit_𝒬 A UNIV)) A)
```

```
fun (in result) distance-ℛ_𝒬 :: ('a, 'v) Election rel ⇒ ('a, 'v) Election Distance ⇒
      ('a, 'v, 'r Result) Consensus-Class ⇒ ('a, 'v) Election set ⇒ 'r Result where
  distance-ℛ_𝒬 r d C A =
    (ℛ_𝒬 r d C A,
     π_𝒬 (λ E. limit (alternatives-ℰ E) UNIV) A − ℛ_𝒬 r d C A,
     {})
```

Proposition 4.17 by Hadjibeyli and Wilson [3].

```
theorem (in result) invar-dr-simple-dist-imp-quotient-dr-winners:
  fixes
    d :: ('a, 'v) Election Distance and
    C :: ('a, 'v, 'r Result) Consensus-Class and
    r :: ('a, 'v) Election rel and
    X A :: ('a, 'v) Election set
  assumes
    simple: simple r X d and
    closed-domain: closed-restricted-rel r X (elections-𝒦 C) and
    invar-res:
      is-symmetry (λ E. limit (alternatives-ℰ E) UNIV) (Invariance r) and
    invar-C: is-symmetry (elect-r ∘ fun_ℰ (rule-𝒦 C))
                  (Invariance (Restr r (elections-𝒦 C))) and
    invar-dr: is-symmetry (fun_ℰ (ℛ_𝒲 d C)) (Invariance r) and
    quot-class: A ∈ X // r and
    equiv-rel: equiv X r and
    cons-subset: elections-𝒦 C ⊆ X
  shows π_𝒬 (fun_ℰ (ℛ_𝒲 d C)) A = ℛ_𝒬 r d C A
proof −
  have preimg-img-imp-cls:
    ∀ y B. B ∈ preimg (π_𝒬 (elect-r ∘ fun_ℰ (rule-𝒦 C))) (elections-𝒦_𝒬 r C) y
        ⟶ B ∈ (elections-𝒦 C) // r
    by simp
  have ∀ y'. ∀ E
        ∈ preimg (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) y'. E ∈ r `` {E}
    using equiv-rel cons-subset equiv-class-self equiv-rel in-mono
    unfolding equiv-def preimg.simps
    by fastforce
  hence ∀ y'.
      ⋃ (preimg (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) y' // r) ⊇
      preimg (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) y'
    unfolding quotient-def
    by blast
  moreover have ∀ y'.
      ⋃ (preimg (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) y' // r) ⊆
      preimg (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) y'
  proof (intro allI subsetI)
    fix
      Y' :: 'r set and
      E :: ('a, 'v) Election
    assume E ∈ ⋃ (preimg (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) Y' // r)
    then obtain B :: ('a, 'v) Election set where
      E-in-B: E ∈ B and
      B ∈ preimg (elect-r ∘ fun_𝓔 (rule-𝒦 C)) (elections-𝒦 C) Y' // r
      by blast
    then obtain E' :: ('a, 'v) Election where
      B = r `` {E'} and
      map-to-Y': E' ∈ preimg (elect-r ∘ fun_𝓔 (rule-𝒦 C)) (elections-𝒦 C) Y'
      using quotientE
      by blast
    hence in-restr-rel: (E', E) ∈ r ∩ (elections-𝒦 C) × X
      using E-in-B equiv-rel
      unfolding preimg.simps equiv-def refl-on-def
      by blast
    hence E ∈ elections-𝒦 C
      using closed-domain
      unfolding closed-restricted-rel.simps restricted-rel.simps Image-def
      by blast
    hence rel-cons-els: (E', E) ∈ Restr r (elections-𝒦 C)
      using in-restr-rel
      by blast
    hence (elect-r ∘ fun_𝓔 (rule-𝒦 C)) E = (elect-r ∘ fun_𝓔 (rule-𝒦 C)) E'
      using invar-C
      unfolding is-symmetry.simps
      by blast
    hence (elect-r ∘ fun_𝓔 (rule-𝒦 C)) E = Y'
      using map-to-Y'
      by simp
    thus E ∈ preimg (elect-r ∘ fun_𝓔 (rule-𝒦 C)) (elections-𝒦 C) Y'
      unfolding preimg.simps
      using rel-cons-els
      by blast
  qed
  ultimately have preimg-partition: ∀ y'.
      ⋃ (preimg (elect-r ∘ fun_𝓔 (rule-𝒦 C)) (elections-𝒦 C) y' // r) =
      preimg (elect-r ∘ fun_𝓔 (rule-𝒦 C)) (elections-𝒦 C) y'
    by blast
  have quot-classes-subset: (elections-𝒦 C) // r ⊆ X // r
    using cons-subset
    unfolding quotient-def
    by blast
  obtain a :: ('a, 'v) Election where
    a-in-A: a ∈ A and
    a-def-inf-dist:
      ∀ B ∈ X // r.
        distance-infimum_𝒬 d A B = Inf {d a b | b. b ∈ B}
    using simple quot-class
    unfolding simple.simps
    by blast
  hence inf-dist-preimg-sets:
    ∀ y' B. B ∈ preimg (π_𝒬 (elect-r ∘ fun_ℰ (rule-𝒦 C))) (elections-𝒦_𝒬 r C) y'
        ⟶ distance-infimum_𝒬 d A B = Inf {d a b | b. b ∈ B}
    using preimg-img-imp-cls quot-classes-subset
    by blast
  have wf-res-eq: singleton-set-system (limit (alternatives-ℰ a) UNIV) =
      singleton-set-system (limit_𝒬 A UNIV)
    using invar-res a-in-A quot-class cons-subset equiv-rel limit-invar
    by metis
  have inf-le-iff: ∀ x.
      (∀ y' ∈ singleton-set-system (limit (alternatives-ℰ a) UNIV).
        Inf (d a ‘ preimg (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) {x})
        ≤ Inf (d a ‘ preimg (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) y'))
      = (∀ y' ∈ singleton-set-system (limit_𝒬 A UNIV).
        Inf (distance-infimum_𝒬 d A ‘ preimg (π_𝒬 (elect-r ∘ fun_ℰ (rule-𝒦 C)))
                (elections-𝒦_𝒬 r C) {x})
        ≤ Inf (distance-infimum_𝒬 d A ‘ preimg (π_𝒬 (elect-r ∘ fun_ℰ (rule-𝒦 C)))
                (elections-𝒦_𝒬 r C) y'))
  proof –
    have preimg-partition-dist: ∀ y'.
        Inf {d a b | b. b ∈
            ⋃ (preimg (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) y' // r)} =
        Inf (d a ‘ preimg (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) y')
      using Setcompr-eq-image preimg-partition
      by metis
    have ∀ y'.
        {Inf {d a b | b. b ∈ B}
          | B. B ∈ preimg (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) y' // r}
      = {Inf E | E. E ∈ {{d a b | b. b ∈ B}
          | B. B ∈ preimg (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) y' // r}}
      by blast
    hence ∀ y'.
        Inf {Inf {d a b | b. b ∈ B} | B.
          B ∈ preimg (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) y' // r} =
        Inf (⋃ {{d a b | b. b ∈ B} | B.
          B ∈ (preimg (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) y' // r)})
      using union-inf
      by presburger
    moreover have
      ∀ y'.
        {d a b | b. b ∈ ⋃
          (preimg (elect-r ∘ fun_ℰ (rule-𝒦 C))
              (elections-𝒦 C) y' // r)} =
            ⋃ {{d a b | b. b ∈ B} | B.
                  B ∈ (preimg (elect-r ∘ fun_ℰ (rule-𝒦 C))
                    (elections-𝒦 C) y' // r)}
      by blast
    ultimately have rewrite-inf-dist:
      ∀ y'. Inf {Inf {d a b | b. b ∈ B}
        | B. B ∈ preimg
```

```
        (elect-r ∘ fun_𝓔 (rule-𝒦 C)) (elections-𝒦 C) y′ // r} =
    Inf {d a b
      | b. b ∈ ⋃ (preimg
          (elect-r ∘ fun_𝓔 (rule-𝒦 C)) (elections-𝒦 C) y′ // r)}
    by presburger
  have ∀ y′. distance-infimum_𝒬 d A ‘ preimg (π_𝒬 (elect-r ∘ fun_𝓔 (rule-𝒦 C)))
            (elections-𝒦_𝒬 r C) y′ =
    {Inf {d a b | b. b ∈ B}
      | B. B ∈ preimg (π_𝒬 (elect-r ∘ fun_𝓔 (rule-𝒦 C))) (elections-𝒦_𝒬 r C) y′}
    using inf-dist-preimg-sets
    unfolding Image-def
    by auto
  moreover have ∀ y′.
      {Inf {d a b | b. b ∈ B} | B.
        B ∈ preimg (π_𝒬 (elect-r ∘ fun_𝓔 (rule-𝒦 C))) (elections-𝒦_𝒬 r C) y′} =
      {Inf {d a b | b. b ∈ B} | B.
        B ∈ (preimg (elect-r ∘ fun_𝓔 (rule-𝒦 C)) (elections-𝒦 C) y′) // r}
    unfolding elections-𝒦_𝒬.simps
    using preimg-invar closed-domain cons-subset equiv-rel invar-C
    by blast
  ultimately have
    ∀ y′. Inf (distance-infimum_𝒬 d A ‘ preimg (π_𝒬 (elect-r ∘ fun_𝓔 (rule-𝒦 C)))
          (elections-𝒦_𝒬 r C) y′) =
      Inf {Inf {d a b | b. b ∈ B}
          | B. B ∈ preimg (elect-r ∘ fun_𝓔 (rule-𝒦 C)) (elections-𝒦 C) y′ // r}
    by simp
  thus ?thesis
    using wf-res-eq rewrite-inf-dist preimg-partition-dist
    by presburger
qed
from a-in-A
have π_𝒬 (fun_𝓔 (ℛ_𝒲 d C)) A = fun_𝓔 (ℛ_𝒲 d C) a
  using invar-dr equiv-rel quot-class pass-to-quotient invariance-is-congruence
  by blast
moreover have ∀ x. x ∈ fun_𝓔 (ℛ_𝒲 d C) a ⟷ x ∈ ℛ_𝒬 r d C A
proof
  fix x :: ′r
  have x ∈ fun_𝓔 (ℛ_𝒲 d C) a =
     (x ∈ ⋃ (minimizer (elect-r ∘ fun_𝓔 (rule-𝒦 C)) (elections-𝒦 C) d
          (singleton-set-system (limit (alternatives-𝓔 a) UNIV)) a))
    using ℛ_𝒲-is-minimizer
    by metis
  also have ... =
      ({x} ∈ minimizer (elect-r ∘ fun_𝓔 (rule-𝒦 C)) (elections-𝒦 C) d
          (singleton-set-system (limit (alternatives-𝓔 a) UNIV)) a)
    using singleton-set-union
    unfolding minimizer.simps arg-min-set.simps is-arg-min-def
    by auto
  also have ... = ({x} ∈ singleton-set-system (limit (alternatives-𝓔 a) UNIV)
        ∧ (∀ y' ∈ singleton-set-system (limit (alternatives-ℰ a) UNIV).
            Inf (d a ` preimg (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) {x})
          ≤ Inf (d a ` preimg (elect-r ∘ fun_ℰ (rule-𝒦 C)) (elections-𝒦 C) y')))
      using minimizer-helper
      by (metis (no-types, lifting))
    also have ... = ({x} ∈ singleton-set-system (limit_𝒬 A UNIV)
      ∧ (∀ y' ∈ singleton-set-system (limit_𝒬 A UNIV).
        Inf (distance-infimum_𝒬 d A ` preimg (π_𝒬 (elect-r ∘ fun_ℰ (rule-𝒦 C)))
              (elections-𝒦_𝒬 r C) {x})
        ≤ Inf (distance-infimum_𝒬 d A ` preimg (π_𝒬 (elect-r ∘ fun_ℰ (rule-𝒦 C)))
              (elections-𝒦_𝒬 r C) y')))
      using wf-res-eq inf-le-iff
      by blast
    also have ... =
        ({x} ∈ minimizer
            (π_𝒬 (elect-r ∘ fun_ℰ (rule-𝒦 C))) (elections-𝒦_𝒬 r C)
            (distance-infimum_𝒬 d)
              (singleton-set-system (limit_𝒬 A UNIV)) A)
      using minimizer-helper
      by (metis (no-types, lifting))
    also have ... =
      (x ∈ ⋃ (minimizer
            (π_𝒬 (elect-r ∘ fun_ℰ (rule-𝒦 C))) (elections-𝒦_𝒬 r C)
            (distance-infimum_𝒬 d)
              (singleton-set-system (limit_𝒬 A UNIV)) A))
      using singleton-set-union
      unfolding minimizer.simps arg-min-set.simps is-arg-min-def
      by auto
    finally show x ∈ fun_ℰ (ℛ_𝒲 d C) a = (x ∈ ℛ_𝒬 r d C A)
      unfolding ℛ_𝒬.simps
      by safe
  qed
  ultimately show π_𝒬 (fun_ℰ (ℛ_𝒲 d C)) A = ℛ_𝒬 r d C A
    by blast
qed

theorem (in result) invar-dr-simple-dist-imp-quotient-dr:
  fixes
    d :: ('a, 'v) Election Distance and
    C :: ('a, 'v, 'r Result) Consensus-Class and
    r :: ('a, 'v) Election rel and
    X A :: ('a, 'v) Election set
  assumes
    simple: simple r X d and
    closed-domain: closed-restricted-rel r X (elections-𝒦 C) and
    invar-res:
      is-symmetry (λ E. limit (alternatives-ℰ E) UNIV)
              (Invariance r) and
    invar-C: is-symmetry (elect-r ∘ fun_ℰ (rule-𝒦 C))
```

```
              (Invariance (Restr r (elections-𝒦 C))) and
    invar-dr: is-symmetry (fun_ℰ (ℛ_𝒲 d C)) (Invariance r) and
    quot-class: A ∈ X // r and
    equiv-rel: equiv X r and
    cons-subset: elections-𝒦 C ⊆ X
  shows π_𝒬 (fun_ℰ (distance-ℛ d C)) A = distance-ℛ_𝒬 r d C A
proof −
  have ∀ E. fun_ℰ (distance-ℛ d C) E =
        (fun_ℰ (ℛ_𝒲 d C) E,
          limit (alternatives-ℰ E) UNIV − fun_ℰ (ℛ_𝒲 d C) E,
          {})
    by simp
  moreover have ∀ E ∈ A. fun_ℰ (ℛ_𝒲 d C) E = π_𝒬 (fun_ℰ (ℛ_𝒲 d C)) A
    using invar-dr invariance-is-congruence pass-to-quotient quot-class equiv-rel
    by blast
  moreover have π_𝒬 (fun_ℰ (ℛ_𝒲 d C)) A = ℛ_𝒬 r d C A
    using invar-dr-simple-dist-imp-quotient-dr-winners assms
    by blast
  moreover have
    ∀ E ∈ A. limit (alternatives-ℰ E) UNIV =
        π_𝒬 (λ E. limit (alternatives-ℰ E) UNIV) A
    using invar-res invariance-is-congruence' pass-to-quotient quot-class equiv-rel
    by blast
  ultimately have all-eq:
    ∀ E ∈ A. fun_ℰ (distance-ℛ d C) E =
      (ℛ_𝒬 r d C A,
        π_𝒬 (λ E. limit (alternatives-ℰ E) UNIV) A − ℛ_𝒬 r d C A,
        {})
    by fastforce
  hence
    {(ℛ_𝒬 r d C A,
        π_𝒬 (λ E. limit (alternatives-ℰ E) UNIV) A − ℛ_𝒬 r d C A,
        {})} ⊇ fun_ℰ (distance-ℛ d C) ‘ A
    by blast
  moreover have A ≠ {}
    using quot-class equiv-rel in-quotient-imp-non-empty
    by metis
  ultimately have single-img:
    {(ℛ_𝒬 r d C A,
        π_𝒬 (λ E. limit (alternatives-ℰ E) UNIV) A − ℛ_𝒬 r d C A,
        {})} =
      fun_ℰ (distance-ℛ d C) ‘ A
    using empty-is-image subset-singletonD
    by (metis (no-types, lifting))
  moreover from this
  have card (fun_ℰ (distance-ℛ d C) ‘ A) = 1
    using is-singleton-altdef is-singletonI
    by (metis (no-types, lifting))
  moreover from this single-img
  have the-inv (λ x. {x}) (fun_ℰ (distance-ℛ d C) ‘ A) =
        (ℛ_𝒬 r d C A,
          π_𝒬 (λ E. limit (alternatives-ℰ E) UNIV) A − ℛ_𝒬 r d C A,
          {})
    using singleton-insert-inj-eq singleton-set.elims singleton-set-def-if-card-one
    by (metis (no-types))
  ultimately show ?thesis
    unfolding distance-ℛ_𝒬.simps π_𝒬.simps singleton-set.simps
    by presburger
qed

end
```

5.8 Code Generation Interpretations for Results and Properties

```
theory Interpretation-Code
  imports Electoral-Module
          Distance-Rationalization
begin
setup Locale-Code.open-block
```

5.8.1 Code Lemmas

Lemmas stating the explicit instantiations of interpreted abstract functions from locales.

```
lemma electoral-module-𝒮𝒞ℱ-code-lemma:
  fixes m :: ('a, 'v, 'a Result) Electoral-Module
  shows 𝒮𝒞ℱ-result.electoral-module m =
        (∀ A V p. profile V A p ⟶ well-formed-𝒮𝒞ℱ A (m V A p))
  unfolding 𝒮𝒞ℱ-result.electoral-module.simps
  by safe

lemma ℛ_𝒲-𝒮𝒞ℱ-code-lemma:
  fixes
    d :: ('a, 'v) Election Distance and
    K :: ('a, 'v, 'a Result) Consensus-Class and
    V :: 'v set and
    A :: 'a set and
    p :: ('a, 'v) Profile
  shows 𝒮𝒞ℱ-result.ℛ_𝒲 d K V A p =
          arg-min-set (score d K (A, V, p)) (limit-𝒮𝒞ℱ A UNIV)
  unfolding 𝒮𝒞ℱ-result.ℛ_𝒲.simps
  by safe

lemma distance-ℛ-𝒮𝒞ℱ-code-lemma:
```

fixes
d :: (*'a*, *'v*) *Election Distance* **and**
K :: (*'a*, *'v*, *'a Result*) *Consensus-Class* **and**
V :: *'v set* **and**
A :: *'a set* **and**
p :: (*'a*, *'v*) *Profile*
shows $\mathcal{SCF}$*-result.distance-*$\mathcal{R}$ *d K V A p* =
($\mathcal{SCF}$*-result.*$\mathcal{R}_\mathcal{W}$ *d K V A p*,
(*limit-*$\mathcal{SCF}$ *A UNIV*) − $\mathcal{SCF}$*-result.*$\mathcal{R}_\mathcal{W}$ *d K V A p*,
{})
unfolding $\mathcal{SCF}$*-result.distance-*$\mathcal{R}$*.simps*
by *safe*

lemma $\mathcal{R}_\mathcal{W}$*-std-*$\mathcal{SCF}$*-code-lemma*:
fixes
d :: (*'a*, *'v*) *Election Distance* **and**
K :: (*'a*, *'v*, *'a Result*) *Consensus-Class* **and**
V :: *'v set* **and**
A :: *'a set* **and**
p :: (*'a*, *'v*) *Profile*
shows $\mathcal{SCF}$*-result.*$\mathcal{R}_\mathcal{W}$*-std d K V A p* =
arg-min-set (*score-std d K* (*A*, *V*, *p*)) (*limit-*$\mathcal{SCF}$ *A UNIV*)
unfolding $\mathcal{SCF}$*-result.*$\mathcal{R}_\mathcal{W}$*-std.simps*
by *safe*

lemma *distance-*$\mathcal{R}$*-std-*$\mathcal{SCF}$*-code-lemma*:
fixes
d :: (*'a*, *'v*) *Election Distance* **and**
K :: (*'a*, *'v*, *'a Result*) *Consensus-Class* **and**
V :: *'v set* **and**
A :: *'a set* **and**
p :: (*'a*, *'v*) *Profile*
shows $\mathcal{SCF}$*-result.distance-*$\mathcal{R}$*-std d K V A p* =
($\mathcal{SCF}$*-result.*$\mathcal{R}_\mathcal{W}$*-std d K V A p*,
(*limit-*$\mathcal{SCF}$ *A UNIV*) − $\mathcal{SCF}$*-result.*$\mathcal{R}_\mathcal{W}$*-std d K V A p*,
{})
unfolding $\mathcal{SCF}$*-result.distance-*$\mathcal{R}$*-std.simps*
by *safe*

lemma *anonymity-*$\mathcal{SCF}$*-code-lemma*: $\mathcal{SCF}$*-result.anonymity* =
(λ *m* :: (*'a*, *'v*, *'a Result*) *Electoral-Module*.
$\mathcal{SCF}$*-result.electoral-module m* ∧
(∀ *A V p* π :: (*'v* ⇒ *'v*).
bij π ⟶ (*let* (*A'*, *V'*, *q*) = (*rename* π (*A*, *V*, *p*)) *in*
profile V A p ∧ *profile V' A' q* ⟶ *m V A p* = *m V' A' q*)))
unfolding $\mathcal{SCF}$*-result.anonymity-def*
by *simp*

5.8.2 Interpretation Declarations and Constants

Declarations for replacing interpreted abstract functions from locales by their explicit instantiations.

```
declare [[lc-add 𝒮𝒞ℱ-result.electoral-module electoral-module-𝒮𝒞ℱ-code-lemma]]
declare [[lc-add 𝒮𝒞ℱ-result.ℛ_𝒲 ℛ_𝒲-𝒮𝒞ℱ-code-lemma]]
declare [[lc-add 𝒮𝒞ℱ-result.ℛ_𝒲-std ℛ_𝒲-std-𝒮𝒞ℱ-code-lemma]]
declare [[lc-add 𝒮𝒞ℱ-result.distance-ℛ distance-ℛ-𝒮𝒞ℱ-code-lemma]]
declare [[lc-add 𝒮𝒞ℱ-result.distance-ℛ-std distance-ℛ-std-𝒮𝒞ℱ-code-lemma]]
declare [[lc-add 𝒮𝒞ℱ-result.anonymity anonymity-𝒮𝒞ℱ-code-lemma]]
```

Constant aliases to use instead of the interpreted functions.

```
definition ℛ_𝒲-𝒮𝒞ℱ-code ≡ 𝒮𝒞ℱ-result.ℛ_𝒲
definition ℛ_𝒲-std-𝒮𝒞ℱ-code ≡ 𝒮𝒞ℱ-result.ℛ_𝒲-std
definition distance-ℛ-𝒮𝒞ℱ-code ≡ 𝒮𝒞ℱ-result.distance-ℛ
definition distance-ℛ-std-𝒮𝒞ℱ-code ≡ 𝒮𝒞ℱ-result.distance-ℛ-std
definition electoral-module-𝒮𝒞ℱ-code ≡ 𝒮𝒞ℱ-result.electoral-module
definition anonymity-𝒮𝒞ℱ-code ≡ 𝒮𝒞ℱ-result.anonymity

setup Locale-Code.close-block

end
```

5.9 Drop Module

```
theory Drop-Module
  imports Component-Types/Electoral-Module
          Component-Types/Social-Choice-Types/Result
begin
```

This is a family of electoral modules. For a natural number n and a lexicon (linear order) r of all alternatives, the according drop module rejects the lexicographically first n alternatives (from A) and defers the rest. It is primarily used as counterpart to the pass module in a parallel composition, in order to segment the alternatives into two groups.

5.9.1 Definition

```
fun drop-module :: nat ⇒ 'a Preference-Relation ⇒
      ('a, 'v, 'a Result) Electoral-Module where
  drop-module n r V A p =
    ({},
    {a ∈ A. rank (limit A r) a ≤ n},
    {a ∈ A. rank (limit A r) a > n})
```

5.9.2 Soundness

```
theorem drop-mod-sound[simp]:
  fixes
    r :: 'a Preference-Relation and
    n :: nat
  shows SCF-result.electoral-module (drop-module n r)
proof (unfold SCF-result.electoral-module.simps, safe)
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assume profile V A p
  let ?mod = drop-module n r
  have ∀ a ∈ A. a ∈ {x ∈ A. rank (limit A r) x ≤ n} ∨
                a ∈ {x ∈ A. rank (limit A r) x > n}
    by auto
  hence {a ∈ A. rank (limit A r) a ≤ n} ∪ {a ∈ A. rank (limit A r) a > n} = A
    by blast
  hence set-partition: set-equals-partition A (drop-module n r V A p)
    by simp
  have ∀ a ∈ A.
          ¬ (a ∈ {x ∈ A. rank (limit A r) x ≤ n} ∧
             a ∈ {x ∈ A. rank (limit A r) x > n})
    by simp
  hence {a ∈ A. rank (limit A r) a ≤ n} ∩ {a ∈ A. rank (limit A r) a > n} = {}
    by blast
  thus well-formed-SCF A (?mod V A p)
    using set-partition
    by simp
qed

lemma voters-determine-drop-mod:
  fixes
    r :: 'a Preference-Relation and
    n :: nat
  shows voters-determine-election (drop-module n r)
  unfolding voters-determine-election.simps
  by simp
```

5.9.3 Non-Electing

The drop module is non-electing.

```
theorem drop-mod-non-electing[simp]:
  fixes
    r :: 'a Preference-Relation and
    n :: nat
  shows non-electing (drop-module n r)
  unfolding non-electing-def
  by auto
```

5.9.4 Properties

The drop module is strictly defer-monotone.

```
theorem drop-mod-def-lift-inv[simp]:
  fixes
    r :: 'a Preference-Relation and
    n :: nat
  shows defer-lift-invariance (drop-module n r)
  unfolding defer-lift-invariance-def
  by force

end
```

5.10 Pass Module

```
theory Pass-Module
  imports Component-Types/Electoral-Module
begin
```

This is a family of electoral modules. For a natural number n and a lexicon (linear order) r of all alternatives, the according pass module defers the lexicographically first n alternatives (from A) and rejects the rest. It is primarily used as counterpart to the drop module in a parallel composition in order to segment the alternatives into two groups.

5.10.1 Definition

```
fun pass-module :: nat ⇒ 'a Preference-Relation ⇒
        ('a, 'v, 'a Result) Electoral-Module where
  pass-module n r V A p =
    ({},
    {a ∈ A. rank (limit A r) a > n},
    {a ∈ A. rank (limit A r) a ≤ n})
```

5.10.2 Soundness

```
theorem pass-mod-sound[simp]:
  fixes
    r :: 'a Preference-Relation and
    n :: nat
  shows 𝒮𝒞ℱ-result.electoral-module (pass-module n r)
proof (unfold 𝒮𝒞ℱ-result.electoral-module.simps, safe)
```

```
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  let ?mod = pass-module n r
  have ∀ a ∈ A. a ∈ {x ∈ A. rank (limit A r) x > n} ∨
               a ∈ {x ∈ A. rank (limit A r) x ≤ n}
    using CollectI not-less
    by metis
  hence {a ∈ A. rank (limit A r) a > n} ∪ {a ∈ A. rank (limit A r) a ≤ n} = A
    by blast
  hence set-equals-partition A (pass-module n r V A p)
    by simp
  moreover have
    ∀ a ∈ A.
      ¬ (a ∈ {x ∈ A. rank (limit A r) x > n} ∧
          a ∈ {x ∈ A. rank (limit A r) x ≤ n})
    by simp
  hence {a ∈ A. rank (limit A r) a > n} ∩ {a ∈ A. rank (limit A r) a ≤ n} = {}
    by blast
  ultimately show well-formed-𝒮𝒞ℱ A (?mod V A p)
    by simp
qed

lemma voters-determine-pass-mod:
  fixes
    r :: 'a Preference-Relation and
    n :: nat
  shows voters-determine-election (pass-module n r)
  unfolding voters-determine-election.simps pass-module.simps
  by blast
```

5.10.3 Non-Blocking

The pass module is non-blocking.

```
theorem pass-mod-non-blocking[simp]:
  fixes
    r :: 'a Preference-Relation and
    n :: nat
  assumes
    order: linear-order r and
    greater-zero: n > 0
  shows non-blocking (pass-module n r)
proof (unfold non-blocking-def, safe)
  show 𝒮𝒞ℱ-result.electoral-module (pass-module n r)
    using pass-mod-sound
    by metis
next
  fix
```

```
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assume
    fin-A: finite A and
    rej-pass-A: reject (pass-module n r) V A p = A and
    a-in-A: a ∈ A
  moreover have lin: linear-order-on A (limit A r)
    using limit-presv-lin-ord order top-greatest
    by metis
  moreover have
    ∃ b ∈ A. above (limit A r) b = {b}
      ∧ (∀ c ∈ A. above (limit A r) c = {c} ⟶ c = b)
    using fin-A a-in-A lin above-one
    by blast
  moreover have {b ∈ A. rank (limit A r) b > n} ≠ A
    using Suc-leI greater-zero leD mem-Collect-eq above-rank calculation
    unfolding One-nat-def
    by (metis (no-types, lifting))
  hence reject (pass-module n r) V A p ≠ A
    by simp
  thus a ∈ {}
    using rej-pass-A
    by simp
qed
```

5.10.4 Non-Electing

The pass module is non-electing.

```
theorem pass-mod-non-electing[simp]:
  fixes
    r :: 'a Preference-Relation and
    n :: nat
  assumes linear-order r
  shows non-electing (pass-module n r)
  unfolding non-electing-def
  using assms
  by force
```

5.10.5 Properties

The pass module is strictly defer-monotone.

```
theorem pass-mod-dl-inv[simp]:
  fixes
    r :: 'a Preference-Relation and
    n :: nat
  assumes linear-order r
  shows defer-lift-invariance (pass-module n r)
  unfolding defer-lift-invariance-def
  using assms pass-mod-sound
  by simp

theorem pass-zero-mod-def-zero[simp]:
  fixes r :: 'a Preference-Relation
  assumes linear-order r
  shows defers 0 (pass-module 0 r)
proof (unfold defers-def, safe)
  show 𝒮𝒞ℱ-result.electoral-module (pass-module 0 r)
    using pass-mod-sound assms
    by metis
next
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assume
    card-pos: 0 ≤ card A and
    finite-A: finite A and
    prof-A: profile V A p
  have linear-order-on A (limit A r)
    using assms limit-presv-lin-ord
    by blast
  hence limit-is-connex: connex A (limit A r)
    using lin-ord-imp-connex
    by simp
  have ∀ n. (n :: nat) ≤ 0 ⟶ n = 0
    by blast
  hence ∀ a A'. a ∈ A' ∧ a ∈ A ⟶ connex A' (limit A r) ⟶
        ¬ rank (limit A r) a ≤ 0
    using above-connex above-presv-limit card-eq-0-iff equals0D finite-A
        assms rev-finite-subset
    unfolding rank.simps
    by (metis (no-types))
  hence {a ∈ A. rank (limit A r) a ≤ 0} = {}
    using limit-is-connex
    by simp
  hence card {a ∈ A. rank (limit A r) a ≤ 0} = 0
    using card.empty
    by metis
  thus card (defer (pass-module 0 r) V A p) = 0
    by simp
qed
```

For any natural number n and any linear order, the according pass module defers n alternatives (if there are n alternatives). NOTE: The induction proof is still missing. The following are the proofs for n=1 and n=2.

```
theorem pass-one-mod-def-one[simp]:
  fixes r :: 'a Preference-Relation
  assumes linear-order r
  shows defers 1 (pass-module 1 r)
proof (unfold defers-def, safe)
  show 𝒮𝒞ℱ-result.electoral-module (pass-module 1 r)
    using pass-mod-sound assms
    by simp
next
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assume
    card-pos: 1 ≤ card A and
    finite-A: finite A and
    prof-A: profile V A p
  show card (defer (pass-module 1 r) V A p) = 1
  proof −
    have A ≠ {}
      using card-pos
      by auto
    moreover have lin-ord-on-A: linear-order-on A (limit A r)
      using assms limit-presv-lin-ord
      by blast
    ultimately have winner-exists:
      ∃ a ∈ A. above (limit A r) a = {a} ∧
          (∀ b ∈ A. above (limit A r) b = {b} ⟶ b = a)
      using finite-A above-one
      by simp
    then obtain w :: 'a where
      w-unique-top:
      above (limit A r) w = {w} ∧
        (∀ a ∈ A. above (limit A r) a = {a} ⟶ a = w)
      using above-one
      by auto
    hence {a ∈ A. rank (limit A r) a ≤ 1} = {w}
    proof
      assume
        w-top: above (limit A r) w = {w} and
        w-unique: ∀ a ∈ A. above (limit A r) a = {a} ⟶ a = w
      have rank (limit A r) w ≤ 1
        using w-top
        by auto
      hence {w} ⊆ {a ∈ A. rank (limit A r) a ≤ 1}
        using winner-exists w-unique-top
        by blast
      moreover have {a ∈ A. rank (limit A r) a ≤ 1} ⊆ {w}
      proof
```

```
        fix a :: 'a
        assume a-in-winner-set: a ∈ {b ∈ A. rank (limit A r) b ≤ 1}
        hence a-in-A: a ∈ A
          by auto
        hence connex-limit: connex A (limit A r)
          using lin-ord-imp-connex lin-ord-on-A
          by simp
        hence let q = limit A r in a ≼q a
          using connex-limit above-connex pref-imp-in-above a-in-A
          by metis
        hence (a, a) ∈ limit A r
          by simp
        hence a-above-a: a ∈ above (limit A r) a
          unfolding above-def
          by simp
        have above (limit A r) a ⊆ A
          using above-presv-limit assms
          by fastforce
        hence above-finite: finite (above (limit A r) a)
          using finite-A finite-subset
          by simp
        have rank (limit A r) a ≤ 1
          using a-in-winner-set
          by simp
        moreover have rank (limit A r) a ≥ 1
          using Suc-leI above-finite card-eq-0-iff equals0D neq0-conv a-above-a
          unfolding rank.simps One-nat-def
          by metis
        ultimately have rank (limit A r) a = 1
          by simp
        hence {a} = above (limit A r) a
          using a-above-a lin-ord-on-A rank-one-imp-above-one
          by metis
        hence a = w
          using w-unique a-in-A
          by simp
        thus a ∈ {w}
          by simp
      qed
      ultimately have {w} = {a ∈ A. rank (limit A r) a ≤ 1}
        by auto
      thus ?thesis
        by simp
    qed
    thus card (defer (pass-module 1 r) V A p) = 1
      by simp
  qed
qed
```

```
theorem pass-two-mod-def-two:
  fixes r :: 'a Preference-Relation
  assumes linear-order r
  shows defers 2 (pass-module 2 r)
proof (unfold defers-def, safe)
  show 𝒮𝒞ℱ-result.electoral-module (pass-module 2 r)
    using assms pass-mod-sound
    by metis
next
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assume
    min-card-two: 2 ≤ card A and
    fin-A: finite A and
    prof-A: profile V A p
  from min-card-two
  have not-empty-A: A ≠ {}
    by auto
  moreover have limit-A-order: linear-order-on A (limit A r)
    using limit-presv-lin-ord assms
    by auto
  ultimately obtain a :: 'a where
    above (limit A r) a = {a}
    using above-one min-card-two fin-A prof-A
    by blast
  hence ∀ b ∈ A. let q = limit A r in (b ⪯_q a)
    using limit-A-order pref-imp-in-above empty-iff lin-ord-imp-connex
          insert-iff insert-subset above-presv-limit assms
    unfolding connex-def
    by metis
  hence a-best: ∀ b ∈ A. (b, a) ∈ limit A r
    by simp
  hence a-above: ∀ b ∈ A. a ∈ above (limit A r) b
    unfolding above-def
    by simp
  hence a ∈ {a ∈ A. rank (limit A r) a ≤ 2}
    using CollectI not-empty-A empty-iff fin-A insert-iff limit-A-order
          above-one above-rank one-le-numeral
    by (metis (no-types, lifting))
  hence a-in-defer: a ∈ defer (pass-module 2 r) V A p
    by simp
  have finite (A − {a})
    using fin-A
    by simp
  moreover have A-not-only-a: A − {a} ≠ {}
    using Diff-empty Diff-idemp Diff-insert0 not-empty-A insert-Diff finite.emptyI
          card.insert-remove card.empty min-card-two Suc-n-not-le-n numeral-2-eq-2
  by metis
moreover have limit-A-without-a-order:
  linear-order-on (A − {a}) (limit (A − {a}) r)
  using limit-presv-lin-ord assms top-greatest
  by blast
ultimately obtain b :: 'a where
  top-b: above (limit (A − {a}) r) b = {b}
  using above-one
  by metis
hence ∀ c ∈ A − {a}. let q = limit (A − {a}) r in (c ≼q b)
  using limit-A-without-a-order pref-imp-in-above empty-iff lin-ord-imp-connex
        insert-iff insert-subset above-presv-limit assms
  unfolding connex-def
  by metis
hence b-in-limit: ∀ c ∈ A − {a}. (c, b) ∈ limit (A − {a}) r
  by simp
hence b-best: ∀ c ∈ A − {a}. (c, b) ∈ limit A r
  by auto
hence ∀ c ∈ A − {a, b}. c ∉ above (limit A r) b
  using top-b Diff-iff Diff-insert2 above-presv-limit insert-subset
        assms limit-presv-above limit-rel-presv-above
  by metis
moreover have above-subset: above (limit A r) b ⊆ A
  using above-presv-limit assms
  by metis
moreover have b-above-b: b ∈ above (limit A r) b
  using top-b b-best above-presv-limit mem-Collect-eq assms insert-subset
  unfolding above-def
  by metis
ultimately have above-b-eq-ab: above (limit A r) b = {a, b}
  using a-above
  by auto
hence card-above-b-eq-two: rank (limit A r) b = 2
  using A-not-only-a b-in-limit
  by auto
hence b-in-defer: b ∈ defer (pass-module 2 r) V A p
  using b-above-b above-subset
  by auto
have b-above: ∀ c ∈ A − {a}. b ∈ above (limit A r) c
  using b-best mem-Collect-eq
  unfolding above-def
  by metis
have connex A (limit A r)
  using limit-A-order lin-ord-imp-connex
  by auto
hence ∀ c ∈ A. c ∈ above (limit A r) c
  using above-connex
  by metis
hence ∀ c ∈ A − {a, b}. {a, b, c} ⊆ above (limit A r) c
```

```
    using a-above b-above
    by auto
  moreover have ∀ c ∈ A − {a, b}. card {a, b, c} = 3
    using DiffE Suc-1 above-b-eq-ab card-above-b-eq-two above-subset fin-A
          card-insert-disjoint finite-subset insert-commute numeral-3-eq-3
    unfolding One-nat-def rank.simps
    by metis
  ultimately have ∀ c ∈ A − {a, b}. rank (limit A r) c ≥ 3
    using card-mono fin-A finite-subset above-presv-limit assms
    unfolding rank.simps
    by metis
  hence ∀ c ∈ A − {a, b}. rank (limit A r) c > 2
    using Suc-le-eq Suc-1 numeral-3-eq-3
    unfolding One-nat-def
    by metis
  hence ∀ c ∈ A − {a, b}. c ∉ defer (pass-module 2 r) V A p
    by (simp add: not-le)
  moreover have defer (pass-module 2 r) V A p ⊆ A
    by auto
  ultimately have defer (pass-module 2 r) V A p ⊆ {a, b}
    by blast
  hence defer (pass-module 2 r) V A p = {a, b}
    using a-in-defer b-in-defer
    by fastforce
  thus card (defer (pass-module 2 r) V A p) = 2
    using above-b-eq-ab card-above-b-eq-two
    unfolding rank.simps
    by presburger
qed

end
```

5.11 Elect Module

```
theory Elect-Module
  imports Component-Types/Electoral-Module
begin
```

The elect module is not concerned about the voter's ballots, and just elects all alternatives. It is primarily used in sequence after an electoral module that only defers alternatives to finalize the decision, thereby inducing a proper voting rule in the social choice sense.

5.11.1 Definition

fun *elect-module* :: (*'a*, *'v*, *'a Result*) *Electoral-Module* **where**
  elect-module V A p = (*A*, {}, {})

5.11.2 Soundness

theorem *elect-mod-sound*[*simp*]: $\mathcal{SCF}$*-result.electoral-module elect-module*
  by *simp*

lemma *elect-mod-only-voters*: *voters-determine-election elect-module*
  by *simp*

5.11.3 Electing

theorem *elect-mod-electing*[*simp*]: *electing elect-module*
  unfolding *electing-def*
  by *simp*

end

5.12 Plurality Module

theory *Plurality-Module*
  imports *Component-Types/Elimination-Module*
begin

The plurality module implements the plurality voting rule. The plurality rule elects all modules with the maximum amount of top preferences among all alternatives, and rejects all the other alternatives. It is electing and induces the classical plurality (voting) rule from social-choice theory.

5.12.1 Definition

fun *plurality-score* :: (*'a*, *'v*) *Evaluation-Function* **where**
  plurality-score V x A p = *win-count V p x*

fun *plurality* :: (*'a*, *'v*, *'a Result*) *Electoral-Module* **where**
  plurality V A p = *max-eliminator plurality-score V A p*

fun *plurality'* :: (*'a*, *'v*, *'a Result*) *Electoral-Module* **where**
  plurality' V A p =
    (*if finite A*
      then ({},
        {$a \in A$. $\exists$ $x \in A$. *win-count V p x* > *win-count V p a*},

```
             {a ∈ A. ∀ x ∈ A. win-count V p x ≤ win-count V p a})
        else ({}, {}, A))

lemma enat-leq-enat-set-max:
  fixes
    x :: enat and
    X :: enat set
  assumes
    x ∈ X and
    finite X
  shows x ≤ Max X
  using assms
  by simp

lemma plurality-mod-equiv:
  fixes
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  shows plurality V A p = plurality' V A p
proof (unfold plurality'.simps)
  have no-winners-or-in-domain:
    finite {win-count V p a | a. a ∈ A} ⟶
      {win-count V p a | a. a ∈ A} = {} ∨
        Max {win-count V p a | a. a ∈ A} ∈ {win-count V p a | a. a ∈ A}
    using Max-in
    by blast
  moreover have only-one-max:
    finite A ⟶
      {a ∈ A. ∃ x ∈ A. win-count V p a < win-count V p x} =
        {a ∈ A. win-count V p a < Max {win-count V p x | x. x ∈ A}}
  proof
    assume fin-A: finite A
    hence finite {win-count V p x | x. x ∈ A}
      by simp
    hence
      ∀ a ∈ A. ∀ b ∈ A. win-count V p a < win-count V p b ⟶
        win-count V p a < Max {win-count V p x | x. x ∈ A}
      using CollectI Max-gr-iff empty-Collect-eq
      by (metis (mono-tags, lifting))
    hence
      {a ∈ A. ∃ x ∈ A. win-count V p a < win-count V p x}
        ⊆ {a ∈ A. win-count V p a < Max {win-count V p x | x. x ∈ A}}
      by blast
    moreover have
      {a ∈ A. win-count V p a < Max {win-count V p x | x. x ∈ A}}
        ⊆ {a ∈ A. ∃ x ∈ A. win-count V p a < win-count V p x}
      using fin-A no-winners-or-in-domain
      by fastforce
    ultimately show
      {a ∈ A. ∃ x ∈ A. win-count V p a < win-count V p x} =
        {a ∈ A. win-count V p a < Max {win-count V p x | x. x ∈ A}}
      by fastforce
  qed
  ultimately have
    finite A ⟶
      reject plurality V A p = {a ∈ A. ∃ x ∈ A. win-count V p a < win-count V p
x}
    by force
  moreover have
    finite A ⟶
      defer plurality V A p = {a ∈ A. ∀ b ∈ A. win-count V p b ≤ win-count V p
a}
  proof
    assume fin-A: finite A
    have {a ∈ A. ∃ x ∈ A. win-count V p x > win-count V p a}
          ∩ {a ∈ A. ∀ x ∈ A. win-count V p x ≤ win-count V p a} = {}
      by force
    moreover have
      {a ∈ A. ∃ x ∈ A. win-count V p x > win-count V p a}
        ∪ {a ∈ A. ∀ x ∈ A. win-count V p x ≤ win-count V p a} = A
      by force
    ultimately have
      {a ∈ A. ∀ b ∈ A. win-count V p b ≤ win-count V p a} =
          A − {a ∈ A. win-count V p a < Max {win-count V p x | x. x ∈ A}}
      using fin-A only-one-max Diff-cancel Int-Diff-Un Un-Diff inf-commute
      by (metis (no-types, lifting))
    moreover have
      {a ∈ A. win-count V p a < Max {win-count V p x | x. x ∈ A}} = A ⟶
      {a ∈ A. ∀ b ∈ A. win-count V p b ≤ win-count V p a} = A
      using fin-A no-winners-or-in-domain
      by fastforce
    ultimately show
      defer plurality V A p = {a ∈ A. ∀ b ∈ A. win-count V p b ≤ win-count V p
a}
      using fin-A
      by force
  qed
  moreover have elect plurality V A p = {}
   unfolding max-eliminator.simps less-eliminator.simps elimination-module.simps
    by force
  ultimately have
    finite A ⟶
      plurality V A p =
        ({}, {a ∈ A. ∃ x ∈ A. win-count V p a < win-count V p x},
          {a ∈ A. ∀ x ∈ A. win-count V p x ≤ win-count V p a})
    using split-pairs
    by (metis (no-types, lifting))
  thus plurality V A p =
    (if finite A
     then ({}, {a ∈ A. ∃ x ∈ A. win-count V p a < win-count V p x},
           {a ∈ A. ∀ x ∈ A. win-count V p x ≤ win-count V p a})
     else ({}, {}, A))
    by force
qed
```

5.12.2 Soundness

```
theorem plurality-sound[simp]: SCF-result.electoral-module plurality
  unfolding plurality.simps
  using max-elim-sound
  by metis

theorem plurality′-sound[simp]: SCF-result.electoral-module plurality′
proof (unfold SCF-result.electoral-module.simps, safe)
  fix
    A :: ′a set and
    V :: ′v set and
    p :: (′a, ′v) Profile
  have disjoint3 (
      {},
      {a ∈ A. ∃ a′ ∈ A. win-count V p a < win-count V p a′},
      {a ∈ A. ∀ a′ ∈ A. win-count V p a′ ≤ win-count V p a})
    by auto
  moreover have
    {a ∈ A. ∃ x ∈ A. win-count V p a < win-count V p x} ∪
      {a ∈ A. ∀ x ∈ A. win-count V p x ≤ win-count V p a} = A
    using not-le-imp-less
    by blast
  ultimately show well-formed-SCF A (plurality′ V A p)
    by simp
qed

lemma voters-determine-plurality-score: voters-determine-evaluation plurality-score
proof (unfold plurality-score.simps voters-determine-evaluation.simps, safe)
  fix
    A :: ′b set and
    V :: ′a set and
    p p′ :: (′b, ′a) Profile and
    a :: ′b
  assume
    ∀ v ∈ V. p v = p′ v and
    a ∈ A
  hence finite V ⟶
    card {v ∈ V. above (p v) a = {a}} = card {v ∈ V. above (p′ v) a = {a}}
    using Collect-cong
    by (metis (no-types, lifting))
  thus win-count V p a = win-count V p′ a
    unfolding win-count.simps
    by presburger
qed

lemma voters-determine-plurality: voters-determine-election plurality
  unfolding plurality.simps
  using voters-determine-max-elim voters-determine-plurality-score
  by blast

lemma voters-determine-plurality′: voters-determine-election plurality′
proof (unfold voters-determine-election.simps, safe)
  fix
    A :: ′k set and
    V :: ′v set and
    p p′ :: (′k, ′v) Profile
  assume ∀ v ∈ V. p v = p′ v
  thus plurality′ V A p = plurality′ V A p′
    using voters-determine-plurality plurality-mod-equiv
    unfolding voters-determine-election.simps
    by (metis (full-types))
qed
```

5.12.3 Non-Blocking

The plurality module is non-blocking.

```
theorem plurality-mod-non-blocking[simp]: non-blocking plurality
  unfolding plurality.simps
  using max-elim-non-blocking
  by metis

theorem plurality′-mod-non-blocking[simp]: non-blocking plurality′
  using plurality-mod-non-blocking plurality-mod-equiv plurality′-sound
  unfolding non-blocking-def
  by metis
```

5.12.4 Non-Electing

The plurality module is non-electing.

```
theorem plurality-non-electing[simp]: non-electing plurality
  using max-elim-non-electing
  unfolding plurality.simps non-electing-def
  by metis

theorem plurality′-non-electing[simp]: non-electing plurality′
  unfolding non-electing-def
  using plurality′-sound
  by simp
```

5.12.5 Property

```
lemma plurality-def-inv-mono-alts:
  fixes
    A :: 'a set and
    V :: 'v set and
    p q :: ('a, 'v) Profile and
    a :: 'a
  assumes
    defer-a: a ∈ defer plurality V A p and
    lift-a: lifted V A p q a
  shows defer plurality V A q = defer plurality V A p
          ∨ defer plurality V A q = {a}
proof −
  have set-disj: ∀ b c. b ∉ {c} ∨ b = c
    by blast
  have lifted-winner: ∀ b ∈ A. ∀ i ∈ V.
      above (p i) b = {b} ⟶ (above (q i) b = {b} ∨ above (q i) a = {a})
    using lift-a lifted-above-winner-alts
    unfolding Profile.lifted-def
    by metis
  hence ∀ i ∈ V. (above (p i) a = {a} ⟶ above (q i) a = {a})
    using defer-a lift-a
    unfolding Profile.lifted-def
    by metis
  hence a-win-subset:
    {i ∈ V. above (p i) a = {a}} ⊆ {i ∈ V. above (q i) a = {a}}
    by blast
  moreover have lifted-prof: profile V A q
    using lift-a
    unfolding Profile.lifted-def
    by metis
  ultimately have win-count-a: win-count V p a ≤ win-count V q a
    by (simp add: card-mono)
  have fin-A: finite A
    using lift-a
    unfolding Profile.lifted-def
    by blast
  hence ∀ b ∈ A − {a}.
        ∀ i ∈ V. (above (q i) a = {a} ⟶ above (q i) b ≠ {b})
    using DiffE above-one lift-a insertCI insert-absorb insert-not-empty
    unfolding Profile.lifted-def profile-def
    by metis
  with lifted-winner
  have above-QtoP:
    ∀ b ∈ A − {a}.
      ∀ i ∈ V. (above (q i) b = {b} ⟶ above (p i) b = {b})
    using lifted-above-winner-other lift-a
    unfolding Profile.lifted-def
    by metis
```

```
hence ∀ b ∈ A − {a}.
        {i ∈ V. above (q i) b = {b}} ⊆ {i ∈ V. above (p i) b = {b}}
  by (simp add: Collect-mono)
hence win-count-other: ∀ b ∈ A − {a}. win-count V p b ≥ win-count V q b
  by (simp add: card-mono)
show defer plurality V A q = defer plurality V A p
      ∨ defer plurality V A q = {a}
proof (cases)
  assume win-count V p a = win-count V q a
  hence card {i ∈ V. above (p i) a = {a}} = card {i ∈ V. above (q i) a = {a}}
    using win-count.simps Profile.lifted-def enat.inject lift-a
    by (metis (mono-tags, lifting))
  moreover have finite {i ∈ V. above (q i) a = {a}}
    using Collect-mem-eq Profile.lifted-def finite-Collect-conjI lift-a
    by (metis (mono-tags))
  ultimately have {i ∈ V. above (p i) a = {a}} = {i ∈ V. above (q i) a = {a}}
    using a-win-subset
    by (simp add: card-subset-eq)
  hence above-pq: ∀ i ∈ V. (above (p i) a = {a}) = (above (q i) a = {a})
    by blast
  moreover have
    ∀ b ∈ A − {a}. ∀ i ∈ V.
        (above (p i) b = {b} ⟶ (above (q i) b = {b} ∨ above (q i) a = {a}))
    using lifted-winner
    by auto
  moreover have
    ∀ b ∈ A − {a}. ∀ i ∈ V. (above (p i) b = {b} ⟶ above (p i) a ≠ {a})
  proof (intro ballI impI, safe)
    fix
      b :: 'a and
      i :: 'v
    assume
      b ∈ A and
      i ∈ V
    moreover from this have A-not-empty: A ≠ {}
      by blast
    ultimately have linear-order-on A (p i)
      using lift-a
      unfolding lifted-def profile-def
      by metis
    moreover assume
      b-neq-a: b ≠ a and
      abv-b: above (p i) b = {b} and
      abv-a: above (p i) a = {a}
    ultimately show False
      using above-one-eq A-not-empty fin-A
      by (metis (no-types))
  qed
  ultimately have above-PtoQ:
    ∀ b ∈ A − {a}. ∀ i ∈ V. (above (p i) b = {b} ⟶ above (q i) b = {b})
    by simp
  hence ∀ b ∈ A.
          card {i ∈ V. above (p i) b = {b}} =
            card {i ∈ V. above (q i) b = {b}}
  proof (safe)
    fix b :: 'a
    assume b ∈ A
    thus card {i ∈ V. above (p i) b = {b}} =
            card {i ∈ V. above (q i) b = {b}}
      using DiffI set-disj above-PtoQ above-QtoP above-pq
      by (metis (no-types, lifting))
  qed
  hence {b ∈ A. ∀ c ∈ A. win-count V p c ≤ win-count V p b} =
          {b ∈ A. ∀ c ∈ A. win-count V q c ≤ win-count V q b}
    by auto
  hence defer plurality' V A q = defer plurality' V A p
        ∨ defer plurality' V A q = {a}
    by simp
  hence defer plurality V A q = defer plurality V A p
          ∨ defer plurality V A q = {a}
    using plurality-mod-equiv lift-a
    unfolding Profile.lifted-def
    by (metis (no-types, opaque-lifting))
  thus ?thesis
    by simp
next
  assume win-count V p a ≠ win-count V q a
  hence strict-less: win-count V p a < win-count V q a
    using win-count-a
    by simp
  have a ∈ defer plurality V A p
    using defer-a plurality.elims
    by (metis (no-types))
  moreover have non-empty-A: A ≠ {}
    using lift-a equals0D equiv-prof-except-a-def
          lifted-imp-equiv-prof-except-a
    by metis
  moreover have fin-A: finite-profile V A p
    using lift-a
    unfolding Profile.lifted-def
    by simp
  ultimately have a ∈ defer plurality' V A p
    using plurality-mod-equiv
    by metis
  hence a-in-win-p:
    a ∈ {b ∈ A. ∀ c ∈ A. win-count V p c ≤ win-count V p b}
    using fin-A
    by simp
```

```
hence ∀ b ∈ A. win-count V p b ≤ win-count V p a
  by simp
hence less: ∀ b ∈ A − {a}. win-count V q b < win-count V q a
  using DiffD1 antisym dual-order.trans not-le-imp-less
        win-count-a strict-less win-count-other
  by metis
hence ∀ b ∈ A − {a}. ¬ (∀ c ∈ A. win-count V q c ≤ win-count V q b)
  using lift-a not-le
  unfolding Profile.lifted-def
  by metis
hence ∀ b ∈ A − {a}.
        b ∉ {c ∈ A. ∀ b ∈ A. win-count V q b ≤ win-count V q c}
  by blast
hence ∀ b ∈ A − {a}. b ∉ defer plurality' V A q
  using fin-A
  by simp
hence ∀ b ∈ A − {a}. b ∉ defer plurality V A q
  using lift-a non-empty-A plurality-mod-equiv
  unfolding Profile.lifted-def
  by (metis (no-types, lifting))
hence ∀ b ∈ A − {a}. b ∉ defer plurality V A q
  by simp
moreover have a ∈ defer plurality V A q
proof −
  have ∀ b ∈ A − {a}. win-count V q b ≤ win-count V q a
    using less less-imp-le
    by metis
  moreover have win-count V q a ≤ win-count V q a
    by simp
  ultimately have ∀ b ∈ A. win-count V q b ≤ win-count V q a
    by auto
  moreover have a ∈ A
    using a-in-win-p
    by simp
  ultimately have
    a ∈ {b ∈ A. ∀ c ∈ A. win-count V q c ≤ win-count V q b}
    by simp
  hence a ∈ defer plurality' V A q
    by simp
  hence a ∈ defer plurality V A q
    using plurality-mod-equiv non-empty-A fin-A lift-a non-empty-A
    unfolding Profile.lifted-def
    by (metis (no-types))
  thus ?thesis
    by simp
qed
moreover have defer plurality V A q ⊆ A
  by simp
ultimately show ?thesis
```

```
      by blast
  qed
qed

lemma plurality′-def-inv-mono-alts:
  fixes
    A :: ′a set and
    V :: ′v set and
    p q :: (′a, ′v) Profile and
    a :: ′a
  assumes
    a ∈ defer plurality′ V A p and
    lifted V A p q a
  shows defer plurality′ V A q = defer plurality′ V A p
          ∨ defer plurality′ V A q = {a}
  using assms plurality-def-inv-mono-alts plurality-mod-equiv
  by (metis (no-types))
```

The plurality rule is invariant-monotone.

```
theorem plurality-mod-def-inv-mono[simp]: defer-invariant-monotonicity plurality
proof (unfold defer-invariant-monotonicity-def, intro conjI impI allI)
  show 𝒮𝒞ℱ-result.electoral-module plurality
    using plurality-sound
    by metis
next
  show non-electing plurality
    by simp
next
  fix
    A :: ′b set and
    V :: ′a set and
    p q :: (′b, ′a) Profile and
    a :: ′b
  assume a ∈ defer plurality V A p ∧ Profile.lifted V A p q a
  hence defer plurality V A q = defer plurality V A p
          ∨ defer plurality V A q = {a}
    using plurality-def-inv-mono-alts
    by metis
  thus defer plurality V A q = defer plurality V A p
          ∨ defer plurality V A q = {a}
    by simp
qed

theorem plurality′-mod-def-inv-mono[simp]: defer-invariant-monotonicity plurality′
  using plurality-mod-def-inv-mono plurality-mod-equiv
        plurality′-non-electing plurality′-sound
  unfolding defer-invariant-monotonicity-def
  by metis
```

end

5.13 Borda Module

theory *Borda-Module*
  imports *Component-Types/Elimination-Module*
begin

This is the Borda module used by the Borda rule. The Borda rule is a voting rule, where on each ballot, each alternative is assigned a score that depends on how many alternatives are ranked below. The sum of all such scores for an alternative is hence called their Borda score. The alternative with the highest Borda score is elected. The module implemented herein only rejects the alternatives not elected by the voting rule, and defers the alternatives that would be elected by the full voting rule.

5.13.1 Definition

fun *borda-score* :: ('*a*, '*v*) *Evaluation-Function* **where**
  borda-score V x A p = ($\sum$ *y* ∈ *A*. (*prefer-count V p x y*))

fun *borda* :: ('*a*, '*v*, '*a Result*) *Electoral-Module* **where**
  borda V A p = *max-eliminator borda-score V A p*

5.13.2 Soundness

theorem *borda-sound*: *𝒮𝒞ℱ-result.electoral-module borda*
  unfolding *borda.simps*
  using *max-elim-sound*
  by *metis*

5.13.3 Non-Blocking

The Borda module is non-blocking.

theorem *borda-mod-non-blocking*[*simp*]: *non-blocking borda*
  unfolding *borda.simps*
  using *max-elim-non-blocking*
  by *metis*

5.13.4 Non-Electing

The Borda module is non-electing.

theorem *borda-mod-non-electing*[*simp*]: *non-electing borda*

```
  using max-elim-non-electing
  unfolding borda.simps non-electing-def
  by metis

end
```

5.14 Condorcet Module

```
theory Condorcet-Module
  imports Component-Types/Elimination-Module
begin
```

This is the Condorcet module used by the Condorcet (voting) rule. The Condorcet rule is a voting rule that implements the Condorcet criterion, i.e., it elects the Condorcet winner if it exists, otherwise a tie remains between all alternatives. The module implemented herein only rejects the alternatives not elected by the voting rule, and defers the alternatives that would be elected by the full voting rule.

5.14.1 Definition

```
fun condorcet-score :: ('a, 'v) Evaluation-Function where
  condorcet-score V x A p =
    (if (condorcet-winner V A p x) then 1 else 0)

fun condorcet :: ('a, 'v, 'a Result) Electoral-Module where
  condorcet V A p = (max-eliminator condorcet-score) V A p
```

5.14.2 Soundness

```
theorem condorcet-sound: 𝒮𝒞ℱ-result.electoral-module condorcet
  unfolding condorcet.simps
  using max-elim-sound
  by metis
```

5.14.3 Property

```
theorem condorcet-score-is-condorcet-rating: condorcet-rating condorcet-score
proof (unfold condorcet-rating-def, safe)
  fix
    A :: 'b set and
    V :: 'a set and
    p :: ('b, 'a) Profile and
    w l :: 'b
  assume
```

```
    c-win: condorcet-winner V A p w and
    l-neq-w: l ≠ w
  have ¬ condorcet-winner V A p l
    using cond-winner-unique-eq c-win l-neq-w
    by metis
  thus condorcet-score V l A p < condorcet-score V w A p
    using c-win zero-less-one
    unfolding condorcet-score.simps
    by (metis (full-types))
qed

theorem condorcet-is-dcc: defer-condorcet-consistency condorcet
proof (unfold defer-condorcet-consistency-def 𝒮𝒞ℱ-result.electoral-module.simps,
        safe)
  fix
    A :: 'b set and
    V :: 'a set and
    p :: ('b, 'a) Profile
  assume profile V A p
  hence well-formed-𝒮𝒞ℱ A (max-eliminator condorcet-score V A p)
    using max-elim-sound
    unfolding 𝒮𝒞ℱ-result.electoral-module.simps
    by metis
  thus well-formed-𝒮𝒞ℱ A (condorcet V A p)
    by simp
next
  fix
    A :: 'b set and
    V :: 'a set and
    p :: ('b, 'a) Profile and
    a :: 'b
  assume c-win-w: condorcet-winner V A p a
  let ?m = (max-eliminator condorcet-score) :: ('b, 'a, 'b Result) Electoral-Module
  have defer-condorcet-consistency ?m
    using cr-eval-imp-dcc-max-elim condorcet-score-is-condorcet-rating
    by metis
  hence ?m V A p =
          ({}, A − defer ?m V A p, {b ∈ A. condorcet-winner V A p b})
    using c-win-w
    unfolding defer-condorcet-consistency-def
    by (metis (no-types))
  thus condorcet V A p =
          ({},
          A − defer condorcet V A p,
          {d ∈ A. condorcet-winner V A p d})
    by simp
qed

end
```

5.15 Copeland Module

```
theory Copeland-Module
  imports Component-Types/Elimination-Module
begin
```

This is the Copeland module used by the Copeland voting rule. The Copeland rule elects the alternatives with the highest difference between the amount of simple-majority wins and the amount of simple-majority losses. The module implemented herein only rejects the alternatives not elected by the voting rule, and defers the alternatives that would be elected by the full voting rule.

5.15.1 Definition

```
fun copeland-score :: ('a, 'v) Evaluation-Function where
  copeland-score V x A p =
    card {y ∈ A . wins V x p y} − card {y ∈ A . wins V y p x}

fun copeland :: ('a, 'v, 'a Result) Electoral-Module where
  copeland V A p = max-eliminator copeland-score V A p
```

5.15.2 Soundness

```
theorem copeland-sound: 𝒮𝒞ℱ-result.electoral-module copeland
  unfolding copeland.simps
  using max-elim-sound
  by metis
```

5.15.3 Lemmas

```
lemma voters-determine-copeland-score: voters-determine-evaluation copeland-score
proof (unfold copeland-score.simps voters-determine-evaluation.simps, safe)
  fix
    A :: 'b set and
    V :: 'a set and
    p p' :: ('b, 'a) Profile and
    a :: 'b
  assume
    ∀ v ∈ V. p v = p' v and
    a ∈ A
  hence ∀ x y. {v ∈ V. (x, y) ∈ p v} = {v ∈ V. (x, y) ∈ p' v}
    by blast
  hence ∀ x y.
```

```
    card {y ∈ A. wins V x p y} = card {y ∈ A. wins V x p' y}
    ∧ card {x ∈ A. wins V x p y} = card {x ∈ A. wins V x p' y}
    by simp
  thus card {y ∈ A. wins V a p y} − card {y ∈ A. wins V y p a} =
       card {y ∈ A. wins V a p' y} − card {y ∈ A. wins V y p' a}
    by presburger
qed

theorem voters-determine-copeland: voters-determine-election copeland
  unfolding copeland.simps
  using voters-determine-max-elim voters-determine-election.simps
        voters-determine-copeland-score
  by blast
```

For a Condorcet winner w, we have: "$|\{y \in A\ .\ wins\ V\ w\ p\ y\}| = |A| - 1$".

```
lemma cond-winner-imp-win-count:
  fixes
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    w :: 'a
  assumes condorcet-winner V A p w
  shows card {a ∈ A. wins V w p a} = card A − 1
proof −
  have ∀ a ∈ A − {w}. wins V w p a
    using assms
    by auto
  hence {a ∈ A − {w}. wins V w p a} = A − {w}
    by blast
  hence winner-wins-against-all-others:
    card {a ∈ A − {w}. wins V w p a} = card (A − {w})
    by simp
  have w ∈ A
    using assms
    by simp
  hence card (A − {w}) = card A − 1
    using card-Diff-singleton assms
    by metis
  hence winner-amount-one: card {a ∈ A − {w}. wins V w p a} = card (A) − 1
    using winner-wins-against-all-others
    by linarith
  have win-for-winner-not-reflexive: ∀ a ∈ {w}. ¬ wins V a p a
    by (simp add: wins-irreflex)
  hence {a ∈ {w}. wins V w p a} = {}
    by blast
  hence winner-amount-zero: card {a ∈ {w}. wins V w p a} = 0
    by simp
  have union:
    {a ∈ A − {w}. wins V w p a} ∪ {x ∈ {w}. wins V w p x} =
      {a ∈ A. wins V w p a}
    using win-for-winner-not-reflexive
    by blast
  have finite-defeated: finite {a ∈ A − {w}. wins V w p a}
    using assms
    by simp
  have finite {a ∈ {w}. wins V w p a}
    by simp
  hence card ({a ∈ A − {w}. wins V w p a} ∪ {a ∈ {w}. wins V w p a}) =
          card {a ∈ A − {w}. wins V w p a} + card {a ∈ {w}. wins V w p a}
    using finite-defeated card-Un-disjoint
    by blast
  hence card {a ∈ A. wins V w p a} =
          card {a ∈ A − {w}. wins V w p a} + card {a ∈ {w}. wins V w p a}
    using union
    by simp
  thus ?thesis
    using winner-amount-one winner-amount-zero
    by linarith
qed
```

For a Condorcet winner w, we have: "|{y ∈ A . wins V y p w}| = 0".

```
lemma cond-winner-imp-loss-count:
  fixes
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    w :: 'a
  assumes condorcet-winner V A p w
  shows card {a ∈ A. wins V a p w} = 0
  using Collect-empty-eq card-eq-0-iff insert-Diff insert-iff wins-antisym assms
  unfolding condorcet-winner.simps
  by (metis (no-types, lifting))
```

Copeland score of a Condorcet winner.

```
lemma cond-winner-imp-copeland-score:
  fixes
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    w :: 'a
  assumes condorcet-winner V A p w
  shows copeland-score V w A p = card A − 1
proof (unfold copeland-score.simps)
  have card {a ∈ A. wins V w p a} = card A − 1
    using cond-winner-imp-win-count assms
    by metis
  moreover have card {a ∈ A. wins V a p w} = 0
    using cond-winner-imp-loss-count assms
    by (metis (no-types))
  ultimately show
    enat (card {a ∈ A. wins V w p a}
      − card {a ∈ A. wins V a p w}) = enat (card A − 1)
    by simp
qed
```

For a non-Condorcet winner l, we have: "$|\{y \in A\ .\ \mathit{wins}\ V\ l\ p\ y\}| = |A| - 2$".

```
lemma non-cond-winner-imp-win-count:
  fixes
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    w l :: 'a
  assumes
    winner: condorcet-winner V A p w and
    loser: l ≠ w and
    l-in-A: l ∈ A
  shows card {a ∈ A . wins V l p a} ≤ card A − 2
proof −
  have wins V w p l
    using assms
    by auto
  hence ¬ wins V l p w
    using wins-antisym
    by simp
  moreover have ¬ wins V l p l
    using wins-irreflex
    by simp
  ultimately have wins-of-loser-eq-without-winner:
    {y ∈ A . wins V l p y} = {y ∈ A − {l, w} . wins V l p y}
    by blast
  have ∀ M f. finite M ⟶ card {x ∈ M . f x} ≤ card M
    by (simp add: card-mono)
  moreover have finite (A − {l, w})
    using finite-Diff winner
    by simp
  ultimately have card {y ∈ A − {l, w} . wins V l p y} ≤ card (A − {l, w})
    using winner
    by (metis (full-types))
  thus ?thesis
    using assms wins-of-loser-eq-without-winner
    by simp
qed
```

5.15.4 Property

The Copeland score is Condorcet rating.

```
theorem copeland-score-is-cr: condorcet-rating copeland-score
proof (unfold condorcet-rating-def, unfold copeland-score.simps, safe)
  fix
    A :: 'b set and
    V :: 'v set and
    p :: ('b, 'v) Profile and
    w l :: 'b
  assume
    winner: condorcet-winner V A p w and
    l-in-A: l ∈ A and
    l-neq-w: l ≠ w
  hence card {y ∈ A. wins V l p y} ≤ card A − 2
    using non-cond-winner-imp-win-count
    by (metis (mono-tags, lifting))
  hence card {y ∈ A. wins V l p y} − card {y ∈ A. wins V y p l} ≤ card A − 2
    using diff-le-self order.trans
    by simp
  moreover have card A − 2 < card A − 1
    using card-0-eq diff-less-mono2 empty-iff l-in-A l-neq-w neq0-conv less-one
          Suc-1 zero-less-diff add-diff-cancel-left' diff-is-0-eq Suc-eq-plus1
          card-1-singleton-iff order-less-le singletonD le-zero-eq winner
    unfolding condorcet-winner.simps
    by metis
  ultimately have
    card {y ∈ A. wins V l p y} − card {y ∈ A. wins V y p l} < card A − 1
    using order-le-less-trans
    by fastforce
  moreover have card {a ∈ A. wins V a p w} = 0
    using cond-winner-imp-loss-count winner
    by metis
  moreover have card A − 1 = card {a ∈ A. wins V w p a}
    using cond-winner-imp-win-count winner
    by (metis (full-types))
  ultimately show
    enat (card {y ∈ A. wins V l p y} − card {y ∈ A. wins V y p l}) <
      enat (card {y ∈ A. wins V w p y} − card {y ∈ A. wins V y p w})
    using enat-ord-simps diff-zero
    by (metis (no-types, lifting))
qed

theorem copeland-is-dcc: defer-condorcet-consistency copeland
proof (unfold defer-condorcet-consistency-def SCF-result.electoral-module.simps,
        safe)
  fix
    A :: 'b set and
    V :: 'a set and
    p :: ('b, 'a) Profile
  assume profile V A p
  moreover from this
  have well-formed-SCF A (max-eliminator copeland-score V A p)
    using max-elim-sound
    unfolding SCF-result.electoral-module.simps
    by metis
  ultimately show well-formed-SCF A (copeland V A p)
    using copeland-sound
    unfolding SCF-result.electoral-module.simps
    by metis
next
  fix
    A :: 'b set and
    V :: 'v set and
    p :: ('b, 'v) Profile and
    w :: 'b
  assume condorcet-winner V A p w
  moreover have defer-condorcet-consistency (max-eliminator copeland-score)
    by (simp add: copeland-score-is-cr)
  ultimately have
    max-eliminator copeland-score V A p =
      ({},
        A − defer (max-eliminator copeland-score) V A p,
        {d ∈ A. condorcet-winner V A p d})
    unfolding defer-condorcet-consistency-def
    by (metis (no-types))
  moreover have copeland V A p = max-eliminator copeland-score V A p
    unfolding copeland.simps
    by safe
  ultimately show
    copeland V A p =
      ({}, A − defer copeland V A p, {d ∈ A. condorcet-winner V A p d})
    by metis
qed

end
```

5.16 Minimax Module

```
theory Minimax-Module
  imports Component-Types/Elimination-Module
begin
```

This is the Minimax module used by the Minimax voting rule. The Minimax rule elects the alternatives with the highest Minimax score. The module implemented herein only rejects the alternatives not elected by the voting rule, and defers the alternatives that would be elected by the full voting rule.

5.16.1 Definition

```
fun minimax-score :: ('a, 'v) Evaluation-Function where
  minimax-score V x A p =
    Min {prefer-count V p x y | y . y ∈ A − {x}}

fun minimax :: ('a, 'v, 'a Result) Electoral-Module where
  minimax A p = max-eliminator minimax-score A p
```

5.16.2 Soundness

```
theorem minimax-sound: 𝒮𝒞ℱ-result.electoral-module minimax
  unfolding minimax.simps
  using max-elim-sound
  by metis
```

5.16.3 Lemma

```
lemma non-cond-winner-minimax-score:
  fixes
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    w l :: 'a
  assumes
    prof: profile V A p and
    winner: condorcet-winner V A p w and
    l-in-A: l ∈ A and
    l-neq-w: l ≠ w
  shows minimax-score V l A p ≤ prefer-count V p l w
proof (unfold minimax-score.simps, intro Min-le)
  have finite V
    using winner
    by simp
  moreover have ∀ E n. infinite E ⟶ (∃ e. ¬ e ≤ enat n ∧ e ∈ E)
    using finite-enat-bounded
    by blast
  ultimately show finite {prefer-count V p l y | y. y ∈ A − {l}}
    using pref-count-voter-set-card
    by fastforce
next
  have w ∈ A
    using winner
    by simp
  thus prefer-count V p l w ∈ {prefer-count V p l y | y. y ∈ A − {l}}
    using l-neq-w
    by blast
qed
```

5.16.4 Property

```
theorem minimax-score-cond-rating: condorcet-rating minimax-score
proof (unfold condorcet-rating-def minimax-score.simps prefer-count.simps,
      safe, rule ccontr)
  fix
    A :: 'b set and
    V :: 'a set and
    p :: ('b, 'a) Profile and
    w l :: 'b
  assume
    winner: condorcet-winner V A p w and
    l-in-A: l ∈ A and
    l-neq-w:l ≠ w and
    min-leq:
      ¬ Min {if finite V
            then enat (card {v ∈ V. let r = p v in y ⪯_r l})
            else ∞ | y. y ∈ A − {l}}
       < Min {if finite V
            then enat (card {v ∈ V. let r = p v in y ⪯_r w})
            else ∞ | y. y ∈ A − {w}}
  hence min-count-ineq:
    Min {prefer-count V p l y | y. y ∈ A − {l}} ≥
        Min {prefer-count V p w y | y. y ∈ A − {w}}
    by simp
  have pref-count-gte-min:
    prefer-count V p l w  ≥ Min {prefer-count V p l y | y . y ∈ A − {l}}
   using l-in-A l-neq-w condorcet-winner.simps winner non-cond-winner-minimax-score
        minimax-score.simps
    by metis
  have l-in-A-without-w: l ∈ A − {w}
    using l-in-A l-neq-w
    by simp
  hence pref-counts-non-empty: {prefer-count V p w y | y . y ∈ A − {w}} ≠ {}
    by blast
  have finite (A − {w})
    using condorcet-winner.simps winner finite-Diff
    by metis
  hence finite {prefer-count V p w y | y . y ∈ A − {w}}
    by simp
  hence ∃ n ∈ A − {w} . prefer-count V p w n =
            Min {prefer-count V p w y | y . y ∈ A − {w}}
    using pref-counts-non-empty Min-in
    by fastforce
  then obtain n :: 'b where
    pref-count-eq-min:
    prefer-count V p w n =
        Min {prefer-count V p w y | y . y ∈ A − {w}} and
    n-not-w: n ∈ A − {w}
    by metis
  hence n-in-A: n ∈ A
    using DiffE
    by metis
  have n-neq-w: n ≠ w
    using n-not-w
    by simp
  have w-in-A: w ∈ A
    using winner
    by simp
  have pref-count-n-w-ineq: prefer-count V p w n > prefer-count V p n w
    using n-not-w winner
    by auto
  have pref-count-l-w-n-ineq: prefer-count V p l w ≥ prefer-count V p w n
    using pref-count-gte-min min-count-ineq pref-count-eq-min
    by auto
  hence prefer-count V p n w ≥ prefer-count V p w l
    using n-in-A w-in-A l-in-A n-neq-w l-neq-w pref-count-sym winner
    unfolding condorcet-winner.simps
    by metis
  hence prefer-count V p l w > prefer-count V p w l
    using n-in-A w-in-A l-in-A n-neq-w l-neq-w pref-count-sym winner
          pref-count-n-w-ineq pref-count-l-w-n-ineq
    unfolding condorcet-winner.simps
    by auto
  hence wins V l p w
    by simp
  thus False
    using l-in-A-without-w wins-antisym winner
    unfolding condorcet-winner.simps
    by metis
qed

theorem minimax-is-dcc: defer-condorcet-consistency minimax
proof (unfold defer-condorcet-consistency-def 𝒮𝒞ℱ-result.electoral-module.simps,
        safe)
  fix
    A :: 'b set and
    V :: 'a set and
    p :: ('b, 'a) Profile
  assume profile V A p
  hence well-formed-𝒮𝒞ℱ A (max-eliminator minimax-score V A p)
    using max-elim-sound par-comp-result-sound
    by metis
  thus well-formed-𝒮𝒞ℱ A (minimax V A p)
    by simp
next
  fix
    A :: 'b set and
    V :: 'a set and
    p :: ('b, 'a) Profile and
    w :: 'b
  assume cwin-w: condorcet-winner V A p w
  have max-mmaxscore-dcc:
    defer-condorcet-consistency ((max-eliminator minimax-score)
                                  :: ('b, 'a, 'b Result) Electoral-Module)
    using cr-eval-imp-dcc-max-elim minimax-score-cond-rating
    by metis
  hence
    max-eliminator minimax-score V A p =
      ({},
       A − defer (max-eliminator minimax-score) V A p,
       {a ∈ A. condorcet-winner V A p a})
    using cwin-w
    unfolding defer-condorcet-consistency-def
    by blast
  thus
    minimax V A p =
      ({},
       A − defer minimax V A p,
       {d ∈ A. condorcet-winner V A p d})
    by simp
qed

end
```

Chapter 6

Compositional Structures

6.1 Drop- and Pass-Compatibility

```
theory Drop-And-Pass-Compatibility
  imports Basic-Modules/Drop-Module
      Basic-Modules/Pass-Module
begin
```

This is a collection of properties about the interplay and compatibility of both the drop module and the pass module.

```
theorem drop-zero-mod-rej-zero[simp]:
  fixes r :: 'a Preference-Relation
  assumes linear-order r
  shows rejects 0 (drop-module 0 r)
proof (unfold rejects-def, safe)
  show 𝒮𝒞ℱ-result.electoral-module (drop-module 0 r)
    using assms drop-mod-sound
    by metis
next
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assume
    fin-A: finite A and
    prof-A: profile V A p
  have connex UNIV r
    using assms lin-ord-imp-connex
    by auto
  hence connex: connex A (limit A r)
    using limit-presv-connex subset-UNIV
    by metis
  have ∀ B a. B ≠ {} ∨ (a :: 'a) ∉ B
    by simp
  hence ∀ a B. a ∈ A ∧ a ∈ B ⟶ connex B (limit A r) ⟶
```

```
      ¬ card (above (limit A r) a) ≤ 0
    using above-connex above-presv-limit card-eq-0-iff
          fin-A finite-subset le-0-eq assms
    by (metis (no-types))
  hence {a ∈ A. card (above (limit A r) a) ≤ 0} = {}
    using connex
    by auto
  hence card {a ∈ A. card (above (limit A r) a) ≤ 0} = 0
    using card.empty
    by (metis (full-types))
  thus card (reject (drop-module 0 r) V A p) = 0
    by simp
qed
```

The drop module rejects n alternatives (if there are at least n alternatives).

```
theorem drop-two-mod-rej-n[simp]:
  fixes r :: 'a Preference-Relation
  assumes linear-order r
  shows rejects n (drop-module n r)
proof (unfold rejects-def, safe)
  show 𝒮𝒞ℱ-result.electoral-module (drop-module n r)
    using drop-mod-sound
    by metis
next
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assume
    card-n: n ≤ card A and
    fin-A: finite A and
    prof: profile V A p
  let ?inv-rank = the-inv-into A (rank (limit A r))
  have lin-ord-limit: linear-order-on A (limit A r)
    using assms limit-presv-lin-ord
    by auto
  hence (limit A r) ⊆ A × A
    unfolding linear-order-on-def partial-order-on-def preorder-on-def refl-on-def
    by simp
  hence ∀ a ∈ A. (above (limit A r) a) ⊆ A
    unfolding above-def
    by auto
  hence leq: ∀ a ∈ A. rank (limit A r) a ≤ card A
    using fin-A
    by (simp add: card-mono)
  have ∀ a ∈ A. {a} ⊆ (above (limit A r) a)
    using lin-ord-limit
    unfolding linear-order-on-def partial-order-on-def
          preorder-on-def refl-on-def above-def
    by auto
  hence ∀ a ∈ A. card {a} ≤ card (above (limit A r) a)
    using card-mono fin-A rev-finite-subset above-presv-limit
    by metis
  hence rank-geq-one: ∀ a ∈ A. 1 ≤ rank (limit A r) a
    by simp
  with leq have ∀ a ∈ A. rank (limit A r) a ∈ {1 .. card A}
    by simp
  hence rank (limit A r) ‘ A ⊆ {1 .. card A}
    by auto
  moreover have inj: inj-on (rank (limit A r)) A
    using fin-A inj-onI rank-unique lin-ord-limit
    by metis
  ultimately have bij-A: bij-betw (rank (limit A r)) A {1 .. card A}
    using bij-betw-def bij-betw-finite bij-betw-iff-card card-seteq
          dual-order.refl ex-bij-betw-nat-finite-1 fin-A
    by metis
  hence bij-inv: bij-betw ?inv-rank {1 .. card A} A
    using bij-betw-the-inv-into
    by blast
  hence ∀ S ⊆ {1..card A}. card (?inv-rank ‘ S) = card S
    using fin-A bij-betw-same-card bij-betw-subset
    by metis
  moreover have subset: {1 .. n} ⊆ {1 .. card A}
    using card-n
    by simp
  ultimately have card (?inv-rank ‘ {1 .. n}) = n
    using numeral-One numeral-eq-iff card-atLeastAtMost diff-Suc-1
    by presburger
  also have ?inv-rank ‘ {1..n} = {a ∈ A. rank (limit A r) a ∈ {1 .. n}}
  proof
    show ?inv-rank ‘ {1..n} ⊆ {a ∈ A. rank (limit A r) a ∈ {1 .. n}}
    proof
      fix a :: 'a
      assume a ∈ ?inv-rank ‘ {1..n}
      then obtain b :: nat where
        b-img: b ∈ {1 .. n} ∧ ?inv-rank b = a
        by auto
      hence rank (limit A r) a = b
        using subset f-the-inv-into-f-bij-betw subsetD bij-A
        by metis
      hence rank (limit A r) a ∈ {1 .. n}
        using b-img
        by simp
      moreover have a ∈ A
        using b-img bij-inv bij-betwE subset
        by blast
      ultimately show a ∈ {a ∈ A. rank (limit A r) a ∈ {1 .. n}}
        by blast
      qed
    next
      show {a ∈ A. rank (limit A r) a ∈ {1 .. n}}
              ⊆ the-inv-into A (rank (limit A r)) ‘ {1 .. n}
      proof
        fix a :: 'a
        assume el: a ∈ {a ∈ A. rank (limit A r) a ∈ {1 .. n}}
        then obtain b :: nat where
          b-img: b ∈ {1..n} ∧ rank (limit A r) a = b
          by auto
        moreover have a ∈ A
          using el
          by simp
        ultimately have ?inv-rank b = a
          using inj the-inv-into-f-f
          by metis
        thus a ∈ ?inv-rank ‘ {1 .. n}
          using b-img
          by auto
      qed
    qed
    finally have card {a ∈ A. rank (limit A r) a ∈ {1 .. n}} = n
      by blast
    also have {a ∈ A. rank (limit A r) a ∈ {1 .. n}} =
                  {a ∈ A. rank (limit A r) a ≤ n}
      using rank-geq-one
      by auto
    also have . . . = reject (drop-module n r) V A p
      by simp
    finally show card (reject (drop-module n r) V A p) = n
      by blast
  qed
```

The pass and drop module are (disjoint-)compatible.

```
theorem drop-pass-disj-compat[simp]:
  fixes
    r :: 'a Preference-Relation and
    n :: nat
  assumes linear-order r
  shows disjoint-compatibility (drop-module n r) (pass-module n r)
proof (unfold disjoint-compatibility-def, safe)
  show 𝒮𝒞ℱ-result.electoral-module (drop-module n r)
    using assms drop-mod-sound
    by simp
next
  show 𝒮𝒞ℱ-result.electoral-module (pass-module n r)
    using assms pass-mod-sound
    by simp
next
```

```
  fix
    A :: 'a set and
    V :: 'b set
  have linear-order-on A (limit A r)
    using assms limit-presv-lin-ord
    by blast
  hence profile V A (λ v. limit A r)
    using profile-def
    by blast
  then obtain p :: ('a, 'b) Profile where
    profile V A p
    by blast
  show ∃ B ⊆ A. (∀ a ∈ B. indep-of-alt (drop-module n r) V A a ∧
                    (∀ p. profile V A p ⟶ a ∈ reject (drop-module n r) V A p)) ∧
             (∀ a ∈ A − B. indep-of-alt (pass-module n r) V A a ∧
                   (∀ p. profile V A p ⟶ a ∈ reject (pass-module n r) V A p))
  proof
    have same-A:
      ∀ p q. (profile V A p ∧ profile V A q) ⟶
        reject (drop-module n r) V A p = reject (drop-module n r) V A q
      by auto
    let ?A = reject (drop-module n r) V A p
    have ?A ⊆ A
      by auto
    moreover have ∀ a ∈ ?A. indep-of-alt (drop-module n r) V A a
      using assms drop-mod-sound
      unfolding drop-module.simps indep-of-alt-def
      by (metis (mono-tags, lifting))
    moreover have
      ∀ a ∈ ?A. ∀ p. profile V A p
          ⟶ a ∈ reject (drop-module n r) V A p
      by auto
    moreover have ∀ a ∈ A − ?A. indep-of-alt (pass-module n r) V A a
      using assms pass-mod-sound
      unfolding pass-module.simps indep-of-alt-def
      by metis
    moreover have
      ∀ a ∈ A − ?A. ∀ p.
        profile V A p ⟶ a ∈ reject (pass-module n r) V A p
      by auto
    ultimately show ?A ⊆ A ∧
        (∀ a ∈ ?A. indep-of-alt (drop-module n r) V A a ∧
          (∀ p. profile V A p ⟶ a ∈ reject (drop-module n r) V A p)) ∧
        (∀ a ∈ A − ?A. indep-of-alt (pass-module n r) V A a ∧
          (∀ p. profile V A p ⟶ a ∈ reject (pass-module n r) V A p))
      by simp
  qed
qed
```

```
end
```

6.2 Revision Composition

```
theory Revision-Composition
  imports Basic-Modules/Component-Types/Electoral-Module
begin
```

A revised electoral module rejects all originally rejected or deferred alternatives, and defers the originally elected alternatives. It does not elect any alternatives.

6.2.1 Definition

```
fun revision-composition :: ('a, 'v, 'a Result) Electoral-Module ⇒
        ('a, 'v, 'a Result) Electoral-Module where
  revision-composition m V A p = ({}, A − elect m V A p, elect m V A p)

abbreviation rev :: ('a, 'v, 'a Result) Electoral-Module ⇒
        ('a, 'v, 'a Result) Electoral-Module (-↓ 50) where
  m↓ ≡ revision-composition m
```

6.2.2 Soundness

```
theorem rev-comp-sound[simp]:
  fixes m :: ('a, 'v, 'a Result) Electoral-Module
  assumes 𝒮𝒞ℱ-result.electoral-module m
  shows 𝒮𝒞ℱ-result.electoral-module (revision-composition m)
proof −
  from assms
  have ∀ A V p. profile V A p ⟶ elect m V A p ⊆ A
    using elect-in-alts
    by metis
  hence ∀ A V p. profile V A p ⟶ (A − elect m V A p) ∪ elect m V A p = A
    by blast
  hence ∀ A V p. profile V A p ⟶
      set-equals-partition A (revision-composition m V A p)
    by simp
  moreover have ∀ A V p. profile V A p ⟶ (A − elect m V A p) ∩ elect m V
A p = {}
    by blast
  hence ∀ A V p. profile V A p ⟶ disjoint3 (revision-composition m V A p)
    by simp
  ultimately show ?thesis
    by simp
```

```
qed

lemma voters-determine-rev-comp:
  fixes m :: ('a, 'v, 'a Result) Electoral-Module
  assumes voters-determine-election m
  shows voters-determine-election (revision-composition m)
  using assms
  unfolding voters-determine-election.simps revision-composition.simps
  by presburger
```

6.2.3 Composition Rules

An electoral module received by revision is never electing.

```
theorem rev-comp-non-electing[simp]:
  fixes m :: ('a, 'v, 'a Result) Electoral-Module
  assumes 𝒮𝒞ℱ-result.electoral-module m
  shows non-electing (m↓)
  using assms fstI rev-comp-sound revision-composition.simps
  using non-electing-def
  by metis
```

Revising an electing electoral module results in a non-blocking electoral module.

```
theorem rev-comp-non-blocking[simp]:
  fixes m :: ('a, 'v, 'a Result) Electoral-Module
  assumes electing m
  shows non-blocking (m↓)
proof (unfold non-blocking-def, safe)
  show 𝒮𝒞ℱ-result.electoral-module (m↓)
    using assms rev-comp-sound
    unfolding electing-def
    by (metis (no-types, lifting))
next
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    x :: 'a
  assume
    fin-A: finite A and
    prof-A: profile V A p and
    reject-A: reject (m↓) V A p = A and
    x-in-A: x ∈ A
  hence non-electing m
    using assms empty-iff Diff-disjoint Int-absorb2
        elect-in-alts prod.collapse prod.inject
    unfolding electing-def revision-composition.simps
    by (metis (no-types, lifting))
  thus x ∈ {}
    using assms fin-A prof-A x-in-A
    unfolding electing-def non-electing-def
    by (metis (no-types, lifting))
qed
```

Revising an invariant monotone electoral module results in a defer-invariant-monotone electoral module.

```
theorem rev-comp-def-inv-mono[simp]:
  fixes m :: ('a, 'v, 'a Result) Electoral-Module
  assumes invariant-monotonicity m
  shows defer-invariant-monotonicity (m↓)
proof (unfold defer-invariant-monotonicity-def, safe)
  show 𝒮𝒞ℱ-result.electoral-module (m↓)
    using assms rev-comp-sound
    unfolding invariant-monotonicity-def
    by metis
next
  show non-electing (m↓)
    using assms rev-comp-non-electing
    unfolding invariant-monotonicity-def
    by simp
next
  fix
    A :: 'a set and
    V :: 'v set and
    p q :: ('a, 'v) Profile and
    a x x' :: 'a
  assume
    rev-p-defer-a: a ∈ defer (m↓) V A p and
    a-lifted: lifted V A p q a and
    rev-q-defer-x: x ∈ defer (m↓) V A q and
    x-non-eq-a: x ≠ a and
    rev-q-defer-x': x' ∈ defer (m↓) V A q
  from rev-p-defer-a
  have elect-a-in-p: a ∈ elect m V A p
    by simp
  from rev-q-defer-x x-non-eq-a
  have elect-no-unique-a-in-q: elect m V A q ≠ {a}
    by force
  from assms
  have elect m V A q = elect m V A p
    using a-lifted elect-a-in-p elect-no-unique-a-in-q
    unfolding invariant-monotonicity-def
    by (metis (no-types))
  thus x' ∈ defer (m↓) V A p
    using rev-q-defer-x'
    by simp
next
```

```
  fix
    A :: 'a set and
    V :: 'v set and
    p q :: ('a, 'v) Profile and
    a x x' :: 'a
  assume
    rev-p-defer-a: a ∈ defer (m↓) V A p and
    a-lifted: lifted V A p q a and
    rev-q-defer-x: x ∈ defer (m↓) V A q and
    x-non-eq-a: x ≠ a and
    rev-p-defer-x': x' ∈ defer (m↓) V A p
  have reject-and-defer:
    (A − elect m V A q, elect m V A q) = snd ((m↓) V A q)
    by force
  have elect-p-eq-defer-rev-p: elect m V A p = defer (m↓) V A p
    by simp
  hence elect-a-in-p: a ∈ elect m V A p
    using rev-p-defer-a
    by presburger
  have elect m V A q ≠ {a}
    using rev-q-defer-x x-non-eq-a
    by force
  with assms
  show x' ∈ defer (m↓) V A q
    using a-lifted rev-p-defer-x' snd-conv elect-a-in-p
          elect-p-eq-defer-rev-p reject-and-defer
    unfolding invariant-monotonicity-def
    by (metis (no-types))
next
  fix
    A :: 'a set and
    V :: 'v set and
    p q :: ('a, 'v) Profile and
    a x x' :: 'a
  assume
    a ∈ defer (m↓) V A p and
    lifted V A p q a and
    x' ∈ defer (m↓) V A q
  with assms
  show x' ∈ defer (m↓) V A p
    using empty-iff insertE snd-conv revision-composition.elims
    unfolding invariant-monotonicity-def
    by metis
next
  fix
    A :: 'a set and
    V :: 'v set and
    p q :: ('a, 'v) Profile and
    a x x' :: 'a
  assume
    rev-p-defer-a: a ∈ defer (m↓) V A p and
    a-lifted: lifted V A p q a and
    rev-q-not-defer-a: a ∉ defer (m↓) V A q
  moreover from assms
  have lifted-inv:
    ∀ A V p q a. a ∈ elect m V A p ∧ lifted V A p q a ⟶
      elect m V A q = elect m V A p ∨ elect m V A q = {a}
    unfolding invariant-monotonicity-def
    by (metis (no-types))
  moreover have p-defer-rev-eq-elect: defer (m↓) V A p = elect m V A p
    by simp
  moreover have defer (m↓) V A q = elect m V A q
    by simp
  ultimately show x' ∈ defer (m↓) V A q
    using rev-p-defer-a rev-q-not-defer-a
    by blast
qed

end
```

6.3 Sequential Composition

```
theory Sequential-Composition
  imports Basic-Modules/Component-Types/Electoral-Module
begin
```

The sequential composition creates a new electoral module from two electoral modules. In a sequential composition, the second electoral module makes decisions over alternatives deferred by the first electoral module.

6.3.1 Definition

```
fun sequential-composition :: ('a, 'v, 'a Result) Electoral-Module ⇒
        ('a, 'v, 'a Result) Electoral-Module ⇒
        ('a, 'v, 'a Result) Electoral-Module where
  sequential-composition m n V A p =
    (let new-A = defer m V A p;
        new-p = limit-profile new-A p in (
                  (elect m V A p) ∪ (elect n V new-A new-p),
                  (reject m V A p) ∪ (reject n V new-A new-p),
                  defer n V new-A new-p))

abbreviation sequence :: ('a, 'v, 'a Result) Electoral-Module ⇒
```

```
      ('a, 'v, 'a Result) Electoral-Module ⇒ ('a, 'v, 'a Result) Electoral-Module
      (infix ▷ 50) where
  m ▷ n ≡ sequential-composition m n

fun sequential-composition' :: ('a, 'v, 'a Result) Electoral-Module ⇒
      ('a, 'v, 'a Result) Electoral-Module ⇒
      ('a, 'v, 'a Result) Electoral-Module where
  sequential-composition' m n V A p =
    (let (m-e, m-r, m-d) = m V A p; new-A = m-d;
        new-p = limit-profile new-A p;
        (n-e, n-r, n-d) = n V new-A new-p in
           (m-e ∪ n-e, m-r ∪ n-r, n-d))

lemma voters-determine-seq-comp:
  fixes m n :: ('a, 'v, 'a Result) Electoral-Module
  assumes voters-determine-election m ∧ voters-determine-election n
  shows voters-determine-election (m ▷ n)
proof (unfold voters-determine-election.simps, clarify)
  fix
    A :: 'a set and
    V :: 'v set and
    p p' :: ('a, 'v) Profile
  assume coincide: ∀ v ∈ V. p v = p' v
  hence eq: m V A p = m V A p' ∧ n V A p = n V A p'
    using assms
    unfolding voters-determine-election.simps
    by blast
  hence coincide-limit:
    ∀ v ∈ V. limit-profile (defer m V A p) p v =
               limit-profile (defer m V A p') p' v
    using coincide
    by simp
  moreover have
    elect m V A p
      ∪ elect n V (defer m V A p) (limit-profile (defer m V A p) p) =
    elect m V A p'
      ∪ elect n V (defer m V A p') (limit-profile (defer m V A p') p')
    using assms eq coincide-limit
    unfolding voters-determine-election.simps
    by metis
  moreover have
    reject m V A p
      ∪ reject n V (defer m V A p) (limit-profile (defer m V A p) p) =
    reject m V A p'
      ∪ reject n V (defer m V A p') (limit-profile (defer m V A p') p')
    using assms eq coincide-limit
    unfolding voters-determine-election.simps
    by metis
  moreover have
```

```
    defer n V (defer m V A p) (limit-profile (defer m V A p) p) =
    defer n V (defer m V A p') (limit-profile (defer m V A p') p')
    using assms eq coincide-limit
    unfolding voters-determine-election.simps
    by metis
  ultimately show (m ▷ n) V A p = (m ▷ n) V A p'
    unfolding sequential-composition.simps
    by metis
qed

lemma seq-comp-presv-disj:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assumes
    module-m: 𝒮𝒞ℱ-result.electoral-module m and
    module-n: 𝒮𝒞ℱ-result.electoral-module n and
    prof: profile V A p
  shows disjoint3 ((m ▷ n) V A p)
proof -
  let ?new-A = defer m V A p
  let ?new-p = limit-profile ?new-A p
  have prof-def-lim: profile V (defer m V A p) (limit-profile (defer m V A p) p)
    using def-presv-prof prof module-m
    by metis
  have defer-in-A:
    ∀ A' V' p' m' a.
      (profile V' A' p' ∧
       𝒮𝒞ℱ-result.electoral-module m' ∧
       a ∈ defer m' V' A' p') ⟶
      a ∈ A'
    using UnCI result-presv-alts
    by (metis (mono-tags))
  from module-m prof
  have disjoint-m: disjoint3 (m V A p)
    unfolding 𝒮𝒞ℱ-result.electoral-module.simps well-formed-𝒮𝒞ℱ.simps
    by blast
  from module-m module-n def-presv-prof prof
  have disjoint-n: disjoint3 (n V ?new-A ?new-p)
    unfolding 𝒮𝒞ℱ-result.electoral-module.simps well-formed-𝒮𝒞ℱ.simps
    by metis
  have disj-n:
    elect m V A p ∩ reject m V A p = {} ∧
      elect m V A p ∩ defer m V A p = {} ∧
      reject m V A p ∩ defer m V A p = {}
    using prof module-m
    by (simp add: result-disj)
```

```
have reject n V (defer m V A p)
        (limit-profile (defer m V A p) p)
      ⊆ defer m V A p
  using def-presv-prof reject-in-alts prof module-m module-n
  by metis
with disjoint-m module-m module-n prof
have elect-reject-diff: elect m V A p ∩ reject n V ?new-A ?new-p = {}
  using disj-n
  by blast
from prof module-m module-n
have elec-n-in-def-m:
  elect n V (defer m V A p) (limit-profile (defer m V A p) p) ⊆ defer m V A p
  using def-presv-prof elect-in-alts
  by metis
have elect-defer-diff: elect m V A p ∩ defer n V ?new-A ?new-p = {}
proof −
  obtain f :: 'a set ⇒ 'a set ⇒ 'a where
    ∀ B B'.
      (∃ a b. a ∈ B' ∧ b ∈ B ∧ a = b) =
        (f B B' ∈ B' ∧ (∃ a. a ∈ B ∧ f B B' = a))
    using disjoint-iff
    by metis
  then obtain g :: 'a set ⇒ 'a set ⇒ 'a where
    ∀ B B'.
      (B ∩ B' = {}
        ⟶ (∀ a b. a ∈ B ∧ b ∈ B' ⟶ a ≠ b)) ∧
        (B ∩ B' ≠ {}
          ⟶ f B B' ∈ B ∧ g B B' ∈ B' ∧ f B B' = g B B')
    by auto
  thus ?thesis
    using defer-in-A disj-n module-n prof-def-lim prof
    by (metis (no-types, opaque-lifting))
qed
have rej-intersect-new-elect-empty:
  reject m V A p ∩ elect n V ?new-A ?new-p = {}
  using disj-n disjoint-m disjoint-n def-presv-prof prof
        module-m module-n elec-n-in-def-m
  by blast
have (elect m V A p ∪ elect n V ?new-A ?new-p) ∩
        (reject m V A p ∪ reject n V ?new-A ?new-p) = {}
proof (safe)
  fix x :: 'a
  assume
    x ∈ elect m V A p and
    x ∈ reject m V A p
  hence x ∈ elect m V A p ∩ reject m V A p
    by simp
  thus x ∈ {}
    using disj-n
    by simp
next
  fix x :: 'a
  assume
    x ∈ elect m V A p and
    x ∈ reject n V (defer m V A p)
      (limit-profile (defer m V A p) p)
  thus x ∈ {}
    using elect-reject-diff
    by blast
next
  fix x :: 'a
  assume
    x ∈ elect n V (defer m V A p)
            (limit-profile (defer m V A p) p) and
    x ∈ reject m V A p
  thus x ∈ {}
    using rej-intersect-new-elect-empty
    by blast
next
  fix x :: 'a
  assume
    x ∈ elect n V (defer m V A p)
          (limit-profile (defer m V A p) p) and
    x ∈ reject n V (defer m V A p)
        (limit-profile (defer m V A p) p)
  thus x ∈ {}
    using disjoint-iff-not-equal module-n prof-def-lim result-disj prof
    by metis
qed
moreover have
  (elect m V A p ∪ elect n V ?new-A ?new-p)
    ∩ (defer n V ?new-A ?new-p) = {}
  using Int-Un-distrib2 Un-empty elect-defer-diff module-n
        prof-def-lim result-disj prof
  by (metis (no-types))
moreover have
  (reject m V A p ∪ reject n V ?new-A ?new-p)
    ∩ (defer n V ?new-A ?new-p) = {}
proof (safe)
  fix x :: 'a
  assume x ∈ defer n V (defer m V A p) (limit-profile (defer m V A p) p)
  hence x ∈ defer m V A p
    using defer-in-A module-n prof-def-lim prof
    by metis
  moreover assume x ∈ reject m V A p
  ultimately have x ∈ reject m V A p ∩ defer m V A p
    by fastforce
  thus x ∈ {}
```

```
      using disj-n
      by blast
  next
    fix x :: 'a
    assume
      x ∈ defer n V (defer m V A p) (limit-profile (defer m V A p) p) and
      x ∈ reject n V (defer m V A p) (limit-profile (defer m V A p) p)
    thus x ∈ {}
      using module-n prof-def-lim reject-not-elected-or-deferred
      by blast
  qed
  ultimately have
    disjoint3 (elect m V A p ∪ elect n V ?new-A ?new-p,
               reject m V A p ∪ reject n V ?new-A ?new-p,
               defer n V ?new-A ?new-p)
    by simp
  thus ?thesis
    unfolding sequential-composition.simps
    by metis
qed

lemma seq-comp-presv-alts:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assumes
    module-m: 𝒮𝒞ℱ-result.electoral-module m and
    module-n: 𝒮𝒞ℱ-result.electoral-module n and
    prof: profile V A p
  shows set-equals-partition A ((m ▷ n) V A p)
proof −
  let ?new-A = defer m V A p
  let ?new-p = limit-profile ?new-A p
  have elect-reject-diff: elect m V A p ∪ reject m V A p ∪ ?new-A = A
    using module-m prof
    by (simp add: result-presv-alts)
  have elect n V ?new-A ?new-p ∪
         reject n V ?new-A ?new-p ∪
           defer n V ?new-A ?new-p = ?new-A
    using module-m module-n prof def-presv-prof result-presv-alts
    by metis
  hence (elect m V A p ∪ elect n V ?new-A ?new-p) ∪
         (reject m V A p ∪ reject n V ?new-A ?new-p) ∪
           defer n V ?new-A ?new-p = A
    using elect-reject-diff
    by blast
  hence set-equals-partition A
```

```
        (elect m V A p ∪ elect n V ?new-A ?new-p,
          reject m V A p ∪ reject n V ?new-A ?new-p,
            defer n V ?new-A ?new-p)
    by simp
  thus ?thesis
    unfolding sequential-composition.simps
    by metis
qed

lemma seq-comp-alt-eq[fundef-cong, code]: sequential-composition = sequential-composition′
proof (unfold sequential-composition′.simps sequential-composition.simps)
  have ∀ m n V A E.
      (case m V A E of (e, r, d) ⇒
        case n V d (limit-profile d E) of (e′, r′, d′) ⇒
        (e ∪ e′, r ∪ r′, d′)) =
          (elect m V A E
            ∪ elect n V (defer m V A E) (limit-profile (defer m V A E) E),
            reject m V A E
            ∪ reject n V (defer m V A E) (limit-profile (defer m V A E) E),
            defer n V (defer m V A E) (limit-profile (defer m V A E) E))
    using case-prod-beta′
    by (metis (no-types, lifting))
  thus
    (λ m n V A p.
        let A′ = defer m V A p; p′ = limit-profile A′ p in
      (elect m V A p ∪ elect n V A′ p′,
        reject m V A p ∪ reject n V A′ p′,
        defer n V A′ p′)) =
      (λ m n V A pr.
        let (e, r, d) = m V A pr; A′ = d; p′ = limit-profile A′ pr;
          (e′, r′, d′) = n V A′ p′ in
      (e ∪ e′, r ∪ r′, d′))
    by metis
qed
```

6.3.2 Soundness

```
theorem seq-comp-sound[simp]:
  fixes m n :: (′a, ′v, ′a Result) Electoral-Module
  assumes
    𝒮𝒞ℱ-result.electoral-module m and
    𝒮𝒞ℱ-result.electoral-module n
  shows 𝒮𝒞ℱ-result.electoral-module (m ▷ n)
proof (unfold 𝒮𝒞ℱ-result.electoral-module.simps, safe)
  fix
    A :: ′a set and
    V :: ′v set and
    p :: (′a, ′v) Profile
  assume profile V A p
  moreover have ∀ r. well-formed-SCF (A :: 'a set) r =
        (disjoint3 r ∧ set-equals-partition A r)
    by simp
  ultimately show well-formed-SCF A ((m ⊳ n) V A p)
    using assms seq-comp-presv-disj seq-comp-presv-alts
    by metis
qed
```

6.3.3 Lemmas

```
lemma seq-comp-decrease-only-defer:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assumes
    module-m: SCF-result.electoral-module m and
    module-n: SCF-result.electoral-module n and
    prof: profile V A p and
    empty-defer: defer m V A p = {}
  shows (m ⊳ n) V A p = m V A p
proof −
  have ∀ m' A' V' p'.
      (SCF-result.electoral-module m' ∧ profile V' A' p') ⟶
        profile V' (defer m' V' A' p') (limit-profile (defer m' V' A' p') p')
    using def-presv-prof prof
    by metis
  hence prof-no-alt: profile V {} (limit-profile (defer m V A p) p)
    using empty-defer prof module-m
    by metis
  show ?thesis
  proof
    have (elect m V A p)
      ∪ (elect n V (defer m V A p) (limit-profile (defer m V A p) p)) =
          elect m V A p
      using elect-in-alts[of - - - limit-profile (defer m V A p) p]
            empty-defer module-n prof prof-no-alt
      by auto
    thus elect (m ⊳ n) V A p = elect m V A p
      using fst-conv
      unfolding sequential-composition.simps
      by metis
  next
    have rej-empty:
      ∀ m' V' p'.
        (SCF-result.electoral-module m'
          ∧ profile V' {} p') ⟶ reject m' V' {} p' = {}
      using bot.extremum-uniqueI reject-in-alts
      by metis
    have (reject m V A p, defer n V {} (limit-profile {} p)) = snd (m V A p)
      using bot.extremum-uniqueI defer-in-alts empty-defer
            module-n prod.collapse prof-no-alt
      by (metis (no-types))
    thus snd ((m ▷ n) V A p) = snd (m V A p)
      unfolding sequential-composition.simps
      using rej-empty empty-defer module-n prof-no-alt prof sndI sup-bot-right
      by metis
  qed
qed

lemma seq-comp-def-then-elect:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assumes
    n-electing-m: non-electing m and
    def-one-m: defers 1 m and
    electing-n: electing n and
    f-prof: finite-profile V A p
  shows elect (m ▷ n) V A p = defer m V A p
proof (cases)
  assume A = {}
  with electing-n n-electing-m f-prof
  show ?thesis
    using bot.extremum-uniqueI defer-in-alts elect-in-alts seq-comp-sound
    unfolding electing-def non-electing-def
    by metis
next
  assume non-empty-A: A ≠ {}
  from n-electing-m f-prof
  have ele: elect m V A p = {}
    unfolding non-electing-def
    by simp
  from non-empty-A def-one-m f-prof finite
  have def-card: card (defer m V A p) = 1
    unfolding defers-def
    by (simp add: Suc-leI card-gt-0-iff)
  with n-electing-m f-prof
  have def: ∃ a ∈ A. defer m V A p = {a}
    using card-1-singletonE defer-in-alts singletonI subsetCE
    unfolding non-electing-def
    by metis
  from ele def n-electing-m
  have rej: ∃ a ∈ A. reject m V A p = A − {a}
    using Diff-empty def-one-m f-prof reject-not-elected-or-deferred
    unfolding defers-def
    by metis
  from ele rej def n-electing-m f-prof
  have res-m: ∃ a ∈ A. m V A p = ({}, A − {a}, {a})
    using Diff-empty elect-rej-def-combination reject-not-elected-or-deferred
    unfolding non-electing-def
    by metis
  hence ∃ a ∈ A. elect (m ▷ n) V A p = elect n V {a} (limit-profile {a} p)
    using prod.sel sup-bot.left-neutral
    unfolding sequential-composition.simps
    by metis
  with def-card def electing-n n-electing-m f-prof
  have ∃ a ∈ A. elect (m ▷ n) V A p = {a}
    using electing-for-only-alt fst-conv def-presv-prof sup-bot.left-neutral
    unfolding non-electing-def sequential-composition.simps
    by metis
  with def def-card electing-n n-electing-m f-prof res-m
  show ?thesis
    using def-presv-prof electing-for-only-alt fst-conv sup-bot.left-neutral
    unfolding non-electing-def sequential-composition.simps
    by metis
qed

lemma seq-comp-def-card-bounded:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assumes
    𝒮𝒞ℱ-result.electoral-module m and
    𝒮𝒞ℱ-result.electoral-module n and
    finite-profile V A p
  shows card (defer (m ▷ n) V A p) ≤ card (defer m V A p)
  using card-mono defer-in-alts assms def-presv-prof snd-conv finite-subset
  unfolding sequential-composition.simps
  by metis

lemma seq-comp-def-set-bounded:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assumes
    𝒮𝒞ℱ-result.electoral-module m and
    𝒮𝒞ℱ-result.electoral-module n and
    profile V A p
  shows defer (m ▷ n) V A p ⊆ defer m V A p
  using defer-in-alts assms snd-conv def-presv-prof
  unfolding sequential-composition.simps
  by metis

lemma seq-comp-defers-def-set:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  shows defer (m ▷ n) V A p =
          defer n V (defer m V A p) (limit-profile (defer m V A p) p)
  using snd-conv
  unfolding sequential-composition.simps
  by metis

lemma seq-comp-def-then-elect-elec-set:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  shows elect (m ▷ n) V A p =
            elect n V (defer m V A p)
              (limit-profile (defer m V A p) p) ∪ (elect m V A p)
  using Un-commute fst-conv
  unfolding sequential-composition.simps
  by metis

lemma seq-comp-elim-one-red-def-set:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assumes
    SCF-result.electoral-module m and
    eliminates 1 n and
    profile V A p and
    card (defer m V A p) > 1
  shows defer (m ▷ n) V A p ⊂ defer m V A p
  using assms snd-conv def-presv-prof single-elim-imp-red-def-set
  unfolding sequential-composition.simps
  by metis

lemma seq-comp-def-set-trans:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assumes
    a ∈ (defer (m ▷ n) V A p) and
    𝒮𝒞ℱ-result.electoral-module m ∧ 𝒮𝒞ℱ-result.electoral-module n and
    profile V A p
  shows a ∈ defer n V (defer m V A p) (limit-profile (defer m V A p) p) ∧
        a ∈ defer m V A p
  using seq-comp-def-set-bounded assms in-mono seq-comp-defers-def-set
  by (metis (no-types, opaque-lifting))
```

6.3.4 Composition Rules

The sequential composition preserves the non-blocking property.

```
theorem seq-comp-presv-non-blocking[simp]:
  fixes m n :: ('a, 'v, 'a Result) Electoral-Module
  assumes
    non-blocking-m: non-blocking m and
    non-blocking-n: non-blocking n
  shows non-blocking (m ▷ n)
proof −
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  let ?input-sound = A ≠ {} ∧ finite-profile V A p
  from non-blocking-m
  have ?input-sound ⟶ reject m V A p ≠ A
    unfolding non-blocking-def
    by simp
  with non-blocking-m
  have A-reject-diff: ?input-sound ⟶ A − reject m V A p ≠ {}
    using Diff-eq-empty-iff reject-in-alts subset-antisym
    unfolding non-blocking-def
    by metis
  from non-blocking-m
  have ?input-sound ⟶ well-formed-𝒮𝒞ℱ A (m V A p)
    unfolding 𝒮𝒞ℱ-result.electoral-module.simps non-blocking-def
    by simp
  hence ?input-sound ⟶ elect m V A p ∪ defer m V A p = A − reject m V A p
    using non-blocking-m elec-and-def-not-rej
    unfolding non-blocking-def
    by metis
  with A-reject-diff
  have ?input-sound ⟶ elect m V A p ∪ defer m V A p ≠ {}
    by simp
  hence ?input-sound ⟶ (elect m V A p ≠ {} ∨ defer m V A p ≠ {})
    by simp
```

```
with non-blocking-m non-blocking-n
show ?thesis
proof (unfold non-blocking-def)
  assume
    emod-reject-m:
    𝒮𝒞ℱ-result.electoral-module m
    ∧ (∀ A V p. A ≠ {} ∧ finite A ∧ profile V A p
        ⟶ reject m V A p ≠ A) and
    emod-reject-n:
    𝒮𝒞ℱ-result.electoral-module n
    ∧ (∀ A V p. A ≠ {} ∧ finite A ∧ profile V A p
        ⟶ reject n V A p ≠ A)
  show
    𝒮𝒞ℱ-result.electoral-module (m ▷ n)
    ∧ (∀ A V p. A ≠ {} ∧ finite A ∧ profile V A p
        ⟶ reject (m ▷ n) V A p ≠ A)
  proof (safe)
    show 𝒮𝒞ℱ-result.electoral-module (m ▷ n)
      using emod-reject-m emod-reject-n seq-comp-sound
      by metis
  next
    fix
      A :: 'a set and
      V :: 'v set and
      p :: ('a, 'v) Profile and
      x :: 'a
    assume
      fin-A: finite A and
      prof-A: profile V A p and
      rej-mn: reject (m ▷ n) V A p = A and
      x-in-A: x ∈ A
    from emod-reject-m fin-A prof-A
    have fin-defer:
      finite (defer m V A p)
      ∧ profile V (defer m V A p) (limit-profile (defer m V A p) p)
      using def-presv-prof defer-in-alts finite-subset
      by (metis (no-types))
    from emod-reject-m emod-reject-n fin-A prof-A
    have seq-elect:
      elect (m ▷ n) V A p =
        elect n V (defer m V A p)
          (limit-profile (defer m V A p) p) ∪ elect m V A p
      using seq-comp-def-then-elect-elec-set
      by metis
    from emod-reject-n emod-reject-m fin-A prof-A
    have def-limit:
      defer (m ▷ n) V A p =
        defer n V (defer m V A p) (limit-profile (defer m V A p) p)
      using seq-comp-defers-def-set
```

```
        by metis
      from emod-reject-n emod-reject-m fin-A prof-A
      have elect (m ▷ n) V A p ∪ defer (m ▷ n) V A p =
              A − reject (m ▷ n) V A p
        using elec-and-def-not-rej seq-comp-sound
        by metis
      hence elect-def-disj:
        elect n V (defer m V A p) (limit-profile (defer m V A p) p) ∪
          elect m V A p ∪
          defer n V (defer m V A p) (limit-profile (defer m V A p) p) = {}
        using def-limit seq-elect Diff-cancel rej-mn
        by auto
      have rej-def-eq-set:
        defer n V (defer m V A p) (limit-profile (defer m V A p) p) −
          defer n V (defer m V A p) (limit-profile (defer m V A p) p) = {} ⟶
            reject n V (defer m V A p) (limit-profile (defer m V A p) p) =
              defer m V A p
        using elect-def-disj emod-reject-n fin-defer
        by (simp add: reject-not-elected-or-deferred)
      have
        defer n V (defer m V A p) (limit-profile (defer m V A p) p) −
          defer n V (defer m V A p) (limit-profile (defer m V A p) p) = {} ⟶
            elect m V A p = elect m V A p ∩ defer m V A p
        using elect-def-disj
        by blast
      thus x ∈ {}
        using rej-def-eq-set result-disj fin-defer Diff-cancel Diff-empty fin-A prof-A
              emod-reject-m emod-reject-n reject-not-elected-or-deferred x-in-A
        by metis
    qed
  qed
qed
```

Sequential composition preserves the non-electing property.

```
theorem seq-comp-presv-non-electing[simp]:
  fixes m n :: ('a, 'v, 'a Result) Electoral-Module
  assumes
    non-electing m and
    non-electing n
  shows non-electing (m ▷ n)
proof (unfold non-electing-def, safe)
  have 𝒮𝒞ℱ-result.electoral-module m ∧ 𝒮𝒞ℱ-result.electoral-module n
    using assms
    unfolding non-electing-def
    by blast
  thus 𝒮𝒞ℱ-result.electoral-module (m ▷ n)
    using seq-comp-sound
    by metis
next
```

```
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    x :: 'a
  assume
    profile V A p and
    x ∈ elect (m ▷ n) V A p
  thus x ∈ {}
    using assms
    unfolding non-electing-def
    using seq-comp-def-then-elect-elec-set def-presv-prof Diff-empty Diff-partition
          empty-subsetI
    by metis
qed
```

Composing an electoral module that defers exactly 1 alternative in sequence after an electoral module that is electing results (still) in an electing electoral module.

```
theorem seq-comp-electing[simp]:
  fixes m n :: ('a, 'v, 'a Result) Electoral-Module
  assumes
    def-one-m:  defers 1 m and
    electing-n: electing n
  shows electing (m ▷ n)
proof −
  have defer-card-eq-one:
    ∀ A V p. (card A ≥ 1 ∧ finite A ∧ profile V A p)
          ⟶ card (defer m V A p) = 1
    using def-one-m
    unfolding defers-def
    by metis
  hence def-m-not-empty:
    ∀ A V p. (A ≠ {} ∧ finite A ∧ profile V A p) ⟶ defer m V A p ≠ {}
    using One-nat-def Suc-leI card-eq-0-iff card-gt-0-iff zero-neq-one
    by metis
  thus ?thesis
  proof −
    have ∀ m'.
          (¬ electing m' ∨ SCF-result.electoral-module m'
          ∧ (∀ A' V' p'. (A' ≠ {} ∧ finite A' ∧ profile V' A' p')
              ⟶ elect m' V' A' p' ≠ {}))
        ∧ (electing m' ∨ ¬ SCF-result.electoral-module m' ∨
            (∃ A V p. (A ≠ {} ∧ finite A ∧ profile V A p ∧ elect m' V A p = {})))
      unfolding electing-def
      by blast
    hence ∀ m'.
          (¬ electing m' ∨ SCF-result.electoral-module m'
          ∧ (∀ A' V' p'. (A' ≠ {} ∧ finite A' ∧ profile V' A' p')
              ⟶ elect m′ V′ A′ p′ ≠ {}))
        ∧ (∃ A V p. (electing m′ ∨ ¬ 𝒮𝒞ℱ-result.electoral-module m′ ∨ A ≠ {}
            ∧ finite A ∧ profile V A p ∧ elect m′ V A p = {}))
      by simp
    then obtain
      A :: (′a, ′v, ′a Result) Electoral-Module ⇒ ′a set and
      V :: (′a, ′v, ′a Result) Electoral-Module ⇒ ′v set and
      p :: (′a, ′v, ′a Result) Electoral-Module ⇒ (′a, ′v) Profile where
      f-mod:
       ∀ m′ :: (′a, ′v, ′a Result) Electoral-Module.
        (¬ electing m′ ∨ 𝒮𝒞ℱ-result.electoral-module m′ ∧
          (∀ A′ V′ p′. (A′ ≠ {} ∧ finite A′ ∧ profile V′ A′ p′)
            ⟶ elect m′ V′ A′ p′ ≠ {}))
        ∧ (electing m′ ∨ ¬ 𝒮𝒞ℱ-result.electoral-module m′ ∨ A m′ ≠ {}
        ∧ finite (A m′) ∧ profile (V m′) (A m′) (p m′)
        ∧ elect m′ (V m′) (A m′) (p m′) = {})
      by metis
    hence f-elect:
      𝒮𝒞ℱ-result.electoral-module n ∧
        (∀ A V p. (A ≠ {} ∧ finite A ∧ profile V A p) ⟶ elect n V A p ≠ {})
      using electing-n
      unfolding electing-def
      by metis
    have def-card-one:
      𝒮𝒞ℱ-result.electoral-module m
      ∧ (∀ A V p. (1 ≤ card A ∧ finite A ∧ profile V A p)
          ⟶ card (defer m V A p) = 1)
      using def-one-m defer-card-eq-one
      unfolding defers-def
      by blast
    hence 𝒮𝒞ℱ-result.electoral-module (m ⊳ n)
      using f-elect seq-comp-sound
      by metis
    with f-mod f-elect def-card-one
    show ?thesis
      using seq-comp-def-then-elect-elec-set def-presv-prof defer-in-alts
            def-m-not-empty bot-eq-sup-iff finite-subset
      unfolding electing-def
      by metis
  qed
qed

lemma def-lift-inv-seq-comp-help:
  fixes
    m n :: (′a, ′v, ′a Result) Electoral-Module and
    A :: ′a set and
    V :: ′v set and
    p q :: (′a, ′v) Profile and
    a :: ′a
  assumes
    monotone-m: defer-lift-invariance m and
    monotone-n: defer-lift-invariance n and
    voters-determine-n: voters-determine-election n and
    def-and-lifted: a ∈ (defer (m ▷ n) V A p) ∧ lifted V A p q a
  shows (m ▷ n) V A p = (m ▷ n) V A q
proof –
  let ?new-Ap = defer m V A p
  let ?new-Aq = defer m V A q
  let ?new-p = limit-profile ?new-Ap p
  let ?new-q = limit-profile ?new-Aq q
  from monotone-m monotone-n
  have modules: 𝒮𝒞ℱ-result.electoral-module m ∧ 𝒮𝒞ℱ-result.electoral-module n
    unfolding defer-lift-invariance-def
    by simp
  hence profile V A p ⟶ defer (m ▷ n) V A p ⊆ defer m V A p
    using seq-comp-def-set-bounded
    by metis
  moreover have profile-p: lifted V A p q a ⟶ finite-profile V A p
    unfolding lifted-def
    by simp
  ultimately have defer-subset: defer (m ▷ n) V A p ⊆ defer m V A p
    using def-and-lifted
    by blast
  hence mono-m: m V A p = m V A q
    using monotone-m def-and-lifted modules profile-p
          seq-comp-def-set-trans
    unfolding defer-lift-invariance-def
    by metis
  hence new-A-eq: ?new-Ap = ?new-Aq
    by presburger
  have defer-eq: defer (m ▷ n) V A p = defer n V ?new-Ap ?new-p
    using snd-conv
    unfolding sequential-composition.simps
    by metis
  have mono-n: n V ?new-Ap ?new-p = n V ?new-Aq ?new-q
  proof (cases)
    assume lifted V ?new-Ap ?new-p ?new-q a
    thus ?thesis
      using defer-eq mono-m monotone-n def-and-lifted
      unfolding defer-lift-invariance-def
      by (metis (no-types, lifting))
  next
    assume unlifted-a: ¬lifted V ?new-Ap ?new-p ?new-q a
    from def-and-lifted
    have finite-profile V A q
      unfolding lifted-def
      by simp
    with modules new-A-eq
    have prof-p: profile V ?new-Ap ?new-q
      using def-presv-prof
      by (metis (no-types))
    moreover from modules profile-p def-and-lifted
    have prof-q: profile V ?new-Ap ?new-p
      using def-presv-prof
      by (metis (no-types))
    moreover from defer-subset def-and-lifted
    have a ∈ ?new-Ap
      by blast
    ultimately have lifted-stmt:
      (∃ v ∈ V.
          Preference-Relation.lifted ?new-Ap (?new-p v) (?new-q v) a) ⟶
       (∃ v ∈ V.
         ¬ Preference-Relation.lifted ?new-Ap (?new-p v) (?new-q v) a ∧
             (?new-p v) ≠ (?new-q v))
      using unlifted-a def-and-lifted defer-in-alts infinite-super modules profile-p
      unfolding lifted-def
      by metis
    from def-and-lifted modules
    have ∀ v ∈ V. (Preference-Relation.lifted A (p v) (q v) a ∨ (p v) = (q v))
      unfolding Profile.lifted-def
      by metis
    with def-and-lifted modules mono-m
    have ∀ v ∈ V.
            (Preference-Relation.lifted ?new-Ap (?new-p v) (?new-q v) a ∨
              (?new-p v) = (?new-q v))
      using limit-lifted-imp-eq-or-lifted defer-in-alts
      unfolding Profile.lifted-def limit-profile.simps
      by (metis (no-types, lifting))
    with lifted-stmt
    have ∀ v ∈ V. (?new-p v) = (?new-q v)
      by blast
    with mono-m
    show ?thesis
      using leI not-less-zero nth-equalityI voters-determine-n
      unfolding voters-determine-election.simps
      by presburger
  qed
  from mono-m mono-n
  show ?thesis
    unfolding sequential-composition.simps
    by (metis (full-types))
qed
```

Sequential composition preserves the property defer-lift-invariance.

```
theorem seq-comp-presv-def-lift-inv[simp]:
  fixes m n :: ('a, 'v, 'a Result) Electoral-Module
  assumes
```

```
    defer-lift-invariance m and
    defer-lift-invariance n and
    voters-determine-election n
  shows defer-lift-invariance (m ▷ n)
proof (unfold defer-lift-invariance-def, safe)
  show 𝒮𝒞ℱ-result.electoral-module (m ▷ n)
    using assms seq-comp-sound
    unfolding defer-lift-invariance-def
    by blast
next
  fix
    A :: 'a set and
    V :: 'v set and
    p q :: ('a, 'v) Profile and
    a :: 'a
  assume
    a ∈ defer (m ▷ n) V A p and
    Profile.lifted V A p q a
  thus (m ▷ n) V A p = (m ▷ n) V A q
    unfolding defer-lift-invariance-def
    using assms def-lift-inv-seq-comp-help
    by metis
qed
```

Composing a non-blocking, non-electing electoral module in sequence with an electoral module that defers exactly one alternative results in an electoral module that defers exactly one alternative.

```
theorem seq-comp-def-one[simp]:
  fixes m n :: ('a, 'v, 'a Result) Electoral-Module
  assumes
    non-blocking-m: non-blocking m and
    non-electing-m: non-electing m and
    def-one-n: defers 1 n
  shows defers 1 (m ▷ n)
proof (unfold defers-def, safe)
  have 𝒮𝒞ℱ-result.electoral-module m
    using non-electing-m
    unfolding non-electing-def
    by simp
  moreover have 𝒮𝒞ℱ-result.electoral-module n
    using def-one-n
    unfolding defers-def
    by simp
  ultimately show 𝒮𝒞ℱ-result.electoral-module (m ▷ n)
    using seq-comp-sound
    by metis
next
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assume
    pos-card: 1 ≤ card A and
    fin-A: finite A and
    prof-A: profile V A p
  from pos-card
  have A ≠ {}
    by auto
  with fin-A prof-A
  have reject m V A p ≠ A
    using non-blocking-m
    unfolding non-blocking-def
    by simp
  hence ∃ a. a ∈ A ∧ a ∉ reject m V A p
    using non-electing-m reject-in-alts fin-A prof-A
          card-seteq infinite-super subsetI upper-card-bound-for-reject
    unfolding non-electing-def
    by metis
  hence defer m V A p ≠ {}
    using electoral-mod-defer-elem empty-iff non-electing-m fin-A prof-A
    unfolding non-electing-def
    by (metis (no-types))
  hence card (defer m V A p) ≥ 1
    using Suc-leI card-gt-0-iff fin-A prof-A
          non-blocking-m defer-in-alts infinite-super
    unfolding One-nat-def non-blocking-def
    by metis
  moreover have
    ∀ i m'. defers i m' =
      (SCF-result.electoral-module m' ∧
        (∀ A' V' p'. (i ≤ card A' ∧ finite A' ∧ profile V' A' p') ⟶
            card (defer m' V' A' p') = i))
    unfolding defers-def
    by simp
  ultimately have
    card (defer n V (defer m V A p) (limit-profile (defer m V A p) p)) = 1
    using def-one-n fin-A prof-A non-blocking-m def-presv-prof
          card.infinite not-one-le-zero
    unfolding non-blocking-def
    by metis
  moreover have
    defer (m ▷ n) V A p =
      defer n V (defer m V A p) (limit-profile (defer m V A p) p)
    using seq-comp-defers-def-set
    by (metis (no-types, opaque-lifting))
  ultimately show card (defer (m ▷ n) V A p) = 1
    by simp
qed
```

Composing a defer-lift invariant and a non-electing electoral module that defers exactly one alternative in sequence with an electing electoral module results in a monotone electoral module.

```
theorem disj-compat-seq[simp]:
  fixes m m′ n :: (′a, ′v, ′a Result) Electoral-Module
  assumes
    compatible: disjoint-compatibility m n and
    module-m′: 𝒮𝒞ℱ-result.electoral-module m′ and
    voters-determine-m′: voters-determine-election m′
  shows disjoint-compatibility (m ▷ m′) n
proof (unfold disjoint-compatibility-def, safe)
  show 𝒮𝒞ℱ-result.electoral-module (m ▷ m′)
    using compatible module-m′ seq-comp-sound
    unfolding disjoint-compatibility-def
    by metis
next
  show 𝒮𝒞ℱ-result.electoral-module n
    using compatible
    unfolding disjoint-compatibility-def
    by metis
next
  fix
    S :: ′a set and
    V :: ′v set
  have modules:
    𝒮𝒞ℱ-result.electoral-module (m ▷ m′) ∧ 𝒮𝒞ℱ-result.electoral-module n
    using compatible module-m′ seq-comp-sound
    unfolding disjoint-compatibility-def
    by metis
  obtain A :: ′a set where
    rej-A:
    A ⊆ S ∧
      (∀ a ∈ A.
        indep-of-alt m V S a ∧ (∀ p. profile V S p ⟶ a ∈ reject m V S p)) ∧
      (∀ a ∈ S − A.
        indep-of-alt n V S a ∧ (∀ p. profile V S p ⟶ a ∈ reject n V S p))
    using compatible
    unfolding disjoint-compatibility-def
    by (metis (no-types, lifting))
  show
    ∃ A ⊆ S.
      (∀ a ∈ A. indep-of-alt (m ▷ m′) V S a ∧
        (∀ p. profile V S p ⟶ a ∈ reject (m ▷ m′) V S p)) ∧
      (∀ a ∈ S − A.
        indep-of-alt n V S a ∧ (∀ p. profile V S p ⟶ a ∈ reject n V S p))
  proof
    have ∀ a p q. a ∈ A ∧ equiv-prof-except-a V S p q a ⟶
          (m ▷ m′) V S p = (m ▷ m′) V S q
    proof (safe)
  fix
    a :: 'a and
    p q :: ('a, 'v) Profile
  assume
    a-in-A: a ∈ A and
    lifting-equiv-p-q: equiv-prof-except-a V S p q a
  hence eq-defer: defer m V S p = defer m V S q
    using rej-A
    unfolding indep-of-alt-def
    by metis
  have profiles: profile V S p ∧ profile V S q
    using lifting-equiv-p-q
    unfolding equiv-prof-except-a-def
    by simp
  hence (defer m V S p) ⊆ S
    using compatible defer-in-alts
    unfolding disjoint-compatibility-def
    by metis
  moreover have a ∉ defer m V S q
    using a-in-A compatible profiles rej-A IntI emptyE result-disj
    unfolding disjoint-compatibility-def
    by metis
  ultimately have
    ∀ v ∈ V. limit-profile (defer m V S p) p v =
                limit-profile (defer m V S q) q v
    using lifting-equiv-p-q negl-diff-imp-eq-limit-prof[of - S]
    unfolding eq-defer limit-profile.simps
    by blast
  hence m' V (defer m V S p) (limit-profile (defer m V S p) p) =
        m' V (defer m V S q) (limit-profile (defer m V S q) q)
    using eq-defer voters-determine-m'
    by simp
  moreover have m V S p = m V S q
    using rej-A a-in-A lifting-equiv-p-q
    unfolding indep-of-alt-def
    by metis
  ultimately show (m ▷ m') V S p = (m ▷ m') V S q
    unfolding sequential-composition.simps
    by (metis (full-types))
qed
moreover have ∀ a' ∈ A. ∀ p'. profile V S p' ⟶ a' ∈ reject (m ▷ m') V S p'
  using rej-A UnI1 prod.sel
  unfolding sequential-composition.simps
  by metis
ultimately show A ⊆ S ∧
    (∀ a' ∈ A. indep-of-alt (m ▷ m') V S a' ∧
      (∀ p'. profile V S p' ⟶ a' ∈ reject (m ▷ m') V S p')) ∧
    (∀ a' ∈ S − A. indep-of-alt n V S a' ∧
      (∀ p'. profile V S p' ⟶ a' ∈ reject n V S p'))
      using rej-A indep-of-alt-def modules
      by (metis (no-types, lifting))
  qed
qed

theorem seq-comp-cond-compat[simp]:
  fixes m n :: ('a, 'v, 'a Result) Electoral-Module
  assumes
    dcc-m: defer-condorcet-consistency m and
    nb-n: non-blocking n and
    ne-n: non-electing n
  shows condorcet-compatibility (m ▷ n)
proof (unfold condorcet-compatibility-def, safe)
  have SCF-result.electoral-module m
    using dcc-m
    unfolding defer-condorcet-consistency-def
    by presburger
  moreover have SCF-result.electoral-module n
    using nb-n
    unfolding non-blocking-def
    by presburger
  ultimately have SCF-result.electoral-module (m ▷ n)
    using seq-comp-sound
    by metis
  thus SCF-result.electoral-module (m ▷ n)
    by presburger
next
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assume
    cw-a: condorcet-winner V A p a and
    a-in-rej-seq-m-n: a ∈ reject (m ▷ n) V A p
  hence ∃ a'. defer-condorcet-consistency m ∧ condorcet-winner V A p a'
    using dcc-m
    by blast
  hence m V A p = ({}, A − (defer m V A p), {a})
    using defer-condorcet-consistency-def cw-a cond-winner-unique
    by (metis (no-types, lifting))
  have sound-m: SCF-result.electoral-module m
    using dcc-m
    unfolding defer-condorcet-consistency-def
    by presburger
  moreover have SCF-result.electoral-module n
    using nb-n
    unfolding non-blocking-def
    by presburger
```

```
ultimately have sound-seq-m-n: 𝒮𝒞ℱ-result.electoral-module (m ▷ n)
  using seq-comp-sound
  by metis
have def-m: defer m V A p = {a}
  using cw-a cond-winner-unique dcc-m snd-conv
  unfolding defer-condorcet-consistency-def
  by (metis (mono-tags, lifting))
have rej-m: reject m V A p = A − {a}
  using cw-a cond-winner-unique dcc-m prod.sel
  unfolding defer-condorcet-consistency-def
  by (metis (mono-tags, lifting))
have elect m V A p = {}
  using cw-a def-m rej-m dcc-m fst-conv
  unfolding defer-condorcet-consistency-def
  by (metis (mono-tags, lifting))
hence diff-elect-m: A − elect m V A p = A
  using Diff-empty
  by (metis (full-types))
have cond-win:
  finite A ∧ finite V ∧ profile V A p
    ∧ a ∈ A ∧ (∀ a'. a' ∈ A − {a'} ⟶ wins V a p a')
  using cw-a condorcet-winner.simps DiffD2 singletonI
  by (metis (no-types))
have ∀ a' A'. (a' :: 'a) ∈ A' ⟶ insert a' (A' − {a'}) = A'
  by blast
have nb-n-full:
  𝒮𝒞ℱ-result.electoral-module n ∧
    (∀ A' V' p'.
      A' ≠ {} ∧ finite A' ∧ finite V' ∧ profile V' A' p'
        ⟶ reject n V' A' p' ≠ A')
  using nb-n non-blocking-def
  by metis
have def-seq-diff:
  defer (m ▷ n) V A p = A − elect (m ▷ n) V A p − reject (m ▷ n) V A p
  using defer-not-elec-or-rej cond-win sound-seq-m-n
  by metis
have set-ins: ∀ a' A'. (a' :: 'a) ∈ A' ⟶ insert a' (A' − {a'}) = A'
  by fastforce
have ∀ p' A' p''. p' = (A' :: 'a set, p'' :: 'a set × 'a set) ⟶ snd p' = p''
  by simp
hence
  snd (elect m V A p
      ∪ elect n V (defer m V A p) (limit-profile (defer m V A p) p),
        reject m V A p
      ∪ reject n V (defer m V A p) (limit-profile (defer m V A p) p),
        defer n V (defer m V A p) (limit-profile (defer m V A p) p)) =
    (reject m V A p
    ∪ reject n V (defer m V A p) (limit-profile (defer m V A p) p),
    defer n V (defer m V A p) (limit-profile (defer m V A p) p))
    by blast
  hence seq-snd-simplified:
    snd ((m ▷ n) V A p) =
      (reject m V A p
      ∪ reject n V (defer m V A p) (limit-profile (defer m V A p) p),
      defer n V (defer m V A p) (limit-profile (defer m V A p) p))
    using sequential-composition.simps
    by metis
  hence seq-rej-union-eq-rej:
    reject m V A p
    ∪ reject n V (defer m V A p) (limit-profile (defer m V A p) p) =
        reject (m ▷ n) V A p
    by simp
  hence seq-rej-union-subset-A:
    reject m V A p
    ∪ reject n V (defer m V A p) (limit-profile (defer m V A p) p) ⊆ A
    using sound-seq-m-n cond-win reject-in-alts
    by (metis (no-types))
  hence A − {a} = reject (m ▷ n) V A p − {a}
    using seq-rej-union-eq-rej defer-not-elec-or-rej cond-win def-m diff-elect-m
          double-diff rej-m sound-m sup-ge1
    by (metis (no-types))
  hence reject (m ▷ n) V A p ⊆ A − {a}
    using seq-rej-union-subset-A seq-snd-simplified set-ins def-seq-diff nb-n-full
          cond-win fst-conv Diff-empty Diff-eq-empty-iff a-in-rej-seq-m-n def-m
          def-presv-prof sound-m ne-n diff-elect-m insert-not-empty defer-in-alts
          reject-not-elected-or-deferred seq-comp-def-then-elect-elec-set finite-subset
          seq-comp-defers-def-set sup-bot.left-neutral
    unfolding non-electing-def
    by (metis (no-types, lifting))
  thus False
    using a-in-rej-seq-m-n
    by blast
next
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a a' :: 'a
  assume
    cw-a: condorcet-winner V A p a and
    not-cw-a': ¬ condorcet-winner V A p a' and
    a'-in-elect-seq-m-n: a' ∈ elect (m ▷ n) V A p
  hence ∃ a''. defer-condorcet-consistency m ∧ condorcet-winner V A p a''
    using dcc-m
    by blast
  hence result-m: m V A p = ({}, A − (defer m V A p), {a})
    using defer-condorcet-consistency-def cw-a cond-winner-unique
    by (metis (no-types, lifting))
```

```
have sound-m: SCF-result.electoral-module m
  using dcc-m
  unfolding defer-condorcet-consistency-def
  by presburger
moreover have SCF-result.electoral-module n
  using nb-n
  unfolding non-blocking-def
  by presburger
ultimately have sound-seq-m-n: SCF-result.electoral-module (m ⊳ n)
  using seq-comp-sound
  by metis
have reject m V A p = A − {a}
  using cw-a dcc-m prod.sel result-m
  unfolding defer-condorcet-consistency-def
  by (metis (mono-tags, lifting))
hence a′-in-rej: a′ ∈ reject m V A p
  using Diff-iff cw-a not-cw-a′ a′-in-elect-seq-m-n subset-iff
      elect-in-alts singleton-iff sound-seq-m-n
  unfolding condorcet-winner.simps
  by (metis (no-types, lifting))
have ∀ p′ A′ p″. p′ = (A′ :: ′a set, p″ :: ′a set × ′a set) ⟶ snd p′ = p″
  by simp
hence m-seq-n:
  snd (elect m V A p
    ∪ elect n V (defer m V A p) (limit-profile (defer m V A p) p),
    reject m V A p
    ∪ reject n V (defer m V A p) (limit-profile (defer m V A p) p),
    defer n V (defer m V A p) (limit-profile (defer m V A p) p)) =
        (reject m V A p
        ∪ reject n V (defer m V A p) (limit-profile (defer m V A p) p),
          defer n V (defer m V A p) (limit-profile (defer m V A p) p))
  by blast
have a′ ∈ elect m V A p
  using a′-in-elect-seq-m-n condorcet-winner.simps cw-a def-presv-prof ne-n
      seq-comp-def-then-elect-elec-set sound-m sup-bot.left-neutral
  unfolding non-electing-def
  by (metis (no-types))
hence a-in-rej-union:
  a ∈ reject m V A p
  ∪ reject n V (defer m V A p) (limit-profile (defer m V A p) p)
  using Diff-iff a′-in-rej condorcet-winner.simps cw-a
      reject-not-elected-or-deferred sound-m
  by (metis (no-types))
have m-seq-n-full:
  (m ⊳ n) V A p =
    (elect m V A p
    ∪ elect n V (defer m V A p) (limit-profile (defer m V A p) p),
    reject m V A p
    ∪ reject n V (defer m V A p) (limit-profile (defer m V A p) p),
      defer n V (defer m V A p) (limit-profile (defer m V A p) p))
    unfolding sequential-composition.simps
    by metis
  have ∀ A′ A″. (A′:: ′a set) = fst (A′, A″ :: ′a set)
    by simp
  hence a ∈ reject (m ▷ n) V A p
    using a-in-rej-union m-seq-n m-seq-n-full
    by presburger
  moreover have
    finite A ∧ finite V ∧ profile V A p
    ∧ a ∈ A ∧ (∀ a″. a″ ∈ A − {a} ⟶ wins V a p a″)
    using cw-a m-seq-n-full a′-in-elect-seq-m-n a′-in-rej ne-n sound-m
    unfolding condorcet-winner.simps
    by metis
  ultimately show False
   using a′-in-elect-seq-m-n IntI empty-iff result-disj sound-seq-m-n a′-in-rej def-presv-prof
         fst-conv m-seq-n-full ne-n non-electing-def sound-m sup-bot.right-neutral
    by metis
next
  fix
    A :: ′a set and
    V :: ′v set and
    p :: (′a, ′v) Profile and
    a a′ :: ′a
  assume
    cw-a: condorcet-winner V A p a and
    a′-in-A: a′ ∈ A and
    not-cw-a′: ¬ condorcet-winner V A p a′
  have reject m V A p = A − {a}
    using cw-a cond-winner-unique dcc-m prod.sel
    unfolding defer-condorcet-consistency-def
    by (metis (mono-tags, lifting))
  moreover have a ≠ a′
    using cw-a not-cw-a′
    by safe
  ultimately have a′ ∈ reject m V A p
    using DiffI a′-in-A singletonD
    by (metis (no-types))
  hence a′ ∈ reject m V A p
      ∪ reject n V (defer m V A p) (limit-profile (defer m V A p) p)
    by blast
  moreover have
    (m ▷ n) V A p =
      (elect m V A p
      ∪ elect n V (defer m V A p) (limit-profile (defer m V A p) p),
        reject m V A p
      ∪ reject n V (defer m V A p) (limit-profile (defer m V A p) p),
       defer n V (defer m V A p) (limit-profile (defer m V A p) p))
    unfolding sequential-composition.simps
    by metis
  moreover have
    snd (elect m V A p
      ∪ elect n V (defer m V A p) (limit-profile (defer m V A p) p),
      reject m V A p
      ∪ reject n V (defer m V A p) (limit-profile (defer m V A p) p),
      defer n V (defer m V A p) (limit-profile (defer m V A p) p)) =
        (reject m V A p
        ∪ reject n V (defer m V A p) (limit-profile (defer m V A p) p),
        defer n V (defer m V A p) (limit-profile (defer m V A p) p))
    using snd-conv
    by metis
  ultimately show a′ ∈ reject (m ▷ n) V A p
    using fst-eqD
    by (metis (no-types))
qed
```

Composing a defer-condorcet-consistent electoral module in sequence with a non-blocking and non-electing electoral module results in a defer-condorcet-consistent module.

```
theorem seq-comp-dcc[simp]:
  fixes m n :: ('a, 'v, 'a Result) Electoral-Module
  assumes
    dcc-m: defer-condorcet-consistency m and
    nb-n: non-blocking n and
    ne-n: non-electing n
  shows defer-condorcet-consistency (m ▷ n)
proof (unfold defer-condorcet-consistency-def, safe)
  have 𝒮𝒞ℱ-result.electoral-module m
    using dcc-m
    unfolding defer-condorcet-consistency-def
    by metis
  thus 𝒮𝒞ℱ-result.electoral-module (m ▷ n)
    using ne-n seq-comp-sound
    unfolding non-electing-def
    by metis
next
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assume cw-a: condorcet-winner V A p a
  hence ∃ a′. defer-condorcet-consistency m ∧ condorcet-winner V A p a′
    using dcc-m
    by blast
  hence result-m: m V A p = ({}, A − (defer m V A p), {a})
    using defer-condorcet-consistency-def cw-a cond-winner-unique
    by (metis (no-types, lifting))
```

```
hence elect-m-empty: elect m V A p = {}
  using eq-fst-iff
  by metis
have sound-m: 𝒮𝒞ℱ-result.electoral-module m
  using dcc-m
  unfolding defer-condorcet-consistency-def
  by metis
hence sound-seq-m-n: 𝒮𝒞ℱ-result.electoral-module (m ▷ n)
  using ne-n seq-comp-sound
  unfolding non-electing-def
  by metis
have defer-eq-a: defer (m ▷ n) V A p = {a}
proof (safe)
  fix a' :: 'a
  assume a'-in-def-seq-m-n: a' ∈ defer (m ▷ n) V A p
  have {a} = {a ∈ A. condorcet-winner V A p a}
    using cond-winner-unique cw-a
    by metis
  moreover have defer-condorcet-consistency m ⟶
        m V A p = ({}, A − defer m V A p, {a ∈ A. condorcet-winner V A p a})
    using cw-a defer-condorcet-consistency-def
    by (metis (no-types))
  ultimately have defer m V A p = {a}
    using dcc-m snd-conv
    by (metis (no-types, lifting))
  hence defer (m ▷ n) V A p = {a}
    using cw-a a'-in-def-seq-m-n empty-iff sound-m nb-n
        seq-comp-def-set-bounded subset-singletonD
    unfolding condorcet-winner.simps non-blocking-def
    by metis
  thus a' = a
    using a'-in-def-seq-m-n
    by blast
next
  have ∃ a'. defer-condorcet-consistency m ∧ condorcet-winner V A p a'
    using cw-a dcc-m
    by blast
  hence m V A p = ({}, A − (defer m V A p), {a})
    using defer-condorcet-consistency-def cw-a cond-winner-unique
    by (metis (no-types, lifting))
  hence elect-m-empty: elect m V A p = {}
    using eq-fst-iff
    by metis
  have profile V (defer m V A p) (limit-profile (defer m V A p) p)
    using condorcet-winner.simps cw-a def-presv-prof sound-m
    by (metis (no-types))
  hence elect n V (defer m V A p) (limit-profile (defer m V A p) p) = {}
    using ne-n non-electing-def
    by metis
    hence elect (m ▷ n) V A p = {}
      using elect-m-empty seq-comp-def-then-elect-elec-set sup-bot.right-neutral
      by (metis (no-types))
    moreover have condorcet-compatibility (m ▷ n)
      using dcc-m nb-n ne-n
      by simp
    hence a ∉ reject (m ▷ n) V A p
      unfolding condorcet-compatibility-def
      using cw-a
      by metis
    ultimately show a ∈ defer (m ▷ n) V A p
      using cw-a electoral-mod-defer-elem empty-iff
            sound-seq-m-n condorcet-winner.simps
      by metis
  qed
  have profile V (defer m V A p) (limit-profile (defer m V A p) p)
    using condorcet-winner.simps cw-a def-presv-prof sound-m
    by (metis (no-types))
  hence elect n V (defer m V A p) (limit-profile (defer m V A p) p) = {}
    using ne-n
    unfolding non-electing-def
    by metis
  hence elect (m ▷ n) V A p = {}
    using elect-m-empty seq-comp-def-then-elect-elec-set sup-bot.right-neutral
    by (metis (no-types))
  moreover have def-seq-m-n-eq-a: defer (m ▷ n) V A p = {a}
    using cw-a defer-eq-a
    by (metis (no-types))
  ultimately have (m ▷ n) V A p = ({}, A − {a}, {a})
    using Diff-empty cw-a elect-rej-def-combination
          reject-not-elected-or-deferred sound-seq-m-n condorcet-winner.simps
    by (metis (no-types))
  moreover have {a′ ∈ A. condorcet-winner V A p a′} = {a}
    using cw-a cond-winner-unique
    by metis
  ultimately show (m ▷ n) V A p
      = ({}, A − defer (m ▷ n) V A p, {a′ ∈ A. condorcet-winner V A p a′})
    using def-seq-m-n-eq-a
    by metis
qed
```

Composing a defer-lift invariant and a non-electing electoral module that defers exactly one alternative in sequence with an electing electoral module results in a monotone electoral module.

```
theorem seq-comp-mono[simp]:
  fixes m n :: ('a, 'v, 'a Result) Electoral-Module
  assumes
    def-monotone-m: defer-lift-invariance m and
    non-ele-m: non-electing m and
    def-one-m: defers 1 m and
    electing-n: electing n
  shows monotonicity (m ▷ n)
proof (unfold monotonicity-def, safe)
  have 𝒮𝒞ℱ-result.electoral-module m
    using non-ele-m
    unfolding non-electing-def
    by simp
  moreover have 𝒮𝒞ℱ-result.electoral-module n
    using electing-n
    unfolding electing-def
    by simp
  ultimately show 𝒮𝒞ℱ-result.electoral-module (m ▷ n)
    using seq-comp-sound
    by metis
next
  fix
    A :: 'a set and
    V :: 'v set and
    p q :: ('a, 'v) Profile and
    w :: 'a
  assume
    elect-w-in-p: w ∈ elect (m ▷ n) V A p and
    lifted-w: Profile.lifted V A p q w
  thus w ∈ elect (m ▷ n) V A q
    unfolding lifted-def
    using seq-comp-def-then-elect lifted-w assms
    unfolding defer-lift-invariance-def
    by metis
qed
```

Composing a defer-invariant-monotone electoral module in sequence before a non-electing, defer-monotone electoral module that defers exactly 1 alternative results in a defer-lift-invariant electoral module.

```
theorem def-inv-mono-imp-def-lift-inv[simp]:
  fixes m n :: ('a, 'v, 'a Result) Electoral-Module
  assumes
    strong-def-mon-m: defer-invariant-monotonicity m and
    non-electing-n: non-electing n and
    defers-one: defers 1 n and
    defer-monotone-n: defer-monotonicity n and
    voters-determine-n: voters-determine-election n
  shows defer-lift-invariance (m ▷ n)
proof (unfold defer-lift-invariance-def, safe)
  have 𝒮𝒞ℱ-result.electoral-module m
    using strong-def-mon-m
    unfolding defer-invariant-monotonicity-def
    by metis
  moreover have 𝒮𝒞ℱ-result.electoral-module n
```

```
    using defers-one
    unfolding defers-def
    by metis
  ultimately show 𝒮𝒞ℱ-result.electoral-module (m ▷ n)
    using seq-comp-sound
    by metis
next
  fix
    A :: 'a set and
    V :: 'v set and
    p q :: ('a, 'v) Profile and
    a :: 'a
  assume
    defer-a-p: a ∈ defer (m ▷ n) V A p and
    lifted-a: Profile.lifted V A p q a
  have non-electing-m: non-electing m
    using strong-def-mon-m
    unfolding defer-invariant-monotonicity-def
    by simp
  have electoral-mod-m: 𝒮𝒞ℱ-result.electoral-module m
    using strong-def-mon-m
    unfolding defer-invariant-monotonicity-def
    by metis
  have electoral-mod-n: 𝒮𝒞ℱ-result.electoral-module n
    using defers-one
    unfolding defers-def
    by metis
  have finite-profile-p: finite-profile V A p
    using lifted-a
    unfolding Profile.lifted-def
    by simp
  have finite-profile-q: finite-profile V A q
    using lifted-a
    unfolding Profile.lifted-def
    by simp
  have 1 ≤ card A
   using Profile.lifted-def card-eq-0-iff emptyE less-one lifted-a linorder-le-less-linear
    by metis
  hence n-defers-exactly-one-p: card (defer n V A p) = 1
    using finite-profile-p defers-one
    unfolding defers-def
    by (metis (no-types))
  have fin-prof-def-m-q:
    profile V (defer m V A q) (limit-profile (defer m V A q) q)
    using def-presv-prof electoral-mod-m finite-profile-q
    by (metis (no-types))
  have def-seq-m-n-q:
    defer (m ▷ n) V A q =
      defer n V (defer m V A q) (limit-profile (defer m V A q) q)
  using seq-comp-defers-def-set
  by simp
have prof-def-m: profile V (defer m V A p) (limit-profile (defer m V A p) p)
  using def-presv-prof electoral-mod-m finite-profile-p
  by (metis (no-types))
hence prof-seq-comp-m-n:
  profile V (defer n V (defer m V A p) (limit-profile (defer m V A p) p))
       (limit-profile (defer n V (defer m V A p) (limit-profile (defer m V A p) p))
          (limit-profile (defer m V A p) p))
  using def-presv-prof electoral-mod-n
  by (metis (no-types))
have a-non-empty: a ∉ {}
  by simp
have def-seq-m-n:
  defer (m ⊳ n) V A p =
    defer n V (defer m V A p) (limit-profile (defer m V A p) p)
  using seq-comp-defers-def-set
  by simp
have 1 ≤ card (defer n V (defer m V A p) (limit-profile (defer m V A p) p))
  using a-non-empty card-gt-0-iff defer-a-p electoral-mod-n prof-def-m
        seq-comp-defers-def-set One-nat-def Suc-leI defer-in-alts
        electoral-mod-m finite-profile-p finite-subset
  by (metis (mono-tags))
hence card (defer n V (defer n V (defer m V A p)
      (limit-profile (defer m V A p) p))
    (limit-profile (defer n V (defer m V A p)
      (limit-profile (defer m V A p) p))
    (limit-profile (defer m V A p) p))) = 1
  using n-defers-exactly-one-p prof-seq-comp-m-n defers-one defer-in-alts
        electoral-mod-m finite-profile-p finite-subset prof-def-m
  unfolding defers-def
  by metis
hence defer-seq-m-n-eq-one: card (defer (m ⊳ n) V A p) = 1
  using One-nat-def Suc-leI a-non-empty card-gt-0-iff def-seq-m-n defer-a-p
        defers-one electoral-mod-m prof-def-m finite-profile-p
        seq-comp-def-set-trans defer-in-alts rev-finite-subset
  unfolding defers-def
  by metis
hence def-seq-m-n-eq-a: defer (m ⊳ n) V A p = {a}
  using defer-a-p is-singleton-altdef is-singleton-the-elem singletonD
  by (metis (no-types))
show (m ⊳ n) V A p = (m ⊳ n) V A q
proof (cases)
  assume defer m V A q ≠ defer m V A p
  hence defer m V A q = {a}
    using defer-a-p electoral-mod-n finite-profile-p lifted-a seq-comp-def-set-trans
          strong-def-mon-m
    unfolding defer-invariant-monotonicity-def
    by (metis (no-types))
  moreover from this
  have (a ∈ defer m V A p) ⟶ card (defer (m ▷ n) V A q) = 1
    using card-eq-0-iff card-insert-disjoint defers-one electoral-mod-m empty-iff
          order-refl finite.emptyI seq-comp-defers-def-set def-presv-prof
          finite-profile-q finite.insertI
    unfolding One-nat-def defers-def
    by metis
  moreover have a ∈ defer m V A p
    using electoral-mod-m electoral-mod-n defer-a-p seq-comp-def-set-bounded
          finite-profile-p finite-profile-q
    by blast
  ultimately have defer (m ▷ n) V A q = {a}
   using Collect-mem-eq card-1-singletonE empty-Collect-eq insertCI subset-singletonD
          def-seq-m-n-q defer-in-alts electoral-mod-n fin-prof-def-m-q
    by (metis (no-types, lifting))
  hence defer (m ▷ n) V A p = defer (m ▷ n) V A q
    using def-seq-m-n-eq-a
    by presburger
  moreover have elect (m ▷ n) V A p = elect (m ▷ n) V A q
   using prof-def-m fin-prof-def-m-q finite-profile-p finite-profile-q non-electing-def
          non-electing-m non-electing-n seq-comp-def-then-elect-elec-set
    by metis
  ultimately show ?thesis
    using electoral-mod-m electoral-mod-n eq-def-and-elect-imp-eq
          finite-profile-p finite-profile-q seq-comp-sound
    by (metis (no-types))
next
  assume ¬ (defer m V A q ≠ defer m V A p)
  hence def-eq: defer m V A q = defer m V A p
    by presburger
  have elect m V A p = {}
    using finite-profile-p non-electing-m
    unfolding non-electing-def
    by simp
  moreover have elect m V A q = {}
    using finite-profile-q non-electing-m
    unfolding non-electing-def
    by simp
  ultimately have elect-m-equal:
    elect m V A p = elect m V A q
    by simp
  have (∀ v ∈ V. (limit-profile (defer m V A p) p) v =
                  (limit-profile (defer m V A p) q) v)
        ∨ lifted V (defer m V A q) (limit-profile (defer m V A p) p)
                (limit-profile (defer m V A p) q) a
    using def-eq defer-in-alts electoral-mod-m lifted-a finite-profile-q
          limit-prof-eq-or-lifted
    by metis
  moreover have
  (∀ v ∈ V. (limit-profile (defer m V A p) p) v =
              (limit-profile (defer m V A p) q) v)
    ⟶ n V (defer m V A p) (limit-profile (defer m V A p) p) =
        n V (defer m V A q) (limit-profile (defer m V A q) q)
  using voters-determine-n def-eq
  unfolding voters-determine-election.simps
  by presburger
moreover have
  lifted V (defer m V A q) (limit-profile (defer m V A p) p)
                           (limit-profile (defer m V A p) q) a
    ⟶ defer n V (defer m V A p) (limit-profile (defer m V A p) p) =
        defer n V (defer m V A q) (limit-profile (defer m V A q) q)
proof (intro impI)
  assume lifted:
    Profile.lifted V (defer m V A q) (limit-profile (defer m V A p) p)
        (limit-profile (defer m V A p) q) a
  hence a ∈ defer n V (defer m V A q) (limit-profile (defer m V A q) q)
    using lifted-a def-seq-m-n defer-a-p defer-monotone-n
        fin-prof-def-m-q def-eq
    unfolding defer-monotonicity-def
    by metis
  hence a ∈ defer (m ▷ n) V A q
    using def-seq-m-n-q
    by simp
  moreover have card (defer (m ▷ n) V A q) = 1
    using def-seq-m-n-q defers-one def-eq defer-seq-m-n-eq-one defers-def lifted
      electoral-mod-m fin-prof-def-m-q finite-profile-p seq-comp-def-card-bounded
        Profile.lifted-def
    by (metis (no-types, lifting))
  ultimately have defer (m ▷ n) V A q = {a}
    using a-non-empty card-1-singletonE insertE
    by metis
  thus defer n V (defer m V A p) (limit-profile (defer m V A p) p)
      = defer n V (defer m V A q) (limit-profile (defer m V A q) q)
    using def-seq-m-n-eq-a def-seq-m-n-q def-seq-m-n
    by presburger
qed
ultimately have defer (m ▷ n) V A p = defer (m ▷ n) V A q
  using def-seq-m-n def-seq-m-n-q
  by presburger
hence defer (m ▷ n) V A p = defer (m ▷ n) V A q
  using a-non-empty def-eq def-seq-m-n def-seq-m-n-q
      defer-a-p defer-monotone-n finite-profile-p
      defer-seq-m-n-eq-one defers-one electoral-mod-m
      fin-prof-def-m-q
  unfolding defers-def
  by (metis (no-types, lifting))
moreover from this
have reject (m ▷ n) V A p = reject (m ▷ n) V A q
```

```
    using electoral-mod-m electoral-mod-n finite-profile-p finite-profile-q non-electing-def
        non-electing-m non-electing-n eq-def-and-elect-imp-eq seq-comp-presv-non-electing
      by (metis (no-types))
    ultimately have snd ((m ▷ n) V A p) = snd ((m ▷ n) V A q)
      using prod-eqI
      by metis
    moreover have elect (m ▷ n) V A p = elect (m ▷ n) V A q
      using prof-def-m fin-prof-def-m-q non-electing-n finite-profile-p finite-profile-q
          non-electing-def def-eq elect-m-equal fst-conv
      unfolding sequential-composition.simps
      by (metis (no-types))
    ultimately show (m ▷ n) V A p = (m ▷ n) V A q
      using prod-eqI
      by metis
  qed
qed

end
```

6.4 Parallel Composition

```
theory Parallel-Composition
  imports Basic-Modules/Component-Types/Aggregator
        Basic-Modules/Component-Types/Electoral-Module
begin
```

The parallel composition composes a new electoral module from two electoral modules combined with an aggregator. Therein, the two modules each make a decision and the aggregator combines them to a single (aggregated) result.

6.4.1 Definition

```
fun parallel-composition :: ('a, 'v, 'a Result) Electoral-Module ⇒
        ('a, 'v, 'a Result) Electoral-Module ⇒ 'a Aggregator ⇒
        ('a, 'v, 'a Result) Electoral-Module where
  parallel-composition m n agg V A p = agg A (m V A p) (n V A p)

abbreviation parallel :: ('a, 'v, 'a Result) Electoral-Module ⇒ 'a Aggregator ⇒
        ('a, 'v, 'a Result) Electoral-Module ⇒ ('a, 'v, 'a Result) Electoral-Module
      (- ∥_- - [50, 1000, 51] 50) where
  m ∥_a n ≡ parallel-composition m n a
```

6.4.2 Soundness

```
theorem par-comp-sound[simp]:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    a :: 'a Aggregator
  assumes
    𝒮𝒞ℱ-result.electoral-module m and
    𝒮𝒞ℱ-result.electoral-module n and
    aggregator a
  shows 𝒮𝒞ℱ-result.electoral-module (m ∥a n)
proof (unfold 𝒮𝒞ℱ-result.electoral-module.simps, safe)
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assume profile V A p
  moreover have
    ∀ a'. aggregator a' =
      (∀ A' e r d e' r' d'.
        (well-formed-𝒮𝒞ℱ (A' :: 'a set) (e, r', d)
        ∧ well-formed-𝒮𝒞ℱ A' (r, d', e'))
            ⟶ well-formed-𝒮𝒞ℱ A' (a' A' (e, r', d) (r, d', e')))
    unfolding aggregator-def
    by blast
  moreover have
    ∀ m' V' A' p'.
      (𝒮𝒞ℱ-result.electoral-module m' ∧ finite (A' :: 'a set)
        ∧ finite (V' :: 'v set) ∧ profile V' A' p')
      ⟶ well-formed-𝒮𝒞ℱ A' (m' V' A' p')
    using par-comp-result-sound
    by (metis (no-types))
  ultimately have well-formed-𝒮𝒞ℱ A (a A (m V A p) (n V A p))
    using elect-rej-def-combination assms
    by (metis par-comp-result-sound)
  thus well-formed-𝒮𝒞ℱ A ((m ∥a n) V A p)
    by simp
qed
```

6.4.3 Composition Rule

Using a conservative aggregator, the parallel composition preserves the property non-electing.

```
theorem conserv-agg-presv-non-electing[simp]:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    a :: 'a Aggregator
  assumes
    non-electing-m: non-electing m and
    non-electing-n: non-electing n and
    conservative: agg-conservative a
  shows non-electing (m ∥a n)
```

```
proof (unfold non-electing-def, safe)
  have 𝒮𝒞ℱ-result.electoral-module m
    using non-electing-m
    unfolding non-electing-def
    by simp
  moreover have 𝒮𝒞ℱ-result.electoral-module n
    using non-electing-n
    unfolding non-electing-def
    by simp
  moreover have aggregator a
    using conservative
    unfolding agg-conservative-def
    by simp
  ultimately show 𝒮𝒞ℱ-result.electoral-module (m ∥a n)
    using par-comp-sound
    by simp
next
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    w :: 'a
  assume
    prof-A: profile V A p and
    w-wins: w ∈ elect (m ∥a n) V A p
  have emod-m: 𝒮𝒞ℱ-result.electoral-module m
    using non-electing-m
    unfolding non-electing-def
    by simp
  have emod-n: 𝒮𝒞ℱ-result.electoral-module n
    using non-electing-n
    unfolding non-electing-def
    by simp
  have ∀ r r' d d' e e' A' f.
        ((well-formed-𝒮𝒞ℱ (A' :: 'a set) (e', r', d') ∧
          well-formed-𝒮𝒞ℱ A' (e, r, d)) ⟶
          elect-r (f A' (e', r', d') (e, r, d)) ⊆ e' ∪ e ∧
            reject-r (f A' (e', r', d') (e, r, d)) ⊆ r' ∪ r ∧
            defer-r (f A' (e', r', d') (e, r, d)) ⊆ d' ∪ d) =
              ((well-formed-𝒮𝒞ℱ A' (e', r', d') ∧
                well-formed-𝒮𝒞ℱ A' (e, r, d)) ⟶
                elect-r (f A' (e', r', d') (e, r, d)) ⊆ e' ∪ e ∧
                  reject-r (f A' (e', r', d') (e, r, d)) ⊆ r' ∪ r ∧
                  defer-r (f A' (e', r', d') (e, r, d)) ⊆ d' ∪ d)
    by linarith
  hence ∀ a'. agg-conservative a' =
        (aggregator a' ∧
          (∀ A' e e' d d' r r'.
            (well-formed-𝒮𝒞ℱ (A' :: 'a set) (e, r, d) ∧
              well-formed-SCF A′ (e′, r′, d′)) ⟶
               elect-r (a′ A′ (e, r, d) (e′, r′, d′)) ⊆ e ∪ e′ ∧
                 reject-r (a′ A′ (e, r, d) (e′, r′, d′)) ⊆ r ∪ r′ ∧
                 defer-r (a′ A′ (e, r, d) (e′, r′, d′)) ⊆ d ∪ d′))
    unfolding agg-conservative-def
    by simp
  hence aggregator a ∧
        (∀ A′ e e′ d d′ r r′.
          (well-formed-SCF A′ (e, r, d) ∧
           well-formed-SCF A′ (e′, r′, d′)) ⟶
            elect-r (a A′ (e, r, d) (e′, r′, d′)) ⊆ e ∪ e′ ∧
              reject-r (a A′ (e, r, d) (e′, r′, d′)) ⊆ r ∪ r′ ∧
              defer-r (a A′ (e, r, d) (e′, r′, d′)) ⊆ d ∪ d′)
    using conservative
    by presburger
  hence let c = (a A (m V A p) (n V A p)) in
        (elect-r c ⊆ ((elect m V A p) ∪ (elect n V A p)))
    using emod-m emod-n par-comp-result-sound
        prod.collapse prof-A
    by metis
  hence w ∈ ((elect m V A p) ∪ (elect n V A p))
    using w-wins
    by auto
  thus w ∈ {}
    using sup-bot-right prof-A
        non-electing-m non-electing-n
    unfolding non-electing-def
    by (metis (no-types, lifting))
qed

end
```

6.5 Loop Composition

```
theory Loop-Composition
  imports Basic-Modules/Component-Types/Termination-Condition
        Basic-Modules/Defer-Module
        Sequential-Composition
begin
```

The loop composition uses the same module in sequence, combined with a termination condition, until either

- the termination condition is met or

- no new decisions are made (i.e., a fixed point is reached).

6.5.1 Definition

```
lemma loop-termination-helper:
  fixes
    m acc :: ('a, 'v, 'a Result) Electoral-Module and
    t :: 'a Termination-Condition and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assumes
    ¬ t (acc V A p) and
    defer (acc ▷ m) V A p ⊂ defer acc V A p and
    finite (defer acc V A p)
  shows ((acc ▷ m, m, t, V, A, p), (acc, m, t, V, A, p)) ∈
          measure (λ (acc, m, t, V, A, p). card (defer acc V A p))
  using assms psubset-card-mono
  by simp
```

This function handles the accumulator for the following loop composition function.

```
function loop-comp-helper :: ('a, 'v, 'a Result) Electoral-Module ⇒
        ('a, 'v, 'a Result) Electoral-Module ⇒ 'a Termination-Condition ⇒
        ('a, 'v, 'a Result) Electoral-Module where
    loop-comp-helper-finite:
    finite (defer acc V A p) ∧ (defer (acc ▷ m) V A p) ⊂ (defer acc V A p)
        ⟶ t (acc V A p) ⟹
    loop-comp-helper acc m t V A p = acc V A p |
    loop-comp-helper-infinite:
    ¬ (finite (defer acc V A p) ∧ (defer (acc ▷ m) V A p) ⊂ (defer acc V A p)
        ⟶ t (acc V A p)) ⟹
    loop-comp-helper acc m t V A p = loop-comp-helper (acc ▷ m) m t V A p
proof −
  fix
    P :: bool and
    accum ::
    ('a, 'v, 'a Result) Electoral-Module × ('a, 'v, 'a Result) Electoral-Module
        × 'a Termination-Condition × 'v set × 'a set × ('a, 'v) Profile
  have accum-exists: ∃ m n t V A p. (m, n, t, V, A, p) = accum
    using prod-cases5
    by metis
  assume
    ⋀ acc V A p m t.
      finite (defer acc V A p) ∧ defer (acc ▷ m) V A p ⊂ defer acc V A p
          ⟶ t (acc V A p) ⟹ accum = (acc, m, t, V, A, p) ⟹ P and
    ⋀ acc V A p m t.
      ¬ (finite (defer acc V A p) ∧ defer (acc ▷ m) V A p ⊂ defer acc V A p
        ⟶ t (acc V A p)) ⟹ accum = (acc, m, t, V, A, p) ⟹ P
  thus P
    using accum-exists
    by metis
next
  fix
    t t′ :: ′a Termination-Condition and
    acc acc′ :: (′a, ′v, ′a Result) Electoral-Module and
    A A′ :: ′a set and
    V V′ :: ′v set and
    p p′ :: (′a, ′v) Profile and
    m m′ :: (′a, ′v, ′a Result) Electoral-Module
  assume
    finite (defer acc V A p)
    ∧ defer (acc ▹ m) V A p ⊂ defer acc V A p
       ⟶ t (acc V A p) and
    finite (defer acc′ V′ A′ p′)
    ∧ defer (acc′ ▹ m′) V′ A′ p′ ⊂ defer acc′ V′ A′ p′
       ⟶ t′ (acc′ V′ A′ p′) and
    (acc, m, t, V, A, p) = (acc′, m′, t′, V′, A′, p′)
  thus acc V A p = acc′ V′ A′ p′
    by fastforce
next
  fix
    t t′ :: ′a Termination-Condition and
    acc acc′ :: (′a, ′v, ′a Result) Electoral-Module and
    A A′ :: ′a set and
    V V′ :: ′v set and
    p p′ :: (′a, ′v) Profile and
    m m′ :: (′a, ′v, ′a Result) Electoral-Module
  assume
    finite (defer acc V A p)
    ∧ defer (acc ▹ m) V A p ⊂ defer acc V A p
         ⟶ t (acc V A p) and
    ¬ (finite (defer acc′ V′ A′ p′)
    ∧ defer (acc′ ▹ m′) V′ A′ p′ ⊂ defer acc′ V′ A′ p′
         ⟶ t′ (acc′ V′ A′ p′)) and
    (acc, m, t, V, A, p) = (acc′, m′, t′, V′, A′, p′)
  thus acc V A p = loop-comp-helper-sumC (acc′ ▹ m′, m′, t′, V′, A′, p′)
    by force
next
  fix
    t t′ :: ′a Termination-Condition and
    acc acc′ :: (′a, ′v, ′a Result) Electoral-Module and
    A A′ :: ′a set and
    V V′ :: ′v set and
    p p′ :: (′a, ′v) Profile and
    m m′ :: (′a, ′v, ′a Result) Electoral-Module
  assume
```

```
    ¬ (finite (defer acc V A p)
    ∧ defer (acc ⊳ m) V A p ⊂ defer acc V A p
        ⟶ t (acc V A p)) and
    ¬ (finite (defer acc′ V′ A′ p′)
    ∧ defer (acc′ ⊳ m′) V′ A′ p′ ⊂ defer acc′ V′ A′ p′
        ⟶ t′ (acc′ V′ A′ p′)) and
    (acc, m, t, V, A, p) = (acc′, m′, t′, V′, A′, p′)
  thus loop-comp-helper-sumC (acc ⊳ m, m, t, V, A, p) =
              loop-comp-helper-sumC (acc′ ⊳ m′, m′, t′, V′, A′, p′)
    by force
qed
termination
proof (safe)
  fix
    m n :: (′b, ′a, ′b Result) Electoral-Module and
    t :: ′b Termination-Condition and
    A :: ′b set and
    V :: ′a set and
    p :: (′b, ′a) Profile
  have term-rel:
    ∃ R. wf R ∧
        (finite (defer m V A p)
        ∧ defer (m ⊳ n) V A p ⊂ defer m V A p
      ⟶ t (m V A p)
        ∨ ((m ⊳ n, n, t, V, A, p), (m, n, t, V, A, p)) ∈ R)
    using loop-termination-helper wf-measure termination
    by (metis (no-types))
  obtain
    R :: (((′b, ′a, ′b Result) Electoral-Module
            × (′b, ′a, ′b Result) Electoral-Module
            × (′b Termination-Condition) × ′a set × ′b set
            × (′b, ′a) Profile)
          × (′b, ′a, ′b Result) Electoral-Module
            × (′b, ′a, ′b Result) Electoral-Module
            × (′b Termination-Condition) × ′a set × ′b set
            × (′b, ′a) Profile) set where
    wf R ∧
      (finite (defer m V A p)
        ∧ defer (m ⊳ n) V A p ⊂ defer m V A p
      ⟶ t (m V A p)
        ∨ ((m ⊳ n, n, t, V, A, p), m, n, t, V, A, p) ∈ R)
    using term-rel
    by presburger
  have ∀ R′.
    All (loop-comp-helper-dom ::
      (′b, ′a, ′b Result) Electoral-Module × (′b, ′a, ′b Result) Electoral-Module
      × ′b Termination-Condition × ′a set × ′b set × (′b, ′a) Profile ⇒ bool) ∨
      (∃ t′ m′ A′ V′ p′ n′. wf R′ ⟶
        ((m′ ⊳ n′, n′, t′, V′ :: ′a set, A′ :: ′b set, p′), m′, n′, t′, V′, A′, p′) ∉ R′
      ∧ finite (defer m′ V′ A′ p′) ∧ defer (m′ ▷ n′) V′ A′ p′ ⊂ defer m′ V′ A′ p′
      ∧ ¬ t′ (m′ V′ A′ p′))
    using termination
    by metis
  thus loop-comp-helper-dom (m, n, t, V, A, p)
    using loop-termination-helper wf-measure
    by metis
qed

lemma loop-comp-code-helper[code]:
  fixes
    m acc :: ('a, 'v, 'a Result) Electoral-Module and
    t :: 'a Termination-Condition and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  shows
    loop-comp-helper acc m t V A p =
      (if (t (acc V A p) ∨ ¬ ((defer (acc ▷ m) V A p) ⊂ (defer acc V A p))
      ∨ infinite (defer acc V A p))
      then (acc V A p) else (loop-comp-helper (acc ▷ m) m t V A p))
  using loop-comp-helper.simps
  by (metis (no-types))

function loop-composition :: ('a, 'v, 'a Result) Electoral-Module ⇒
        'a Termination-Condition ⇒ ('a, 'v, 'a Result) Electoral-Module where
  t ({}, {}, A)
    ⟹ loop-composition m t V A p = defer-module V A p |
  ¬(t ({}, {}, A))
    ⟹ loop-composition m t V A p = (loop-comp-helper m m t) V A p
  by (fastforce, simp-all)
termination
  using termination wf-empty
  by blast

abbreviation loop :: ('a, 'v, 'a Result) Electoral-Module ⇒
        'a Termination-Condition ⇒ ('a, 'v, 'a Result) Electoral-Module
        (- ↺- 50) where
  m ↺_t ≡ loop-composition m t

lemma loop-comp-code[code]:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    t :: 'a Termination-Condition and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  shows loop-composition m t V A p =
          (if (t ({}, {}, A))
      then (defer-module V A p) else (loop-comp-helper m m t) V A p)
  by simp

lemma loop-comp-helper-imp-partit:
  fixes
    m acc :: ('a, 'v, 'a Result) Electoral-Module and
    t :: 'a Termination-Condition and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    n :: nat
  assumes
    module-m: SCF-result.electoral-module m and
    profile: profile V A p and
    module-acc: SCF-result.electoral-module acc and
    defer-card-n: n = card (defer acc V A p)
  shows well-formed-SCF A (loop-comp-helper acc m t V A p)
  using assms
proof (induct arbitrary: acc rule: less-induct)
  case (less)
  have ∀ m' n'.
    (SCF-result.electoral-module m' ∧ SCF-result.electoral-module n')
      ⟶ SCF-result.electoral-module (m' ⊳ n')
    using seq-comp-sound
    by metis
  hence SCF-result.electoral-module (acc ⊳ m)
    using less.prems module-m
    by blast
  hence ¬ t (acc V A p) ∧ defer (acc ⊳ m) V A p ⊂ defer acc V A p ∧
          finite (defer acc V A p) ⟶
        well-formed-SCF A (loop-comp-helper acc m t V A p)
    using less.hyps less.prems loop-comp-helper-infinite
        psubset-card-mono
  by metis
  moreover have well-formed-SCF A (acc V A p)
    using less.prems profile
    unfolding SCF-result.electoral-module.simps
    by metis
  ultimately show ?case
    using loop-comp-code-helper
    by (metis (no-types))
qed
```

6.5.2 Soundness

```
theorem loop-comp-sound:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    t :: 'a Termination-Condition
  assumes SCF-result.electoral-module m
  shows SCF-result.electoral-module (m ↻t)
  using def-mod-sound loop-composition.simps
        loop-comp-helper-imp-partit assms
  unfolding SCF-result.electoral-module.simps
  by metis

lemma loop-comp-helper-imp-no-def-incr:
  fixes
    m acc :: ('a, 'v, 'a Result) Electoral-Module and
    t :: 'a Termination-Condition and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    n :: nat
  assumes
    module-m: SCF-result.electoral-module m and
    profile: profile V A p and
    mod-acc: SCF-result.electoral-module acc and
    card-n-defer-acc: n = card (defer acc V A p)
  shows defer (loop-comp-helper acc m t) V A p ⊆ defer acc V A p
  using assms
proof (induct arbitrary: acc rule: less-induct)
  case (less)
  have emod-acc-m: SCF-result.electoral-module (acc ▷ m)
    using less.prems module-m seq-comp-sound
    by blast
  have ∀ A A'. (finite A ∧ A' ⊂ A) ⟶ card A' < card A
    using psubset-card-mono
    by metis
  hence ¬ t (acc V A p) ∧ defer (acc ▷ m) V A p ⊂ defer acc V A p ∧
          finite (defer acc V A p) ⟶
        defer (loop-comp-helper (acc ▷ m) m t) V A p ⊆ defer acc V A p
    using emod-acc-m less.hyps less.prems
    by blast
  hence ¬ t (acc V A p) ∧ defer (acc ▷ m) V A p ⊂ defer acc V A p ∧
          finite (defer acc V A p) ⟶
        defer (loop-comp-helper acc m t) V A p ⊆ defer acc V A p
    using loop-comp-helper-infinite
    by (metis (no-types))
  thus ?case
    using eq-iff loop-comp-code-helper
    by (metis (no-types))
qed
```

6.5.3 Lemmas

```
lemma loop-comp-helper-def-lift-inv-helper:
  fixes
```

```
    m acc :: ('a, 'v, 'a Result) Electoral-Module and
    t :: 'a Termination-Condition and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    n :: nat
  assumes
    monotone-m: defer-lift-invariance m and
    prof: profile V A p and
    dli-acc: defer-lift-invariance acc and
    card-n-defer: n = card (defer acc V A p) and
    defer-finite: finite (defer acc V A p) and
    voters-determine-m: voters-determine-election m
  shows
    ∀ q a. a ∈ (defer (loop-comp-helper acc m t) V A p) ∧ lifted V A p q a ⟶
        (loop-comp-helper acc m t) V A p = (loop-comp-helper acc m t) V A q
  using assms
proof (induct n arbitrary: acc rule: less-induct)
  case (less n)
  have defer-card-comp:
    defer-lift-invariance acc ⟶
        (∀ q a. a ∈ (defer (acc ▷ m) V A p) ∧ lifted V A p q a ⟶
            card (defer (acc ▷ m) V A p) = card (defer (acc ▷ m) V A q))
    using monotone-m def-lift-inv-seq-comp-help voters-determine-m
    by metis
  have defer-lift-invariance acc ⟶
          (∀ q a. a ∈ (defer acc V A p) ∧ lifted V A p q a ⟶
            card (defer acc V A p) = card (defer acc V A q))
    unfolding defer-lift-invariance-def
    by simp
  hence defer-card-acc:
    defer-lift-invariance acc ⟶
        (∀ q a. (a ∈ (defer (acc ▷ m) V A p) ∧ lifted V A p q a) ⟶
            card (defer acc V A p) = card (defer acc V A q))
    using assms seq-comp-def-set-trans
    unfolding defer-lift-invariance-def
    by metis
  thus ?case
  proof (cases)
    assume card-unchanged:
      card (defer (acc ▷ m) V A p) = card (defer acc V A p)
    have defer-lift-invariance acc ⟶
            (∀ q a. a ∈ (defer acc V A p) ∧ lifted V A p q a ⟶
              (loop-comp-helper acc m t) V A q = acc V A q)
    proof (safe)
      fix
        q :: ('a, 'v) Profile and
        a :: 'a
      assume
        dli-acc: defer-lift-invariance acc and
        a-in-def-acc: a ∈ defer acc V A p and
        lifted-A: Profile.lifted V A p q a
      moreover have 𝒮𝒞ℱ-result.electoral-module m
        using monotone-m
        unfolding defer-lift-invariance-def
        by simp
      moreover have emod-acc: 𝒮𝒞ℱ-result.electoral-module acc
        using dli-acc
        unfolding defer-lift-invariance-def
        by simp
      moreover have acc-eq-pq: acc V A q = acc V A p
        using a-in-def-acc dli-acc lifted-A
        unfolding defer-lift-invariance-def
        by (metis (full-types))
      ultimately have finite (defer acc V A p)
                       ⟶ loop-comp-helper acc m t V A q = acc V A q
        using card-unchanged defer-card-comp prof loop-comp-code-helper
              psubset-card-mono dual-order.strict-iff-order
              seq-comp-def-set-bounded less
        by (metis (mono-tags, lifting))
      thus loop-comp-helper acc m t V A q = acc V A q
        using acc-eq-pq loop-comp-code-helper
        by (metis (full-types))
    qed
    moreover from card-unchanged
    have (loop-comp-helper acc m t) V A p = acc V A p
      using loop-comp-code-helper order.strict-iff-order psubset-card-mono
      by metis
    ultimately have
      defer-lift-invariance (acc ▷ m) ∧ defer-lift-invariance acc
      ⟶ (∀ q a. a ∈ (defer (loop-comp-helper acc m t) V A p)
                  ∧ lifted V A p q a
            ⟶ (loop-comp-helper acc m t) V A p =
                  (loop-comp-helper acc m t) V A q)
      unfolding defer-lift-invariance-def
      by metis
    moreover have defer-lift-invariance (acc ▷ m)
      using less monotone-m seq-comp-presv-def-lift-inv
      by safe
    ultimately show ?thesis
      using less monotone-m
      by metis
  next
    assume card-changed:
      ¬ (card (defer (acc ▷ m) V A p) = card (defer acc V A p))
    with prof
    have card-smaller-for-p:
      𝒮𝒞ℱ-result.electoral-module acc ∧ finite A ⟶
      card (defer (acc ⊳ m) V A p) < card (defer acc V A p)
    using monotone-m order.not-eq-order-implies-strict
          card-mono less.prems seq-comp-def-set-bounded
    unfolding defer-lift-invariance-def
    by metis
  with defer-card-acc defer-card-comp
  have card-changed-for-q:
    defer-lift-invariance acc ⟶
        (∀ q a. a ∈ (defer (acc ⊳ m) V A p) ∧ lifted V A p q a ⟶
            card (defer (acc ⊳ m) V A q) < card (defer acc V A q))
    using lifted-def less
    unfolding defer-lift-invariance-def
    by (metis (no-types, lifting))
  thus ?thesis
  proof (cases)
    assume t-not-satisfied-for-p: ¬ t (acc V A p)
    hence t-not-satisfied-for-q:
      defer-lift-invariance acc ⟶
          (∀ q a. a ∈ (defer (acc ⊳ m) V A p) ∧ lifted V A p q a
            ⟶ ¬ t (acc V A q))
      using monotone-m prof seq-comp-def-set-trans
      unfolding defer-lift-invariance-def
      by metis
    have dli-card-defer:
      defer-lift-invariance (acc ⊳ m) ∧ defer-lift-invariance acc
        ⟶ (∀ q a. a ∈ (defer (acc ⊳ m) V A p) ∧ Profile.lifted V A p q a
              ⟶ card (defer (acc ⊳ m) V A q) ≠ (card (defer acc V A q)))
    proof -
      have
        ∀ m'.
          (¬ defer-lift-invariance m' ∧ 𝒮𝒞ℱ-result.electoral-module m'
          ⟶ (∃ V' A' p' q' a.
                m' V' A' p' ≠ m' V' A' q' ∧ lifted V' A' p' q' a
              ∧ a ∈ defer m' V' A' p'))
          ∧ (defer-lift-invariance m'
            ⟶ 𝒮𝒞ℱ-result.electoral-module m'
              ∧ (∀ V' A' p' q' a.
                m' V' A' p' ≠ m' V' A' q'
              ⟶ lifted V' A' p' q' a ⟶ a ∉ defer m' V' A' p'))
        unfolding defer-lift-invariance-def
        by blast
      thus ?thesis
        using card-changed monotone-m prof seq-comp-def-set-trans
        by (metis (no-types, opaque-lifting))
    qed
    hence dli-def-subset:
      defer-lift-invariance (acc ⊳ m) ∧ defer-lift-invariance acc
        ⟶ (∀ p' a. a ∈ (defer (acc ⊳ m) V A p) ∧ lifted V A p p' a
            ⟶ defer (acc ⊳ m) V A p' ⊂ defer acc V A p')
      using Profile.lifted-def dli-card-defer defer-lift-invariance-def
            monotone-m psubsetI seq-comp-def-set-bounded
      by (metis (no-types, opaque-lifting))
    with t-not-satisfied-for-p
    have rec-step-q:
      defer-lift-invariance (acc ▷ m) ∧ defer-lift-invariance acc
        ⟶ (∀ q a. a ∈ (defer (acc ▷ m) V A p) ∧ lifted V A p q a
            ⟶ loop-comp-helper acc m t V A q =
                  loop-comp-helper (acc ▷ m) m t V A q)
    proof (safe)
      fix
        q :: ('a, 'v) Profile and
        a :: 'a
      assume
        a-in-def-imp-def-subset:
        ∀ q' a'. a' ∈ defer (acc ▷ m) V A p ∧ lifted V A p q' a' ⟶
          defer (acc ▷ m) V A q' ⊂ defer acc V A q' and
        dli-acc: defer-lift-invariance acc and
        a-in-def-seq-acc-m: a ∈ defer (acc ▷ m) V A p and
        lifted-pq-a: lifted V A p q a
      hence defer (acc ▷ m) V A q ⊂ defer acc V A q
        by metis
      moreover have 𝒮𝒞ℱ-result.electoral-module acc
        using dli-acc
        unfolding defer-lift-invariance-def
        by simp
      moreover have ¬ t (acc V A q)
        using dli-acc a-in-def-seq-acc-m lifted-pq-a t-not-satisfied-for-q
        by metis
      ultimately show loop-comp-helper acc m t V A q
                      = loop-comp-helper (acc ▷ m) m t V A q
        using loop-comp-code-helper defer-in-alts finite-subset lifted-pq-a
        unfolding lifted-def
        by (metis (mono-tags, lifting))
    qed
    have rec-step-p:
      𝒮𝒞ℱ-result.electoral-module acc ⟶
         loop-comp-helper acc m t V A p = loop-comp-helper (acc ▷ m) m t V A p
    proof (safe)
      assume emod-acc: 𝒮𝒞ℱ-result.electoral-module acc
      have sound-imp-defer-subset:
        𝒮𝒞ℱ-result.electoral-module m
          ⟶ defer (acc ▷ m) V A p ⊆ defer acc V A p
        using emod-acc prof seq-comp-def-set-bounded
        by blast
      hence card-ineq: card (defer (acc ▷ m) V A p) < card (defer acc V A p)
        using card-changed card-mono less order-neq-le-trans
        unfolding defer-lift-invariance-def
        by metis
      have def-limited-acc:
        profile V (defer acc V A p) (limit-profile (defer acc V A p) p)
        using def-presv-prof emod-acc prof
        by metis
      have defer (acc ▷ m) V A p ⊆ defer acc V A p
        using sound-imp-defer-subset defer-lift-invariance-def monotone-m
        by blast
      hence defer (acc ▷ m) V A p ⊂ defer acc V A p
        using def-limited-acc card-ineq card-psubset less
        by metis
      with def-limited-acc
      show loop-comp-helper acc m t V A p =
            loop-comp-helper (acc ▷ m) m t V A p
        using loop-comp-code-helper t-not-satisfied-for-p less
        by (metis (no-types))
    qed
    show ?thesis
    proof (safe)
      fix
        q :: ('a, 'v) Profile and
        a :: 'a
      assume
        a-in-defer-lch: a ∈ defer (loop-comp-helper acc m t) V A p and
        a-lifted: Profile.lifted V A p q a
      have mod-acc: 𝒮𝒞ℱ-result.electoral-module acc
        using less.prems
        unfolding defer-lift-invariance-def
        by simp
      hence loop-comp-equiv:
        loop-comp-helper acc m t V A p = loop-comp-helper (acc ▷ m) m t V A p
        using rec-step-p
        by blast
      hence a ∈ defer (loop-comp-helper (acc ▷ m) m t) V A p
        using a-in-defer-lch
        by presburger
      moreover have l-inv: defer-lift-invariance (acc ▷ m)
        using less.prems monotone-m voters-determine-m
            seq-comp-presv-def-lift-inv
        by blast
      ultimately have a ∈ defer (acc ▷ m) V A p
        using prof monotone-m in-mono loop-comp-helper-imp-no-def-incr
        unfolding defer-lift-invariance-def
        by (metis (no-types, lifting))
      with l-inv loop-comp-equiv show
        loop-comp-helper acc m t V A p = loop-comp-helper acc m t V A q
      proof −
        assume
           dli-acc-seq-m: defer-lift-invariance (acc ▷ m) and
          a-in-def-seq: a ∈ defer (acc ▷ m) V A p
```

```
          moreover from this have SCF-result.electoral-module (acc ▷ m)
            unfolding defer-lift-invariance-def
            by blast
          moreover have a ∈ defer (loop-comp-helper (acc ▷ m) m t) V A p
            using loop-comp-equiv a-in-defer-lch
            by presburger
          ultimately have
            loop-comp-helper (acc ▷ m) m t V A p
               = loop-comp-helper (acc ▷ m) m t V A q
            using monotone-m mod-acc less a-lifted card-smaller-for-p
                  defer-in-alts infinite-super less
            unfolding lifted-def
            by (metis (no-types))
          moreover have loop-comp-helper acc m t V A q
                        = loop-comp-helper (acc ▷ m) m t V A q
            using dli-acc-seq-m a-in-def-seq less a-lifted rec-step-q
            by blast
          ultimately show ?thesis
            using loop-comp-equiv
            by presburger
        qed
      qed
    next
      assume ¬ ¬t (acc V A p)
      thus ?thesis
        using loop-comp-code-helper less
        unfolding defer-lift-invariance-def
        by metis
    qed
  qed
qed

lemma loop-comp-helper-def-lift-inv:
  fixes
    m acc :: ('a, 'v, 'a Result) Electoral-Module and
    t :: 'a Termination-Condition and
    A :: 'a set and
    V :: 'v set and
    p q :: ('a, 'v) Profile and
    a :: 'a
  assumes
    defer-lift-invariance m and
    voters-determine-election m and
    defer-lift-invariance acc and
    profile V A p and
    lifted V A p q a and
    a ∈ defer (loop-comp-helper acc m t) V A p
  shows (loop-comp-helper acc m t) V A p = (loop-comp-helper acc m t) V A q
  using assms loop-comp-helper-def-lift-inv-helper lifted-def
```

```
      defer-in-alts defer-lift-invariance-def finite-subset
  by metis

lemma lifted-imp-fin-prof:
  fixes
    A :: 'a set and
    V :: 'v set and
    p q :: ('a, 'v) Profile and
    a :: 'a
  assumes lifted V A p q a
  shows finite-profile V A p
  using assms
  unfolding lifted-def
  by simp

lemma loop-comp-helper-presv-def-lift-inv:
  fixes
    m acc :: ('a, 'v, 'a Result) Electoral-Module and
    t :: 'a Termination-Condition
  assumes
    defer-lift-invariance m and
    voters-determine-election m and
    defer-lift-invariance acc
  shows defer-lift-invariance (loop-comp-helper acc m t)
proof (unfold defer-lift-invariance-def, safe)
  show 𝒮𝒞ℱ-result.electoral-module (loop-comp-helper acc m t)
    using loop-comp-helper-imp-partit assms
    unfolding 𝒮𝒞ℱ-result.electoral-module.simps
          defer-lift-invariance-def
    by metis
next
  fix
    A :: 'a set and
    V :: 'v set and
    p q :: ('a, 'v) Profile and
    a :: 'a
  assume
    a ∈ defer (loop-comp-helper acc m t) V A p and
    lifted V A p q a
  thus loop-comp-helper acc m t V A p = loop-comp-helper acc m t V A q
    using lifted-imp-fin-prof loop-comp-helper-def-lift-inv assms
    by metis
qed

lemma loop-comp-presv-non-electing-helper:
  fixes
    m acc :: ('a, 'v, 'a Result) Electoral-Module and
    t :: 'a Termination-Condition and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    n :: nat
  assumes
    non-electing-m: non-electing m and
    non-electing-acc: non-electing acc and
    prof: profile V A p and
    acc-defer-card: n = card (defer acc V A p)
  shows elect (loop-comp-helper acc m t) V A p = {}
  using acc-defer-card non-electing-acc
proof (induct n arbitrary: acc rule: less-induct)
  case (less n)
  thus ?case
  proof (safe)
    fix x :: 'a
    assume
      acc-no-elect:
      (⋀ i acc'. i < card (defer acc V A p) ⟹
        i = card (defer acc' V A p) ⟹ non-electing acc' ⟹
          elect (loop-comp-helper acc' m t) V A p = {}) and
      acc-non-elect: non-electing acc and
      x-in-acc-elect: x ∈ elect (loop-comp-helper acc m t) V A p
    have ∀ m' n'. non-electing m' ∧ non-electing n' ⟶ non-electing (m' ▷ n')
      by simp
    hence seq-acc-m-non-elect: non-electing (acc ▷ m)
      using acc-non-elect non-electing-m
      by blast
    have ∀ i m'.
            i < card (defer acc V A p) ∧ i = card (defer m' V A p) ∧
                non-electing m' ⟶
              elect (loop-comp-helper m' m t) V A p = {}
      using acc-no-elect
      by blast
    hence ∀ m'.
            finite (defer acc V A p) ∧ defer m' V A p ⊂ defer acc V A p ∧
                non-electing m' ⟶
              elect (loop-comp-helper m' m t) V A p = {}
      using psubset-card-mono
      by metis
    hence ¬ t (acc V A p) ∧ defer (acc ▷ m) V A p ⊂ defer acc V A p ∧
                finite (defer acc V A p) ⟶
              elect (loop-comp-helper acc m t) V A p = {}
      using loop-comp-code-helper seq-acc-m-non-elect
      by (metis (no-types))
    moreover have elect acc V A p = {}
      using acc-non-elect prof non-electing-def
      by blast
    ultimately show x ∈ {}
      using loop-comp-code-helper x-in-acc-elect
      by (metis (no-types))
  qed
qed


lemma loop-comp-helper-iter-elim-def-n-helper:
  fixes
    m acc :: ('a, 'v, 'a Result) Electoral-Module and
    t :: 'a Termination-Condition and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    n x :: nat
  assumes
    non-electing-m: non-electing m and
    single-elimination: eliminates 1 m and
    terminate-if-n-left: ∀ r. t r = (card (defer-r r) = x) and
    x-greater-zero: x > 0 and
    prof: profile V A p and
    n-acc-defer-card: n = card (defer acc V A p) and
    n-ge-x: n ≥ x and
    def-card-gt-one: card (defer acc V A p) > 1 and
    acc-nonelect: non-electing acc
  shows card (defer (loop-comp-helper acc m t) V A p) = x
  using n-ge-x def-card-gt-one acc-nonelect n-acc-defer-card
proof (induct n arbitrary: acc rule: less-induct)
  case (less n)
  have mod-acc: 𝒮𝒞ℱ-result.electoral-module acc
    using less
    unfolding non-electing-def
    by metis
  hence step-reduces-defer-set: defer (acc ▷ m) V A p ⊂ defer acc V A p
    using seq-comp-elim-one-red-def-set single-elimination prof less
    by metis
  thus ?case
  proof (cases t (acc V A p))
    case True
    assume term-satisfied: t (acc V A p)
    thus card (defer-r (loop-comp-helper acc m t V A p)) = x
      using loop-comp-code-helper term-satisfied terminate-if-n-left
      by metis
  next
    case False
    hence card-not-eq-x: card (defer acc V A p) ≠ x
      using terminate-if-n-left
      by metis
    have fin-def-acc: finite (defer acc V A p)
      using prof mod-acc less card.infinite not-one-less-zero
      by metis
```

```
hence rec-step:
  loop-comp-helper acc m t V A p = loop-comp-helper (acc ▷ m) m t V A p
  using False step-reduces-defer-set
  by simp
have card-too-big: card (defer acc V A p) > x
  using card-not-eq-x dual-order.order-iff-strict less
  by simp
hence enough-leftover: card (defer acc V A p) > 1
  using x-greater-zero
  by simp
obtain k :: nat where
  new-card-k: k = card (defer (acc ▷ m) V A p)
  by metis
have defer acc V A p ⊆ A
  using defer-in-alts prof mod-acc
  by metis
hence step-profile:
  profile V (defer acc V A p) (limit-profile (defer acc V A p) p)
  using prof limit-profile-sound
  by metis
hence
  card (defer m V (defer acc V A p) (limit-profile (defer acc V A p) p)) =
    card (defer acc V A p) − 1
  using enough-leftover non-electing-m
        single-elimination single-elim-decr-def-card'
  by blast
hence k-card: k = card (defer acc V A p) − 1
  using mod-acc prof new-card-k non-electing-m seq-comp-defers-def-set
  by metis
hence new-card-still-big-enough: x ≤ k
  using card-too-big
  by linarith
show ?thesis
proof (cases x < k)
  case True
  hence 1 < card (defer (acc ▷ m) V A p)
    using new-card-k x-greater-zero
    by linarith
  moreover have k < n
    using step-reduces-defer-set step-profile psubset-card-mono
          new-card-k less fin-def-acc
    by metis
  moreover have 𝒮𝒞ℱ-result.electoral-module (acc ▷ m)
    using mod-acc eliminates-def seq-comp-sound single-elimination
    by metis
  moreover have non-electing (acc ▷ m)
    using less non-electing-m
    by simp
  ultimately have card (defer (loop-comp-helper (acc ▷ m) m t) V A p) = x
```

```
        using new-card-k new-card-still-big-enough less
        by metis
      thus ?thesis
        using rec-step
        by presburger
    next
      case False
      thus ?thesis
        using dual-order.strict-iff-order new-card-k
              new-card-still-big-enough rec-step
              terminate-if-n-left
        by simp
    qed
  qed
qed

lemma loop-comp-helper-iter-elim-def-n:
  fixes
    m acc :: ('a, 'v, 'a Result) Electoral-Module and
    t :: 'a Termination-Condition and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    x :: nat
  assumes
    non-electing m and
    eliminates 1 m and
    ∀ r. (t r) = (card (defer-r r) = x) and
    x > 0 and
    profile V A p and
    card (defer acc V A p) ≥ x and
    non-electing acc
  shows card (defer (loop-comp-helper acc m t) V A p) = x
 using assms gr-implies-not0 le-neq-implies-less less-one linorder-neqE-nat nat-neq-iff
       less-le loop-comp-helper-iter-elim-def-n-helper loop-comp-code-helper
  by (metis (no-types, lifting))

lemma iter-elim-def-n-helper:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    t :: 'a Termination-Condition and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    x :: nat
  assumes
    non-electing-m: non-electing m and
    single-elimination: eliminates 1 m and
    terminate-if-n-left: ∀ r. (t r) = (card (defer-r r) = x) and
    x-greater-zero: x > 0 and
    prof: profile V A p and
    enough-alternatives: card A ≥ x
  shows card (defer (m ⟲_t) V A p) = x
proof (cases)
  assume card A = x
  thus ?thesis
    using terminate-if-n-left
    by simp
next
  assume card-not-x: ¬ card A = x
  thus ?thesis
  proof (cases)
    assume card A < x
    thus ?thesis
      using enough-alternatives not-le
      by blast
  next
    assume ¬ card A < x
    hence card A > x
      using card-not-x
      by linarith
    moreover from this
    have card (defer m V A p) = card A − 1
      using non-electing-m single-elimination single-elim-decr-def-card'
            prof x-greater-zero
      by fastforce
    ultimately have card (defer m V A p) ≥ x
      by linarith
    moreover have (m ⟲_t) V A p = (loop-comp-helper m m t) V A p
      using card-not-x terminate-if-n-left
      by simp
    ultimately show ?thesis
      using non-electing-m prof single-elimination terminate-if-n-left x-greater-zero
            loop-comp-helper-iter-elim-def-n
      by metis
  qed
qed
```

6.5.4 Composition Rules

The loop composition preserves defer-lift-invariance.

```
theorem loop-comp-presv-def-lift-inv[simp]:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    t :: 'a Termination-Condition
  assumes
    defer-lift-invariance m and
    voters-determine-election m
  shows defer-lift-invariance (m ↺_t)
proof (unfold defer-lift-invariance-def, safe)
  have 𝒮𝒞ℱ-result.electoral-module m
    using assms
    unfolding defer-lift-invariance-def
    by simp
  thus 𝒮𝒞ℱ-result.electoral-module (m ↺_t)
    using loop-comp-sound
    by blast
next
  fix
    A :: 'a set and
    V :: 'v set and
    p q :: ('a, 'v) Profile and
    a :: 'a
  assume
    a ∈ defer (m ↺_t) V A p and
    lifted V A p q a
  moreover have
    ∀ p' q' a'. a' ∈ (defer (m ↺_t) V A p') ∧ lifted V A p' q' a' ⟶
        (m ↺_t) V A p' = (m ↺_t) V A q'
    using assms lifted-imp-fin-prof loop-comp-helper-def-lift-inv
          loop-composition.simps defer-module.simps
    by (metis (full-types))
  ultimately show (m ↺_t) V A p = (m ↺_t) V A q
    by metis
qed
```

The loop composition preserves the property non-electing.

```
theorem loop-comp-presv-non-electing[simp]:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    t :: 'a Termination-Condition
  assumes non-electing m
  shows non-electing (m ↺_t)
proof (unfold non-electing-def, safe)
  show 𝒮𝒞ℱ-result.electoral-module (m ↺_t)
    using loop-comp-sound assms
    unfolding non-electing-def
    by metis
next
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assume
    profile V A p and
    a ∈ elect (m ↺_t) V A p
  thus a ∈ {}
    using def-mod-non-electing loop-comp-presv-non-electing-helper
        assms empty-iff loop-comp-code
    unfolding non-electing-def
    by (metis (no-types, lifting))
qed

theorem iter-elim-def-n[simp]:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    t :: 'a Termination-Condition and
    n :: nat
  assumes
    non-electing-m: non-electing m and
    single-elimination: eliminates 1 m and
    terminate-if-n-left: ∀ r. t r = (card (defer-r r) = n) and
    x-greater-zero: n > 0
  shows defers n (m ↺_t)
proof (unfold defers-def, safe)
  show 𝒮𝒞ℱ-result.electoral-module (m ↺_t)
    using loop-comp-sound non-electing-m
    unfolding non-electing-def
    by metis
next
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assume
    n ≤ card A and
    profile V A p
  thus card (defer (m ↺_t) V A p) = n
    using iter-elim-def-n-helper assms
    by metis
qed

end
```

6.6 Maximum Parallel Composition

```
theory Maximum-Parallel-Composition
  imports Basic-Modules/Component-Types/Maximum-Aggregator
        Parallel-Composition
begin
```

This is a family of parallel compositions. It composes a new electoral module from two electoral modules combined with the maximum aggregator. Therein, the two modules each make a decision and then a partition is returned where every alternative receives the maximum result of the two input partitions. This means that, if any alternative is elected by at least one of the modules, then it gets elected, if any non-elected alternative is deferred by at least one of the modules, then it gets deferred, only alternatives rejected by both modules get rejected.

6.6.1 Definition

```
fun maximum-parallel-composition :: ('a, 'v, 'a Result) Electoral-Module ⇒
        ('a, 'v, 'a Result) Electoral-Module ⇒
        ('a, 'v, 'a Result) Electoral-Module where
  maximum-parallel-composition m n =
    (let a = max-aggregator in (m ‖_a n))
```

```
abbreviation max-parallel :: ('a, 'v, 'a Result) Electoral-Module ⇒
        ('a, 'v, 'a Result) Electoral-Module ⇒
        ('a, 'v, 'a Result) Electoral-Module (infix ‖↑ 50) where
  m ‖↑ n ≡ maximum-parallel-composition m n
```

6.6.2 Soundness

```
theorem max-par-comp-sound:
  fixes m n :: ('a, 'v, 'a Result) Electoral-Module
  assumes
    𝒮𝒞ℱ-result.electoral-module m and
    𝒮𝒞ℱ-result.electoral-module n
  shows 𝒮𝒞ℱ-result.electoral-module (m ‖↑ n)
  using assms max-agg-sound par-comp-sound
  unfolding maximum-parallel-composition.simps
  by metis
```

```
lemma voters-determine-max-par-comp:
  fixes m n :: ('a, 'v, 'a Result) Electoral-Module
  assumes
    voters-determine-election m and
    voters-determine-election n
  shows voters-determine-election (m ‖↑ n)
  using max-aggregator.simps assms
  unfolding Let-def maximum-parallel-composition.simps
          parallel-composition.simps
          voters-determine-election.simps
  by presburger
```

6.6.3 Lemmas

```
lemma max-agg-eq-result:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assumes
    module-m: 𝒮𝒞ℱ-result.electoral-module m and
    module-n: 𝒮𝒞ℱ-result.electoral-module n and
    prof-p: profile V A p and
    a-in-A: a ∈ A
  shows mod-contains-result (m ‖↑ n) m V A p a ∨
          mod-contains-result (m ‖↑ n) n V A p a
proof (cases)
  assume a-elect: a ∈ elect (m ‖↑ n) V A p
  hence let (e, r, d) = m V A p;
          (e', r', d') = n V A p in
         a ∈ e ∪ e'
    by auto
  hence a ∈ (elect m V A p) ∪ (elect n V A p)
    by auto
  moreover have
    ∀ m' n' V' A' p' a'.
      mod-contains-result m' n' V' A' p' (a' :: 'a) =
        (𝒮𝒞ℱ-result.electoral-module m'
          ∧ 𝒮𝒞ℱ-result.electoral-module n'
          ∧ profile V' A' p' ∧ a' ∈ A'
          ∧ (a' ∉ elect m' V' A' p' ∨ a' ∈ elect n' V' A' p')
          ∧ (a' ∉ reject m' V' A' p' ∨ a' ∈ reject n' V' A' p')
          ∧ (a' ∉ defer m' V' A' p' ∨ a' ∈ defer n' V' A' p'))
    unfolding mod-contains-result-def
    by simp
  moreover have module-mn: 𝒮𝒞ℱ-result.electoral-module (m ‖↑ n)
    using module-m module-n max-par-comp-sound
    by metis
  moreover have a ∉ defer (m ‖↑ n) V A p
    using module-mn IntI a-elect empty-iff prof-p result-disj
    by (metis (no-types))
  moreover have a ∉ reject (m ‖↑ n) V A p
    using module-mn IntI a-elect empty-iff prof-p result-disj
    by (metis (no-types))
  ultimately show ?thesis
    using assms
    by blast
next
  assume not-a-elect: a ∉ elect (m ‖↑ n) V A p
  thus ?thesis
  proof (cases)
    assume a-in-defer: a ∈ defer (m ‖↑ n) V A p
```

```
thus ?thesis
proof (safe)
  assume not-mod-cont-mn: ¬ mod-contains-result (m ‖↑ n) n V A p a
  have par-emod: ∀ m′ n′.
    SCF-result.electoral-module m′ ∧
    SCF-result.electoral-module n′ ⟶
    SCF-result.electoral-module (m′ ‖↑ n′)
    using max-par-comp-sound
    by blast
  have set-intersect: ∀ a′ A′ A′′. (a′ ∈ A′ ∩ A′′) = (a′ ∈ A′ ∧ a′ ∈ A′′)
    by blast
  have wf-n: well-formed-SCF A (n V A p)
    using prof-p module-n
    unfolding SCF-result.electoral-module.simps
    by blast
  have wf-m: well-formed-SCF A (m V A p)
    using prof-p module-m
    unfolding SCF-result.electoral-module.simps
    by blast
  have e-mod-par: SCF-result.electoral-module (m ‖↑ n)
    using par-emod module-m module-n
    by blast
  hence SCF-result.electoral-module (m ‖_m ax-aggregator n)
    by simp
  hence result-disj-max:
    elect (m ‖_m ax-aggregator n) V A p ∩
        reject (m ‖_m ax-aggregator n) V A p = {} ∧
      elect (m ‖_m ax-aggregator n) V A p ∩
        defer (m ‖_m ax-aggregator n) V A p = {} ∧
      reject (m ‖_m ax-aggregator n) V A p ∩
        defer (m ‖_m ax-aggregator n) V A p = {}
    using prof-p result-disj
    by metis
  have a-not-elect: a ∉ elect (m ‖_m ax-aggregator n) V A p
    using result-disj-max a-in-defer
    by force
  have result-m: (elect m V A p, reject m V A p, defer m V A p) = m V A p
    by auto
  have result-n: (elect n V A p, reject n V A p, defer n V A p) = n V A p
    by auto
  have max-pq:
    ∀ (A′ :: ′a set) m′ n′.
      elect-r (max-aggregator A′ m′ n′) = elect-r m′ ∪ elect-r n′
    by force
  have a ∉ elect (m ‖_m ax-aggregator n) V A p
    using a-not-elect
    by blast
  hence a ∉ elect m V A p ∪ elect n V A p
    using max-pq
  by simp
hence a-not-elect-mn: a ∉ elect m V A p ∧ a ∉ elect n V A p
  by blast
have a-not-mpar-rej: a ∉ reject (m ∥↑ n) V A p
  using result-disj-max a-in-defer
  by fastforce
have mod-cont-res-fg:
  ∀ m′ n′ A′ V′ p′ (a′ :: ′a).
    mod-contains-result m′ n′ V′ A′ p′ a′ =
      (SCF-result.electoral-module m′
        ∧ SCF-result.electoral-module n′
        ∧ profile V′ A′ p′ ∧ a′ ∈ A′
        ∧ (a′ ∈ elect m′ V′ A′ p′ ⟶ a′ ∈ elect n′ V′ A′ p′)
        ∧ (a′ ∈ reject m′ V′ A′ p′ ⟶ a′ ∈ reject n′ V′ A′ p′)
        ∧ (a′ ∈ defer m′ V′ A′ p′ ⟶ a′ ∈ defer n′ V′ A′ p′))
  unfolding mod-contains-result-def
  by simp
have max-agg-res:
  max-aggregator A (elect m V A p, reject m V A p, defer m V A p)
    (elect n V A p, reject n V A p, defer n V A p) =
  (m ∥max-aggregator n) V A p
  by simp
have well-f-max:
  ∀ r′ r″ e′ e″ d′ d″ A′.
    well-formed-SCF A′ (e′, r′, d′) ∧
    well-formed-SCF A′ (e″, r″, d″) ⟶
      reject-r (max-aggregator A′ (e′, r′, d′) (e″, r″, d″)) =
  r′ ∩ r″
  using max-agg-rej-set
  by metis
have e-mod-disj:
  ∀ m′ (V′ :: ′v set) (A′ :: ′a set) p′.
    SCF-result.electoral-module m′ ∧ profile V′ A′ p′
    ⟶ elect m′ V′ A′ p′ ∪ reject m′ V′ A′ p′ ∪ defer m′ V′ A′ p′ = A′
  using result-presv-alts
  by blast
hence e-mod-disj-n: elect n V A p ∪ reject n V A p ∪ defer n V A p = A
  using prof-p module-n
  by metis
have ∀ m′ n′ A′ V′ p′ (b :: ′a).
        mod-contains-result m′ n′ V′ A′ p′ b =
          (SCF-result.electoral-module m′
            ∧ SCF-result.electoral-module n′
            ∧ profile V′ A′ p′ ∧ b ∈ A′
            ∧ (b ∈ elect m′ V′ A′ p′ ⟶ b ∈ elect n′ V′ A′ p′)
            ∧ (b ∈ reject m′ V′ A′ p′ ⟶ b ∈ reject n′ V′ A′ p′)
            ∧ (b ∈ defer m′ V′ A′ p′ ⟶ b ∈ defer n′ V′ A′ p′))
  unfolding mod-contains-result-def
  by simp
```

```
    hence a ∉ defer n V A p
      using a-not-mpar-rej a-in-A e-mod-par module-n not-a-elect
            not-mod-cont-mn prof-p
      by blast
    hence a ∈ reject n V A p
      using a-in-A a-not-elect-mn module-n not-rej-imp-elec-or-defer prof-p
      by metis
    hence a ∉ reject m V A p
      using well-f-max max-agg-res result-m result-n set-intersect
            wf-m wf-n a-not-mpar-rej
      unfolding maximum-parallel-composition.simps
      by (metis (no-types))
    hence a ∉ defer (m ∥↑ n) V A p ∨ a ∈ defer m V A p
        using e-mod-disj prof-p a-in-A module-m a-not-elect-mn
        by blast
    thus mod-contains-result (m ∥↑ n) m V A p a
      using a-not-mpar-rej mod-cont-res-fg e-mod-par prof-p a-in-A
            module-m a-not-elect
      unfolding maximum-parallel-composition.simps
      by metis
  qed
next
  assume not-a-defer: a ∉ defer (m ∥↑ n) V A p
  have el-rej-defer: (elect m V A p, reject m V A p, defer m V A p) = m V A p
    by auto
  from not-a-elect not-a-defer
  have a-reject: a ∈ reject (m ∥↑ n) V A p
    using electoral-mod-defer-elem a-in-A module-m
          module-n prof-p max-par-comp-sound
    by metis
  hence case snd (m V A p) of (r, d) ⇒
          case n V A p of (e′, r′, d′) ⇒
            a ∈ reject-r (max-aggregator A (elect m V A p, r, d) (e′, r′, d′))
    using el-rej-defer
    by force
  hence let (e, r, d) = m V A p;
          (e′, r′, d′) = n V A p in
            a ∈ reject-r (max-aggregator A (e, r, d) (e′, r′, d′))
    unfolding case-prod-unfold
    by simp
  hence let (e, r, d) = m V A p;
          (e′, r′, d′) = n V A p in
            a ∈ A − (e ∪ e′ ∪ d ∪ d′)
    by simp
  hence a ∉ elect m V A p ∪ (defer n V A p ∪ defer m V A p)
    by force
  thus ?thesis
    using mod-contains-result-comm mod-contains-result-def Un-iff
          a-reject prof-p a-in-A module-m module-n max-par-comp-sound
      by (metis (no-types))
  qed
qed

lemma max-agg-rej-iff-both-reject:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assumes
    finite-profile V A p and
    SCF-result.electoral-module m and
    SCF-result.electoral-module n
  shows (a ∈ reject (m ∥↑ n) V A p) =
          (a ∈ reject m V A p ∧ a ∈ reject n V A p)
proof
  assume rej-a: a ∈ reject (m ∥↑ n) V A p
  hence case n V A p of (e, r, d) ⇒
          a ∈ reject-r (max-aggregator A
              (elect m V A p, reject m V A p, defer m V A p) (e, r, d))
    by auto
  hence case snd (m V A p) of (r, d) ⇒
          case n V A p of (e', r', d') ⇒
            a ∈ reject-r (max-aggregator A (elect m V A p, r, d) (e', r', d'))
    by force
  with rej-a
  have let (e, r, d) = m V A p;
          (e', r', d') = n V A p in
            a ∈ reject-r (max-aggregator A (e, r, d) (e', r', d'))
    unfolding prod.case-eq-if
    by simp
  hence let (e, r, d) = m V A p;
            (e', r', d') = n V A p in
              a ∈ A − (e ∪ e' ∪ d ∪ d')
    by simp
  hence
    a ∈ A − (elect m V A p ∪ elect n V A p ∪ defer m V A p ∪ defer n V A p)
    by auto
  thus a ∈ reject m V A p ∧ a ∈ reject n V A p
    using Diff-iff Un-iff electoral-mod-defer-elem assms
    by metis
next
  assume a ∈ reject m V A p ∧ a ∈ reject n V A p
  moreover from this
  have a ∉ elect m V A p ∧ a ∉ defer m V A p
      ∧ a ∉ elect n V A p ∧ a ∉ defer n V A p
    using IntI empty-iff assms result-disj
    by metis
  ultimately show a ∈ reject (m ‖↑ n) V A p
   using DiffD1 max-agg-eq-result mod-contains-result-comm mod-contains-result-def
         reject-not-elected-or-deferred assms
    by (metis (no-types))
qed

lemma max-agg-rej-fst-imp-seq-contained:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assumes
    f-prof: finite-profile V A p and
    module-m: 𝒮𝒞ℱ-result.electoral-module m and
    module-n: 𝒮𝒞ℱ-result.electoral-module n and
    rejected: a ∈ reject n V A p
  shows mod-contains-result m (m ‖↑ n) V A p a
  using assms
proof (unfold mod-contains-result-def, safe)
  show 𝒮𝒞ℱ-result.electoral-module (m ‖↑ n)
    using module-m module-n max-par-comp-sound
    by metis
next
  show a ∈ A
    using f-prof module-n rejected reject-in-alts
    by blast
next
  assume a-in-elect: a ∈ elect m V A p
  hence a-not-reject: a ∉ reject m V A p
    using disjoint-iff-not-equal f-prof module-m result-disj
    by metis
  have reject n V A p ⊆ A
    using f-prof module-n
    by (simp add: reject-in-alts)
  hence a ∈ A
    using in-mono rejected
    by metis
  with a-in-elect a-not-reject
  show a ∈ elect (m ‖↑ n) V A p
    using f-prof max-agg-eq-result module-m module-n rejected
          max-agg-rej-iff-both-reject mod-contains-result-comm
          mod-contains-result-def
    by metis
next
  assume a ∈ reject m V A p
  hence a ∈ reject m V A p ∧ a ∈ reject n V A p
```

```
    using rejected
    by simp
  thus a ∈ reject (m ∥↑ n) V A p
    using f-prof max-agg-rej-iff-both-reject module-m module-n
    by (metis (no-types))
next
  assume a-in-defer: a ∈ defer m V A p
  then obtain d :: 'a where
    defer-a: a = d ∧ d ∈ defer m V A p
    by metis
  have a-not-rej: a ∉ reject m V A p
    using disjoint-iff-not-equal f-prof defer-a module-m result-disj
    by (metis (no-types))
  have
    ∀ m' A' V' p'.
      SCF-result.electoral-module m' ∧ finite A' ∧ finite V' ∧ profile V' A' p'
        ⟶ elect m' V' A' p' ∪ reject m' V' A' p' ∪ defer m' V' A' p' = A'
    using result-presv-alts
    by metis
  hence a ∈ A
    using a-in-defer f-prof module-m
    by blast
  with defer-a a-not-rej
  show a ∈ defer (m ∥↑ n) V A p
    using f-prof max-agg-eq-result max-agg-rej-iff-both-reject
          mod-contains-result-comm mod-contains-result-def
          module-m module-n rejected
    by metis
qed

lemma max-agg-rej-fst-equiv-seq-contained:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assumes
    finite-profile V A p and
    SCF-result.electoral-module m and
    SCF-result.electoral-module n and
    a ∈ reject n V A p
  shows mod-contains-result-sym (m ∥↑ n) m V A p a
  using assms
proof (unfold mod-contains-result-sym-def, safe)
  assume a ∈ reject (m ∥↑ n) V A p
  thus a ∈ reject m V A p
    using assms max-agg-rej-iff-both-reject
    by (metis (no-types))
```

```
next
  have mod-contains-result m (m ∥↑ n) V A p a
    using assms max-agg-rej-fst-imp-seq-contained
    by (metis (full-types))
  thus
    a ∈ elect (m ∥↑ n) V A p ⟹ a ∈ elect m V A p and
    a ∈ defer (m ∥↑ n) V A p ⟹ a ∈ defer m V A p
    using mod-contains-result-comm
    unfolding mod-contains-result-def
    by (metis (full-types), metis (full-types))
next
  show
    𝒮𝒞ℱ-result.electoral-module (m ∥↑ n) and
    a ∈ A
    using assms max-agg-rej-fst-imp-seq-contained
    unfolding mod-contains-result-def
    by (metis (full-types), metis (full-types))
next
  show
    a ∈ elect m V A p ⟹ a ∈ elect (m ∥↑ n) V A p and
    a ∈ reject m V A p ⟹ a ∈ reject (m ∥↑ n) V A p and
    a ∈ defer m V A p ⟹ a ∈ defer (m ∥↑ n) V A p
    using assms max-agg-rej-fst-imp-seq-contained
    unfolding mod-contains-result-def
    by (metis (no-types), metis (no-types), metis (no-types))
qed

lemma max-agg-rej-snd-imp-seq-contained:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assumes
    f-prof: finite-profile V A p and
    module-m: 𝒮𝒞ℱ-result.electoral-module m and
    module-n: 𝒮𝒞ℱ-result.electoral-module n and
    rejected: a ∈ reject m V A p
  shows mod-contains-result n (m ∥↑ n) V A p a
  using assms
proof (unfold mod-contains-result-def, safe)
  show 𝒮𝒞ℱ-result.electoral-module (m ∥↑ n)
    using module-m module-n max-par-comp-sound
    by metis
next
  show a ∈ A
    using f-prof in-mono module-m reject-in-alts rejected
    by (metis (no-types))
```

```
next
  assume a ∈ elect n V A p
  thus a ∈ elect (m ∥↑ n) V A p
    using max-aggregator.simps[of
           - elect m V A p reject m V A p defer m V A p
           elect n V A p reject n V A p defer n V A p]
    by simp
next
  assume a ∈ reject n V A p
  thus a ∈ reject (m ∥↑ n) V A p
    using f-prof max-agg-rej-iff-both-reject module-m module-n rejected
    by metis
next
  assume a ∈ defer n V A p
  moreover have a ∈ A
    using f-prof max-agg-rej-fst-imp-seq-contained module-m rejected
    unfolding mod-contains-result-def
    by metis
  ultimately show a ∈ defer (m ∥↑ n) V A p
    using disjoint-iff-not-equal max-agg-eq-result max-agg-rej-iff-both-reject
          f-prof mod-contains-result-comm mod-contains-result-def
          module-m module-n rejected result-disj
    by (metis (no-types, opaque-lifting))
qed

lemma max-agg-rej-snd-equiv-seq-contained:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    a :: 'a
  assumes
    finite-profile V A p and
    SCF-result.electoral-module m and
    SCF-result.electoral-module n and
    a ∈ reject m V A p
  shows mod-contains-result-sym (m ∥↑ n) n V A p a
  using assms
proof (unfold mod-contains-result-sym-def, safe)
  assume a ∈ reject (m ∥↑ n) V A p
  thus a ∈ reject n V A p
    using assms max-agg-rej-iff-both-reject
    by (metis (no-types))
next
  have mod-contains-result n (m ∥↑ n) V A p a
    using assms max-agg-rej-snd-imp-seq-contained
    by (metis (full-types))
  thus
```

```
    a ∈ elect (m ∥↑ n) V A p ⟹ a ∈ elect n V A p and
    a ∈ defer (m ∥↑ n) V A p ⟹ a ∈ defer n V A p
    using mod-contains-result-comm
    unfolding mod-contains-result-def
    by (metis (full-types), metis (full-types))
next
  show
    𝒮𝒞ℱ-result.electoral-module (m ∥↑ n) and
    a ∈ A
    using assms max-agg-rej-snd-imp-seq-contained
    unfolding mod-contains-result-def
    by (metis (full-types), metis (full-types))
next
  show
    a ∈ elect n V A p ⟹ a ∈ elect (m ∥↑ n) V A p and
    a ∈ reject n V A p ⟹ a ∈ reject (m ∥↑ n) V A p and
    a ∈ defer n V A p ⟹ a ∈ defer (m ∥↑ n) V A p
    using assms max-agg-rej-snd-imp-seq-contained
    unfolding mod-contains-result-def
    by (metis (no-types), metis (no-types), metis (no-types))
qed

lemma max-agg-rej-intersect:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assumes
    𝒮𝒞ℱ-result.electoral-module m and
    𝒮𝒞ℱ-result.electoral-module n and
    profile V A p and
    finite A
  shows reject (m ∥↑ n) V A p = (reject m V A p) ∩ (reject n V A p)
proof −
  have A = (elect m V A p) ∪ (reject m V A p) ∪ (defer m V A p)
      ∧ A = (elect n V A p) ∪ (reject n V A p) ∪ (defer n V A p)
    using assms result-presv-alts
    by metis
  hence A − ((elect m V A p) ∪ (defer m V A p)) = (reject m V A p)
      ∧ A − ((elect n V A p) ∪ (defer n V A p)) = (reject n V A p)
    using assms reject-not-elected-or-deferred
    by fastforce
  hence
    A − ((elect m V A p) ∪ (elect n V A p)
        ∪ (defer m V A p) ∪ (defer n V A p)) =
    (reject m V A p) ∩ (reject n V A p)
    by blast
  hence let (e, r, d) = m V A p;
```

```
        (e′, r′, d′) = n V A p in
          A − (e ∪ e′ ∪ d ∪ d′) = r ∩ r′
    by fastforce
  thus ?thesis
    by auto
qed

lemma dcompat-dec-by-one-mod:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    a :: 'a
  assumes
     disjoint-compatibility m n and
     a ∈ A
   shows
    (∀ p. finite-profile V A p ⟶ mod-contains-result m (m ‖↑ n) V A p a)
      ∨ (∀ p. finite-profile V A p ⟶ mod-contains-result n (m ‖↑ n) V A p a)
 using DiffI assms max-agg-rej-fst-imp-seq-contained max-agg-rej-snd-imp-seq-contained
 unfolding disjoint-compatibility-def
 by metis
```

6.6.4 Composition Rules

Using a conservative aggregator, the parallel composition preserves the property non-electing.

```
theorem conserv-max-agg-presv-non-electing[simp]:
  fixes m n :: ('a, 'v, 'a Result) Electoral-Module
  assumes
    non-electing m and
    non-electing n
  shows non-electing (m ‖↑ n)
  using assms
  by simp
```

Using the max aggregator, composing two compatible electoral modules in parallel preserves defer-lift-invariance.

```
theorem par-comp-def-lift-inv[simp]:
  fixes m n :: ('a, 'v, 'a Result) Electoral-Module
  assumes
    compatible: disjoint-compatibility m n and
    monotone-m: defer-lift-invariance m and
    monotone-n: defer-lift-invariance n
  shows defer-lift-invariance (m ‖↑ n)
proof (unfold defer-lift-invariance-def, safe)
  have mod-m: 𝒮𝒞ℱ-result.electoral-module m
    using monotone-m
    unfolding defer-lift-invariance-def
    by simp
  moreover have mod-n: 𝒮𝒞ℱ-result.electoral-module n
    using monotone-n
    unfolding defer-lift-invariance-def
    by simp
  ultimately show 𝒮𝒞ℱ-result.electoral-module (m ‖↑ n)
    using max-par-comp-sound
    by metis
  fix
    A :: 'a set and
    V :: 'v set and
    p q :: ('a, 'v) Profile and
    a :: 'a
  assume
    defer-a: a ∈ defer (m ‖↑ n) V A p and
    lifted-a: Profile.lifted V A p q a
  hence f-profs: finite-profile V A p ∧ finite-profile V A q
    unfolding lifted-def
    by simp
  from compatible
  obtain B :: 'a set where
    alts: B ⊆ A
        ∧ (∀ b ∈ B. indep-of-alt m V A b ∧
            (∀ p'. finite-profile V A p' ⟶ b ∈ reject m V A p'))
        ∧ (∀ b ∈ A − B. indep-of-alt n V A b ∧
            (∀ p'. finite-profile V A p' ⟶ b ∈ reject n V A p'))
    using f-profs
    unfolding disjoint-compatibility-def
    by (metis (no-types, lifting))
  have ∀ b ∈ A. prof-contains-result (m ‖↑ n) V A p q b
  proof (cases)
    assume a-in-B: a ∈ B
    hence a ∈ reject m V A p
      using alts f-profs
      by blast
    with defer-a
    have defer-n: a ∈ defer n V A p
      using compatible f-profs max-agg-rej-snd-equiv-seq-contained
      unfolding disjoint-compatibility-def mod-contains-result-sym-def
      by metis
    have ∀ b ∈ B. mod-contains-result-sym (m ‖↑ n) n V A p b
      using alts compatible max-agg-rej-snd-equiv-seq-contained f-profs
      unfolding disjoint-compatibility-def
      by metis
    moreover have ∀ b ∈ A. prof-contains-result n V A p q b
    proof (unfold prof-contains-result-def, clarify)
      fix b :: 'a
      assume b-in-A: b ∈ A
    show 𝒮𝒞ℱ-result.electoral-module n ∧ profile V A p
           ∧ profile V A q ∧ b ∈ A ∧
           (b ∈ elect n V A p ⟶ b ∈ elect n V A q) ∧
           (b ∈ reject n V A p ⟶ b ∈ reject n V A q) ∧
           (b ∈ defer n V A p ⟶ b ∈ defer n V A q)
    proof (safe)
      show 𝒮𝒞ℱ-result.electoral-module n
        using monotone-n
        unfolding defer-lift-invariance-def
        by metis
    next
      show
        profile V A p and
        profile V A q and
        b ∈ A
        using f-profs b-in-A
        by (simp, simp, simp)
    next
      show
        b ∈ elect n V A p ⟹ b ∈ elect n V A q and
        b ∈ reject n V A p ⟹ b ∈ reject n V A q and
        b ∈ defer n V A p ⟹ b ∈ defer n V A q
        using defer-n lifted-a monotone-n f-profs
        unfolding defer-lift-invariance-def
        by (metis, metis, metis)
    qed
  qed
  moreover have ∀ b ∈ B. mod-contains-result n (m ‖↑ n) V A q b
    using alts compatible max-agg-rej-snd-imp-seq-contained f-profs
    unfolding disjoint-compatibility-def
    by metis
  ultimately have prof-contains-result-of-comps-for-elems-in-B:
    ∀ b ∈ B. prof-contains-result (m ‖↑ n) V A p q b
    unfolding mod-contains-result-def mod-contains-result-sym-def
              prof-contains-result-def
    by simp
  have ∀ b ∈ A − B. mod-contains-result-sym (m ‖↑ n) m V A p b
   using alts max-agg-rej-fst-equiv-seq-contained monotone-m monotone-n f-profs
    unfolding defer-lift-invariance-def
    by metis
  moreover have ∀ b ∈ A. prof-contains-result m V A p q b
  proof (unfold prof-contains-result-def, clarify)
    fix b :: 'a
    assume b-in-A: b ∈ A
    show 𝒮𝒞ℱ-result.electoral-module m ∧ profile V A p ∧
           profile V A q ∧ b ∈ A ∧
           (b ∈ elect m V A p ⟶ b ∈ elect m V A q) ∧
           (b ∈ reject m V A p ⟶ b ∈ reject m V A q) ∧
           (b ∈ defer m V A p ⟶ b ∈ defer m V A q)
    proof (safe)
      show 𝒮𝒞ℱ-result.electoral-module m
        using monotone-m
        unfolding defer-lift-invariance-def
        by metis
    next
      show
        profile V A p and
        profile V A q and
        b ∈ A
        using f-profs b-in-A
        by (simp, simp, simp)
    next
      show
        b ∈ elect m V A p ⟹ b ∈ elect m V A q and
        b ∈ reject m V A p ⟹ b ∈ reject m V A q and
        b ∈ defer m V A p ⟹ b ∈ defer m V A q
        using alts a-in-B lifted-a lifted-imp-equiv-prof-except-a
        unfolding indep-of-alt-def
        by (metis, metis, metis)
    qed
  qed
  moreover have ∀ b ∈ A − B. mod-contains-result m (m ∥↑ n) V A q b
    using alts max-agg-rej-fst-imp-seq-contained monotone-m monotone-n f-profs
    unfolding defer-lift-invariance-def
    by metis
  ultimately have ∀ b ∈ A − B. prof-contains-result (m ∥↑ n) V A p q b
    unfolding mod-contains-result-def mod-contains-result-sym-def
              prof-contains-result-def
    by simp
  thus ?thesis
    using prof-contains-result-of-comps-for-elems-in-B
    by blast
next
  assume a ∉ B
  hence a-in-set-diff: a ∈ A − B
    using DiffI lifted-a compatible f-profs
    unfolding Profile.lifted-def
    by (metis (no-types, lifting))
  hence reject-n: a ∈ reject n V A p
    using alts f-profs
    by blast
  hence defer-m: a ∈ defer m V A p
    using mod-m mod-n defer-a f-profs max-agg-rej-fst-equiv-seq-contained
    unfolding mod-contains-result-sym-def
    by (metis (no-types))
  have ∀ b ∈ B. mod-contains-result (m ∥↑ n) n V A p b
    using alts compatible f-profs max-agg-rej-snd-imp-seq-contained mod-contains-result-comm
    unfolding disjoint-compatibility-def
    by metis
  have ∀ b ∈ B. mod-contains-result-sym (m ∥↑ n) n V A p b
  using alts max-agg-rej-snd-equiv-seq-contained monotone-m monotone-n f-profs
    unfolding defer-lift-invariance-def
    by metis
  moreover have ∀ b ∈ A. prof-contains-result n V A p q b
  proof (unfold prof-contains-result-def, clarify)
    fix b :: 'a
    assume b-in-A: b ∈ A
    show 𝒮𝒞ℱ-result.electoral-module n ∧ profile V A p ∧
            profile V A q ∧ b ∈ A ∧
            (b ∈ elect n V A p ⟶ b ∈ elect n V A q) ∧
            (b ∈ reject n V A p ⟶ b ∈ reject n V A q) ∧
            (b ∈ defer n V A p ⟶ b ∈ defer n V A q)
    proof (safe)
      show 𝒮𝒞ℱ-result.electoral-module n
        using monotone-n
        unfolding defer-lift-invariance-def
        by metis
    next
      show
        profile V A p and
        profile V A q and
        b ∈ A
        using f-profs b-in-A
        by (simp, simp, simp)
    next
      show
        b ∈ elect n V A p ⟹ b ∈ elect n V A q and
        b ∈ reject n V A p ⟹ b ∈ reject n V A q and
        b ∈ defer n V A p ⟹ b ∈ defer n V A q
        using alts a-in-set-diff lifted-a lifted-imp-equiv-prof-except-a
        unfolding indep-of-alt-def
        by (metis, metis, metis)
    qed
  qed
moreover have ∀ b ∈ B. mod-contains-result n (m ∥↑ n) V A q b
  using alts compatible max-agg-rej-snd-imp-seq-contained f-profs
  unfolding disjoint-compatibility-def
  by metis
ultimately have prof-contains-result-of-comps-for-elems-in-B:
  ∀ b ∈ B. prof-contains-result (m ∥↑ n) V A p q b
    unfolding mod-contains-result-def mod-contains-result-sym-def
              prof-contains-result-def
  by simp
have ∀ b ∈ A − B. mod-contains-result-sym (m ∥↑ n) m V A p b
  using alts max-agg-rej-fst-equiv-seq-contained monotone-m monotone-n f-profs
  unfolding defer-lift-invariance-def
  by metis
  moreover have ∀ b ∈ A. prof-contains-result m V A p q b
  proof (unfold prof-contains-result-def, clarify)
    fix b :: 'a
    assume b-in-A: b ∈ A
    show 𝒮𝒞ℱ-result.electoral-module m ∧ profile V A p
        ∧ profile V A q ∧ b ∈ A
        ∧ (b ∈ elect m V A p ⟶ b ∈ elect m V A q)
        ∧ (b ∈ reject m V A p ⟶ b ∈ reject m V A q)
        ∧ (b ∈ defer m V A p ⟶ b ∈ defer m V A q)
    proof (safe)
      show 𝒮𝒞ℱ-result.electoral-module m
        using monotone-m
        unfolding defer-lift-invariance-def
        by simp
    next
      show
        profile V A p and
        profile V A q and
        b ∈ A
        using f-profs b-in-A
        by (simp, simp, simp)
    next
      show
        b ∈ elect m V A p ⟹ b ∈ elect m V A q and
        b ∈ reject m V A p ⟹ b ∈ reject m V A q and
        b ∈ defer m V A p ⟹ b ∈ defer m V A q
        using defer-m lifted-a monotone-m
        unfolding defer-lift-invariance-def
        by (metis, metis, metis)
    qed
  qed
  moreover have ∀ x ∈ A − B. mod-contains-result m (m ∥↑ n) V A q x
    using alts max-agg-rej-fst-imp-seq-contained monotone-m monotone-n f-profs
    unfolding defer-lift-invariance-def
    by metis
  ultimately have ∀ x ∈ A − B. prof-contains-result (m ∥↑ n) V A p q x
    unfolding mod-contains-result-def mod-contains-result-sym-def
            prof-contains-result-def
    by simp
  thus ?thesis
    using prof-contains-result-of-comps-for-elems-in-B
    by blast
  qed
  thus (m ∥↑ n) V A p = (m ∥↑ n) V A q
    using compatible f-profs eq-alts-in-profs-imp-eq-results max-par-comp-sound
    unfolding disjoint-compatibility-def
    by metis
qed
```

```
lemma par-comp-rej-card:
  fixes
    m n :: ('a, 'v, 'a Result) Electoral-Module and
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    c :: nat
  assumes
    compatible: disjoint-compatibility m n and
    prof: profile V A p and
    fin-A: finite A and
    reject-sum: card (reject m V A p) + card (reject n V A p) = card A + c
  shows card (reject (m ∥↑ n) V A p) = c
proof −
  obtain B :: 'a set where
    alt-set: B ⊆ A
      ∧ (∀ a ∈ B. indep-of-alt m V A a ∧
            (∀ q. profile V A q ⟶ a ∈ reject m V A q))
      ∧ (∀ a ∈ A − B. indep-of-alt n V A a ∧
            (∀ q. profile V A q ⟶ a ∈ reject n V A q))
    using compatible prof
    unfolding disjoint-compatibility-def
    by metis
  have reject-representation:
    reject (m ∥↑ n) V A p = (reject m V A p) ∩ (reject n V A p)
    using prof fin-A compatible max-agg-rej-intersect
    unfolding disjoint-compatibility-def
    by metis
  have 𝒮𝒞ℱ-result.electoral-module m ∧ 𝒮𝒞ℱ-result.electoral-module n
    using compatible
    unfolding disjoint-compatibility-def
    by simp
  hence subsets: (reject m V A p) ⊆ A ∧ (reject n V A p) ⊆ A
    using prof
    by (simp add: reject-in-alts)
  hence finite (reject m V A p) ∧ finite (reject n V A p)
    using rev-finite-subset prof fin-A
    by metis
  hence card-difference:
    card (reject (m ∥↑ n) V A p)
      = card A + c − card ((reject m V A p) ∪ (reject n V A p))
    using card-Un-Int reject-representation reject-sum
    by fastforce
  have ∀ a ∈ A. a ∈ (reject m V A p) ∨ a ∈ (reject n V A p)
    using alt-set prof fin-A
    by blast
  hence A = reject m V A p ∪ reject n V A p
    using subsets
    by force
  thus card (reject (m ‖↑ n) V A p) = c
    using card-difference
    by simp
qed
```

Using the max-aggregator for composing two compatible modules in parallel, whereof the first one is non-electing and defers exactly one alternative, and the second one rejects exactly two alternatives, the composition results in an electoral module that eliminates exactly one alternative.

```
theorem par-comp-elim-one[simp]:
  fixes m n :: ('a, 'v, 'a Result) Electoral-Module
  assumes
    defers-m-one: defers 1 m and
    non-elec-m: non-electing m and
    rejec-n-two: rejects 2 n and
    disj-comp: disjoint-compatibility m n
  shows eliminates 1 (m ‖↑ n)
proof (unfold eliminates-def, safe)
  have 𝒮𝒞ℱ-result.electoral-module m
    using non-elec-m
    unfolding non-electing-def
    by simp
  moreover have 𝒮𝒞ℱ-result.electoral-module n
    using rejec-n-two
    unfolding rejects-def
    by simp
  ultimately show 𝒮𝒞ℱ-result.electoral-module (m ‖↑ n)
    using max-par-comp-sound
    by metis
next
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assume
    min-card-two: 1 < card A and
    prof: profile V A p
  hence card-geq-one: card A ≥ 1
    by presburger
  have fin-A: finite A
    using min-card-two card.infinite not-one-less-zero
    by metis
  have module: 𝒮𝒞ℱ-result.electoral-module m
    using non-elec-m
    unfolding non-electing-def
    by simp
  have elect-card-zero: card (elect m V A p) = 0
    using prof non-elec-m card-eq-0-iff
    unfolding non-electing-def
    by simp
  moreover from card-geq-one
  have def-card-one: card (defer m V A p) = 1
    using defers-m-one module prof fin-A
    unfolding defers-def
    by blast
  ultimately have card-reject-m: card (reject m V A p) = card A − 1
  proof −
    have well-formed-𝒮𝒞ℱ A (elect m V A p, reject m V A p, defer m V A p)
      using prof module
      unfolding 𝒮𝒞ℱ-result.electoral-module.simps
      by simp
    hence card A =
        card (elect m V A p) + card (reject m V A p) + card (defer m V A p)
      using result-count fin-A
      by blast
    thus ?thesis
      using def-card-one elect-card-zero
      by simp
  qed
  have card A ≥ 2
    using min-card-two
    by simp
  hence card (reject n V A p) = 2
    using prof rejec-n-two fin-A
    unfolding rejects-def
    by blast
  moreover from this
  have card (reject m V A p) + card (reject n V A p) = card A + 1
    using card-reject-m card-geq-one
    by linarith
  ultimately show card (reject (m ∥↑ n) V A p) = 1
    using disj-comp prof card-reject-m par-comp-rej-card fin-A
    by blast
qed

end
```

6.7 Elect Composition

```
theory Elect-Composition
  imports Basic-Modules/Elect-Module
          Sequential-Composition
begin
```

The elect composition sequences an electoral module and the elect module. It finalizes the module's decision as it simply elects all their non-rejected alternatives. Thereby, any such elect-composed module induces a proper voting rule in the social choice sense, as all alternatives are either rejected or elected.

6.7.1 Definition

```
fun elector :: ('a, 'v, 'a Result) Electoral-Module ⇒
      ('a, 'v, 'a Result) Electoral-Module where
  elector m = (m ▷ elect-module)
```

6.7.2 Auxiliary Lemmas

```
lemma elector-seqcomp-assoc:
  fixes a b :: ('a, 'v, 'a Result) Electoral-Module
  shows (a ▷ (elector b)) = (elector (a ▷ b))
  unfolding elector.simps elect-module.simps sequential-composition.simps
  using boolean-algebra-cancel.sup2 sup-commute fst-conv snd-conv
  by (metis (no-types, opaque-lifting))
```

6.7.3 Soundness

```
theorem elector-sound[simp]:
  fixes m :: ('a, 'v, 'a Result) Electoral-Module
  assumes 𝒮𝒞ℱ-result.electoral-module m
  shows 𝒮𝒞ℱ-result.electoral-module (elector m)
  using assms elect-mod-sound seq-comp-sound
  unfolding elector.simps
  by metis

lemma voters-determine-elector:
  fixes m :: ('a, 'v, 'a Result) Electoral-Module
  assumes voters-determine-election m
  shows voters-determine-election (elector m)
  using assms elect-mod-only-voters voters-determine-seq-comp
  unfolding elector.simps
  by metis
```

6.7.4 Electing

```
theorem elector-electing[simp]:
  fixes m :: ('a, 'v, 'a Result) Electoral-Module
  assumes
    module-m: 𝒮𝒞ℱ-result.electoral-module m and
    non-block-m: non-blocking m
  shows electing (elector m)
proof –
  have ∀ m'.
```

```
      (¬ electing m′ ∨ 𝒮𝒞ℱ-result.electoral-module m′ ∧
        (∀ A′ V′ p′. (A′ ≠ {} ∧ finite A′ ∧ profile V′ A′ p′)
          ⟶ elect m′ V′ A′ p′ ≠ {})) ∧
        (electing m′ ∨ ¬ 𝒮𝒞ℱ-result.electoral-module m′
         ∨ (∃ A V p. (A ≠ {} ∧ finite A ∧ profile V A p ∧ elect m′ V A p = {})))
  unfolding electing-def
  by blast
hence ∀ m′.
      (¬ electing m′ ∨ 𝒮𝒞ℱ-result.electoral-module m′ ∧
        (∀ A′ V′ p′. (A′ ≠ {} ∧ finite A′ ∧ profile V′ A′ p′)
          ⟶ elect m′ V′ A′ p′ ≠ {})) ∧
      (∃ A V p. (electing m′ ∨ ¬ 𝒮𝒞ℱ-result.electoral-module m′ ∨ A ≠ {}
        ∧ finite A ∧ profile V A p ∧ elect m′ V A p = {}))
  by simp
then obtain
  A :: ('a, 'v, 'a Result) Electoral-Module ⇒ 'a set and
  V :: ('a, 'v, 'a Result) Electoral-Module ⇒ 'v set and
  p :: ('a, 'v, 'a Result) Electoral-Module ⇒ ('a, 'v) Profile where
  electing-mod:
   ∀ m′ :: ('a, 'v, 'a Result) Electoral-Module.
    (¬ electing m′ ∨ 𝒮𝒞ℱ-result.electoral-module m′ ∧
      (∀ A′ V′ p′. (A′ ≠ {} ∧ finite A′ ∧ profile V′ A′ p′)
        ⟶ elect m′ V′ A′ p′ ≠ {})) ∧
      (electing m′ ∨ ¬ 𝒮𝒞ℱ-result.electoral-module m′
      ∨ A m′ ≠ {} ∧ finite (A m′) ∧ profile (V m′) (A m′) (p m′)
                ∧ elect m′ (V m′) (A m′) (p m′) = {})
  by metis
moreover have non-block:
  non-blocking (elect-module :: 'v set ⇒ 'a set ⇒ ('a, 'v) Profile ⇒ 'a Result)
  by (simp add: electing-imp-non-blocking)
moreover obtain
  e :: 'a Result ⇒ 'a set and
  r :: 'a Result ⇒ 'a set and
  d :: 'a Result ⇒ 'a set where
  result: ∀ s. (e s, r s, d s) = s
  using disjoint3.cases
  by (metis (no-types))
moreover from this
have ∀ s. (elect-r s, r s, d s) = s
  by simp
moreover from this
have
  profile (V (elector m)) (A (elector m)) (p (elector m)) ∧ finite (A (elector m))
    ⟶ d (elector m (V (elector m)) (A (elector m)) (p (elector m))) = {}
  by simp
moreover have 𝒮𝒞ℱ-result.electoral-module (elector m)
  using elector-sound module-m
  by simp
moreover from electing-mod result
```

```
  have finite (A (elector m)) ∧
        profile (V (elector m)) (A (elector m)) (p (elector m)) ∧
        elect (elector m) (V (elector m)) (A (elector m)) (p (elector m)) = {} ∧
        d (elector m (V (elector m)) (A (elector m)) (p (elector m))) = {} ∧
        reject (elector m) (V (elector m)) (A (elector m)) (p (elector m)) =
          r (elector m (V (elector m)) (A (elector m)) (p (elector m))) ⟶
            electing (elector m)
    using Diff-empty elector.simps non-block-m snd-conv non-blocking-def
        reject-not-elected-or-deferred non-block seq-comp-presv-non-blocking
    by (metis (mono-tags, opaque-lifting))
  ultimately show ?thesis
    using non-block-m
    unfolding elector.simps
    by auto
qed
```

6.7.5 Composition Rule

If m is defer-Condorcet-consistent, then elector(m) is Condorcet consistent.

```
lemma dcc-imp-cc-elector:
  fixes m :: ('a, 'v, 'a Result) Electoral-Module
  assumes defer-condorcet-consistency m
  shows condorcet-consistency (elector m)
proof (unfold defer-condorcet-consistency-def condorcet-consistency-def, safe)
  show 𝒮𝒞ℱ-result.electoral-module (elector m)
    using assms elector-sound
    unfolding defer-condorcet-consistency-def
    by metis
next
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile and
    w :: 'a
  assume c-win: condorcet-winner V A p w
  have fin-A: finite A
    using condorcet-winner.simps c-win
    by metis
  have fin-V: finite V
    using condorcet-winner.simps c-win
    by metis
  have prof-A: profile V A p
    using c-win
    by simp
  have max-card-w: ∀ y ∈ A − {w}.
        card {i ∈ V. (w, y) ∈ (p i)}
          < card {i ∈ V. (y, w) ∈ (p i)}
    using c-win fin-V
    by simp
```

```
have rej-is-complement:
  reject m V A p = A − (elect m V A p ∪ defer m V A p)
  using double-diff sup-bot.left-neutral Un-upper2 assms fin-A prof-A fin-V
        defer-condorcet-consistency-def elec-and-def-not-rej reject-in-alts
  by (metis (no-types, opaque-lifting))
have subset-in-win-set: elect m V A p ∪ defer m V A p ⊆
    {e ∈ A. e ∈ A ∧ (∀ x ∈ A − {e}.
      card {i ∈ V. (e, x) ∈ p i} < card {i ∈ V. (x, e) ∈ p i})}
proof (safe-step)
  fix x :: 'a
  assume x-in-elect-or-defer: x ∈ elect m V A p ∪ defer m V A p
  hence x-eq-w: x = w
   using Diff-empty Diff-iff assms cond-winner-unique c-win fin-A fin-V insert-iff
         snd-conv sup-bot.left-neutral fst-eqD
    unfolding defer-condorcet-consistency-def
    by (metis (mono-tags, lifting))
  have ∀ x. x ∈ elect m V A p ⟶ x ∈ A
    using fin-A prof-A fin-V assms elect-in-alts in-mono
    unfolding defer-condorcet-consistency-def
    by metis
  moreover have ∀ x. x ∈ defer m V A p ⟶ x ∈ A
    using fin-A prof-A fin-V assms defer-in-alts in-mono
    unfolding defer-condorcet-consistency-def
    by metis
  ultimately have x ∈ A
    using x-in-elect-or-defer
    by auto
  thus x ∈ {e ∈ A. e ∈ A ∧
          (∀ x ∈ A − {e}.
            card {i ∈ V. (e, x) ∈ p i}
              < card {i ∈ V. (x, e) ∈ p i})}
    using x-eq-w max-card-w
    by auto
qed
moreover have
  {e ∈ A. e ∈ A ∧
      (∀ x ∈ A − {e}.
          card {i ∈ V. (e, x) ∈ p i} <
            card {i ∈ V. (x, e) ∈ p i})}
        ⊆ elect m V A p ∪ defer m V A p
proof (safe)
  fix x :: 'a
  assume
    x-not-in-defer: x ∉ defer m V A p and
    x ∈ A and
    ∀ x' ∈ A − {x}.
      card {i ∈ V. (x, x') ∈ p i}
        < card {i ∈ V. (x', x) ∈ p i}
  hence c-win-x: condorcet-winner V A p x
```

```
      using fin-A prof-A fin-V
      by simp
    have (𝒮𝒞ℱ-result.electoral-module m ∧ ¬ defer-condorcet-consistency m ⟶
          (∃ A V rs a. condorcet-winner V A rs a ∧
            m V A rs ≠ ({}, A − defer m V A rs,
            {a ∈ A. condorcet-winner V A rs a})))
        ∧ (defer-condorcet-consistency m ⟶
          (∀ A V rs a. finite A ⟶ finite V ⟶ condorcet-winner V A rs a ⟶
            m V A rs =
      ({}, A − defer m V A rs, {a ∈ A. condorcet-winner V A rs a})))
      unfolding defer-condorcet-consistency-def
      by blast
    hence
      m V A p = ({}, A − defer m V A p, {a ∈ A. condorcet-winner V A p a})
      using c-win-x assms fin-A fin-V
      by blast
    thus x ∈ elect m V A p
      using assms x-not-in-defer fin-A fin-V cond-winner-unique
            defer-condorcet-consistency-def insertCI snd-conv c-win-x
      by (metis (no-types, lifting))
  qed
  ultimately have
    elect m V A p ∪ defer m V A p =
      {e ∈ A. e ∈ A ∧
        (∀ x ∈ A − {e}.
          card {i ∈ V. (e, x) ∈ p i} <
            card {i ∈ V. (x, e) ∈ p i})}
    by blast
  thus elector m V A p =
          ({e ∈ A. condorcet-winner V A p e}, A − elect (elector m) V A p, {})
    using fin-A prof-A fin-V rej-is-complement
    by simp
qed

end
```

6.8 Defer-One Loop Composition

```
theory Defer-One-Loop-Composition
  imports Basic-Modules/Component-Types/Defer-Equal-Condition
          Loop-Composition
          Elect-Composition
begin
```

This is a family of loop compositions. It uses the same module in sequence until either no new decisions are made or only one alternative is remaining in the defer-set. The second family herein uses the above family and subsequently elects the remaining alternative.

```
fun iter :: ('a, 'v, 'a Result) Electoral-Module ⇒
        ('a, 'v, 'a Result) Electoral-Module where
  iter m =
    (let t = defer-equal-condition 1 in
      (m ↺_t))

abbreviation defer-one-loop :: ('a, 'v, 'a Result) Electoral-Module ⇒
        ('a, 'v, 'a Result) Electoral-Module (-↺_∃!d 50) where
  m ↺_∃!d ≡ iter m

fun iter-elect :: ('a, 'v, 'a Result) Electoral-Module ⇒
        ('a, 'v, 'a Result) Electoral-Module where
  iter-elect m = elector (m ↺_∃!d)
```

6.8.1 Soundness

```
theorem defer-one-loop-comp-sound:
  fixes
    m :: ('a, 'v, 'a Result) Electoral-Module and
    t :: 'a Termination-Condition
  assumes 𝒮𝒞ℱ-result.electoral-module m
  shows 𝒮𝒞ℱ-result.electoral-module (m ↺_∃!d)
  using assms loop-comp-sound
  unfolding Defer-One-Loop-Composition.iter.simps
  by metis

end
```

Chapter 7

Voting Rules

7.1 Plurality Rule

```
theory Plurality-Rule
  imports Compositional-Structures/Basic-Modules/Plurality-Module
          Compositional-Structures/Revision-Composition
          Compositional-Structures/Elect-Composition
begin
```

This is a definition of the plurality voting rule as elimination module as well as directly. In the former one, the max operator of the set of the scores of all alternatives is evaluated and is used as the threshold value.

7.1.1 Definition

```
fun plurality-rule :: ('a, 'v, 'a Result) Electoral-Module where
  plurality-rule V A p = elector plurality V A p

fun plurality-rule' :: ('a, 'v, 'a Result) Electoral-Module where
  plurality-rule' V A p =
      (if finite A
        then ({a ∈ A. ∀ x ∈ A. win-count V p x ≤ win-count V p a},
              {a ∈ A. ∃ x ∈ A. win-count V p x > win-count V p a},
              {})
        else (A, {}, {}))

lemma plurality-revision-equiv:
  fixes
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  shows plurality V A p = (plurality-rule↓) V A p
  by fastforce

lemma plurality'-revision-equiv:
  fixes
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  shows plurality' V A p = (plurality-rule'↓) V A p
proof (unfold plurality'.simps revision-composition.simps,
        intro prod-eqI equalityI subsetI prod.sel)
  fix b :: 'a
  assume
    assm: b ∈ fst
            (if finite A
             then ({}, {a ∈ A. ∃ x ∈ A. win-count V p a < win-count V p x},
                   {a ∈ A. ∀ x ∈ A. win-count V p x ≤ win-count V p a})
             else ({}, {}, A))
  have finite A ⟶ b ∈ {}
    using assm
    by force
  moreover have infinite A ⟶ b ∈ {}
    using assm
    by fastforce
  ultimately show
    b ∈ fst ({}, A − elect plurality-rule' V A p, elect plurality-rule' V A p)
    by safe
next
  fix b :: 'a
  assume b ∈ fst ({}, A − elect plurality-rule' V A p, elect plurality-rule' V A p)
  thus b ∈ fst
        (if finite A
         then ({}, {a ∈ A. ∃ x ∈ A. win-count V p a < win-count V p x},
               {a ∈ A. ∀ x ∈ A. win-count V p x ≤ win-count V p a})
         else ({}, {}, A))
    by force
next
  fix b :: 'a
  assume
    assm: b ∈ fst (snd
            (if finite A
             then ({}, {a ∈ A. ∃ x ∈ A. win-count V p a < win-count V p x},
                   {a ∈ A. ∀ x ∈ A. win-count V p x ≤ win-count V p a})
             else ({}, {}, A)))
  have finite A ⟶
    b ∈ A − {a ∈ A. ∀ x ∈ A. win-count V p x ≤ win-count V p a}
    using assm
    by fastforce
  moreover have infinite A ⟶ b ∈ {}
    using assm
    by fastforce
  ultimately show
    b ∈ fst (snd ({}, A − elect plurality-rule' V A p, elect plurality-rule' V A p))
    by force
next
  fix b :: 'a
  assume
    assm: b ∈
      fst (snd ({}, A − elect plurality-rule' V A p, elect plurality-rule' V A p))
  have finite A ⟶
    b ∈ A − {a ∈ A. ∀ x ∈ A. win-count V p x ≤ win-count V p a}
    using assm
    by fastforce
  moreover have infinite A ⟶ b ∈ {}
    using assm
    by fastforce
  ultimately show
    b ∈ fst (snd
        (if finite A
         then ({}, {a ∈ A. ∃ x ∈ A. win-count V p a < win-count V p x},
               {a ∈ A. ∀ x ∈ A. win-count V p x ≤ win-count V p a})
         else ({}, {}, A)))
    using linorder-not-less
    by force
next
  fix b :: 'a
  assume
    assm: b ∈ snd (snd
              (if finite A
               then ({}, {a ∈ A. ∃ x ∈ A. win-count V p a < win-count V p x},
                     {a ∈ A. ∀ x ∈ A. win-count V p x ≤ win-count V p a})
               else ({}, {}, A)))
  have finite A ⟶
    b ∈ A − {a ∈ A. ∃ x ∈ A. win-count V p a < win-count V p x}
    using assm
    by fastforce
  moreover have infinite A ⟶ b ∈ A
    using assm
    by fastforce
  ultimately have b ∈ elect plurality-rule' V A p
    by force
  thus b ∈ snd (snd
    ({}, A − elect plurality-rule' V A p, elect plurality-rule' V A p))
    by simp
next
  fix b :: 'a
  assume assm:
    b ∈ snd (snd ({}, A − elect plurality-rule' V A p, elect plurality-rule' V A p))
  have finite A ⟶
    b ∈ A − {a ∈ A. ∃ x ∈ A. win-count V p a < win-count V p x}
    using assm
    by fastforce
  moreover have infinite A ⟶ b ∈ A
    using assm
    by fastforce
  ultimately show b ∈ snd (snd
            (if finite A
             then ({}, {a ∈ A. ∃ x ∈ A. win-count V p a < win-count V p x},
                  {a ∈ A. ∀ x ∈ A. win-count V p x ≤ win-count V p a})
             else ({}, {}, A)))
    by force
qed

lemma plurality-rule-equiv:
  fixes
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  shows plurality-rule V A p = plurality-rule' V A p
proof (unfold plurality-rule.simps)
  have plurality-rule'-rev-equiv-plurality:
    plurality V A p = (plurality-rule'↓) V A p
    using plurality-mod-equiv plurality'-revision-equiv
    by metis
  have defer (elector plurality) V A p = defer plurality-rule' V A p
    by force
  moreover have reject (elector plurality) V A p = reject plurality-rule' V A p
    using plurality-rule'-rev-equiv-plurality
    by force
  moreover have elect (elector plurality) V A p = elect plurality-rule' V A p
    using plurality-rule'-rev-equiv-plurality
    by force
  ultimately show elector plurality V A p = plurality-rule' V A p
    using prod-eqI
    by (metis (mono-tags, lifting))
qed
```

7.1.2 Soundness

```
theorem plurality-rule-sound[simp]: 𝒮𝒞ℱ-result.electoral-module plurality-rule
  unfolding plurality-rule.simps
  using elector-sound plurality-sound
  by metis

theorem plurality-rule'-sound[simp]: 𝒮𝒞ℱ-result.electoral-module plurality-rule'
proof (unfold 𝒮𝒞ℱ-result.electoral-module.simps, safe)
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  have disjoint3 (
      {a ∈ A. ∀ a' ∈ A. win-count V p a' ≤ win-count V p a},
      {a ∈ A. ∃ a' ∈ A. win-count V p a < win-count V p a'},
      {})
    by auto
  moreover have
    {a ∈ A. ∀ x ∈ A. win-count V p x ≤ win-count V p a} ∪
      {a ∈ A. ∃ x ∈ A. win-count V p a < win-count V p x} = A
    using not-le-imp-less
    by auto
  ultimately show well-formed-𝒮𝒞ℱ A (plurality-rule' V A p)
    by simp
qed

lemma voters-determine-plurality-rule: voters-determine-election plurality-rule
  unfolding plurality-rule.simps
  using voters-determine-elector voters-determine-plurality
  by blast

lemma voters-determine-plurality-rule': voters-determine-election plurality-rule'
proof (unfold voters-determine-election.simps, safe)
  fix
    A :: 'k set and
    V :: 'v set and
    p p' :: ('k, 'v) Profile
  assume ∀ v ∈ V. p v = p' v
  thus plurality-rule' V A p = plurality-rule' V A p'
    using voters-determine-plurality-rule plurality-rule-equiv
    unfolding voters-determine-election.simps
    by (metis (full-types))
qed
```

7.1.3 Electing

```
lemma plurality-rule-elect-non-empty:
  fixes
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assumes
    A ≠ {} and
    finite A
  shows elect plurality-rule V A p ≠ {}
proof
  assume plurality-elect-none: elect plurality-rule V A p = {}
  obtain max :: enat where
    max: max = Max (win-count V p ‘ A)
    by simp
  then obtain a :: 'a where
    max-a: win-count V p a = max ∧ a ∈ A
    using Max-in assms empty-is-image finite-imageI imageE
    by (metis (no-types, lifting))
  hence ∀ a' ∈ A. win-count V p a' ≤ win-count V p a
    using assms max
    by simp
  moreover have a ∈ A
    using max-a
    by simp
  ultimately have a ∈ {a' ∈ A. ∀ c ∈ A. win-count V p c ≤ win-count V p a'}
    by blast
  hence a ∈ elect plurality-rule' V A p
    by simp
  moreover have elect plurality-rule' V A p = defer plurality V A p
    using plurality-revision-equiv plurality-rule-equiv snd-conv
    unfolding revision-composition.simps
    by (metis (no-types, opaque-lifting))
  ultimately have a ∈ defer plurality V A p
    by blast
  hence a ∈ elect plurality-rule V A p
    by simp
  thus False
    using plurality-elect-none all-not-in-conv
    by metis
qed

lemma plurality-rule'-elect-non-empty:
  fixes
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assumes
    A ≠ {} and
    profile V A p and
    finite A
  shows elect plurality-rule' V A p ≠ {}
  using assms plurality-rule-elect-non-empty plurality-rule-equiv
  by metis
```

The plurality module is electing.

```
theorem plurality-rule-electing[simp]: electing plurality-rule
proof (unfold electing-def, safe)
  show 𝒮𝒞ℱ-result.electoral-module plurality-rule
    using plurality-rule-sound
    by simp
next
  fix
    A :: 'b set and
    V :: 'a set and
    p :: ('b, 'a) Profile and
    a :: 'b
  assume
    fin-A: finite A and
    prof-p: profile V A p and
    elect-none: elect plurality-rule V A p = {} and
    a-in-A: a ∈ A
  have ∀ B W q. B ≠ {} ∧ finite B ∧ profile W B q
        ⟶ elect plurality-rule W B q ≠ {}
    using plurality-rule-elect-non-empty
    by (metis (no-types))
  hence empty-A: A = {}
    using fin-A prof-p elect-none
    by (metis (no-types))
  thus a ∈ {}
    using a-in-A
    by simp
qed

theorem plurality-rule'-electing[simp]: electing plurality-rule'
proof (unfold electing-def, safe)
  show 𝒮𝒞ℱ-result.electoral-module plurality-rule'
    using plurality-rule'-sound
    by metis
next
  fix
    A :: 'b set and
    V :: 'a set and
    p :: ('b, 'a) Profile and
    a :: 'b
  assume
    fin-A: finite A and
    prof-p: profile V A p and
    no-elect': elect plurality-rule' V A p = {} and
    a-in-A: a ∈ A
  have A-nonempty: A ≠ {}
    using a-in-A
    by blast
  have elect plurality-rule V A p = {}
    using no-elect' plurality-rule-equiv
    by metis
  moreover have elect plurality-rule V A p ≠ {}
   using fin-A prof-p A-nonempty plurality-rule'-elect-non-empty plurality-rule-equiv
    by metis
  ultimately show a ∈ {}
    by force
qed
```

7.1.4 Properties

```
lemma plurality-rule-inv-mono-eq:
  fixes
    A :: 'a set and
    V :: 'v set and
    p q :: ('a, 'v) Profile and
    a :: 'a
  assumes
    elect-a: a ∈ elect plurality-rule V A p and
    lift-a: lifted V A p q a
  shows elect plurality-rule V A q = elect plurality-rule V A p
          ∨ elect plurality-rule V A q = {a}
proof −
  have a ∈ elect (elector plurality) V A p
    using elect-a
    by simp
  moreover have eq-p: elect (elector plurality) V A p = defer plurality V A p
    by simp
  ultimately have a ∈ defer plurality V A p
    by blast
  hence defer plurality V A q = defer plurality V A p
          ∨ defer plurality V A q = {a}
    using lift-a plurality-def-inv-mono-alts
    by metis
  moreover have elect (elector plurality) V A q = defer plurality V A q
    by simp
  ultimately show
    elect plurality-rule V A q = elect plurality-rule V A p
      ∨ elect plurality-rule V A q = {a}
    using eq-p
    by simp
qed

lemma plurality-rule'-inv-mono-eq:
  fixes
    A :: 'a set and
    V :: 'v set and
    p q :: ('a, 'v) Profile and
    a :: 'a
  assumes
    a ∈ elect plurality-rule' V A p and
    lifted V A p q a
  shows elect plurality-rule' V A q = elect plurality-rule' V A p
          ∨ elect plurality-rule' V A q = {a}
  using assms plurality-rule-equiv plurality-rule-inv-mono-eq
  by (metis (no-types))
```

The plurality rule is invariant-monotone.

```
theorem plurality-rule-inv-mono[simp]: invariant-monotonicity plurality-rule
```

```
proof (unfold invariant-monotonicity-def, intro conjI impI allI)
  show 𝒮𝒞ℱ-result.electoral-module plurality-rule
    using plurality-rule-sound
    by metis
next
  fix
    A :: 'b set and
    V :: 'a set and
    p q :: ('b, 'a) Profile and
    a :: 'b
  assume a ∈ elect plurality-rule V A p ∧ Profile.lifted V A p q a
  thus elect plurality-rule V A q = elect plurality-rule V A p
        ∨ elect plurality-rule V A q = {a}
    using plurality-rule-inv-mono-eq
    by metis
qed

theorem plurality-rule′-inv-mono[simp]: invariant-monotonicity plurality-rule′
proof –
  have (plurality-rule::('k, 'v, 'k Result) Electoral-Module) = plurality-rule′
    using plurality-rule-equiv
    by blast
  thus ?thesis
    using plurality-rule-inv-mono
    by (metis (full-types))
qed
```

(Weak) Monotonicity

```
theorem plurality-rule-monotone: monotonicity plurality-rule
proof (unfold monotonicity-def, safe)
  show 𝒮𝒞ℱ-result.electoral-module plurality-rule
    using plurality-rule-sound
    by (metis (no-types))
next
  fix
    A :: 'b set and
    V :: 'a set and
    p q :: ('b, 'a) Profile and
    a :: 'b
  assume
    a ∈ elect plurality-rule V A p and
    Profile.lifted V A p q a
  thus a ∈ elect plurality-rule V A q
    using insertI1 plurality-rule-inv-mono-eq
    by (metis (no-types))
qed

end
```

7.2 Borda Rule

```
theory Borda-Rule
  imports Compositional-Structures/Basic-Modules/Borda-Module
      Compositional-Structures/Basic-Modules/Component-Types/Votewise-Distance-Rationalization
        Compositional-Structures/Elect-Composition
begin
```

This is the Borda rule. On each ballot, each alternative is assigned a score that depends on how many alternatives are ranked below. The sum of all such scores for an alternative is hence called their Borda score. The alternative with the highest Borda score is elected.

7.2.1 Definition

```
fun borda-rule :: ('a, 'v, 'a Result) Electoral-Module where
  borda-rule V A p = elector borda V A p

fun borda-rule_ℛ :: ('a, 'v :: wellorder, 'a Result) Electoral-Module where
  borda-rule_ℛ V A p = swap-ℛ unanimity V A p
```

7.2.2 Soundness

```
theorem borda-rule-sound: 𝒮𝒞ℱ-result.electoral-module borda-rule
  unfolding borda-rule.simps
  using elector-sound borda-sound
  by metis

theorem borda-rule_ℛ-sound: 𝒮𝒞ℱ-result.electoral-module borda-rule_ℛ
  unfolding borda-rule_ℛ.simps swap-ℛ.simps
  using 𝒮𝒞ℱ-result.ℛ-sound
  by metis
```

7.2.3 Anonymity

```
theorem borda-rule_ℛ-anonymous: 𝒮𝒞ℱ-result.anonymity borda-rule_ℛ
proof (unfold borda-rule_ℛ.simps swap-ℛ.simps)
  let ?swap-dist = votewise-distance swap l-one
  from l-one-is-sym
  have distance-anonymity ?swap-dist
    using symmetric-norm-imp-distance-anonymous[of l-one]
    by simp
  with unanimity-anonymous
  show 𝒮𝒞ℱ-result.anonymity (𝒮𝒞ℱ-result.distance-ℛ ?swap-dist unanimity)
    using 𝒮𝒞ℱ-result.anonymous-distance-and-consensus-imp-rule-anonymity
  by metis
qed

end
```

7.3 Pairwise Majority Rule

```
theory Pairwise-Majority-Rule
  imports Compositional-Structures/Basic-Modules/Condorcet-Module
          Compositional-Structures/Defer-One-Loop-Composition
begin
```

This is the pairwise majority rule, a voting rule that implements the Condorcet criterion, i.e., it elects the Condorcet winner if it exists, otherwise a tie remains between all alternatives.

7.3.1 Definition

```
fun pairwise-majority-rule :: ('a, 'v, 'a Result) Electoral-Module where
  pairwise-majority-rule V A p = elector condorcet V A p

fun condorcet' :: ('a, 'v, 'a Result) Electoral-Module where
  condorcet' V A p = ((min-eliminator condorcet-score) ↻∃!d) V A p

fun pairwise-majority-rule' :: ('a, 'v, 'a Result) Electoral-Module where
  pairwise-majority-rule' V A p = iter-elect condorcet' V A p
```

7.3.2 Soundness

```
theorem pairwise-majority-rule-sound: SCℱ-result.electoral-module pairwise-majority-rule
  unfolding pairwise-majority-rule.simps
  using condorcet-sound elector-sound
  by metis

theorem condorcet'-sound: SCℱ-result.electoral-module condorcet'
  using Defer-One-Loop-Composition.iter.elims loop-comp-sound min-elim-sound
  unfolding condorcet'.simps loop-comp-sound
  by metis

theorem pairwise-majority-rule'-sound: SCℱ-result.electoral-module pairwise-majority-rule'
  unfolding pairwise-majority-rule'.simps
  using condorcet'-sound elector-sound iter.simps iter-elect.simps loop-comp-sound
  by metis
```

7.3.3 Condorcet Consistency

theorem *condorcet-condorcet*: *condorcet-consistency pairwise-majority-rule*
proof (*unfold pairwise-majority-rule.simps*)
  show *condorcet-consistency* (*elector condorcet*)
    using *condorcet-is-dcc dcc-imp-cc-elector*
    by *metis*
qed

end

7.4 Copeland Rule

theory *Copeland-Rule*
  imports *Compositional-Structures/Basic-Modules/Copeland-Module*
    Compositional-Structures/Elect-Composition
begin

This is the Copeland voting rule. The idea is to elect the alternatives with the highest difference between the amount of simple-majority wins and the amount of simple-majority losses.

7.4.1 Definition

fun *copeland-rule* :: (*'a*, *'v*, *'a Result*) *Electoral-Module* **where**
  copeland-rule V A p = *elector copeland V A p*

7.4.2 Soundness

theorem *copeland-rule-sound*: $\mathcal{SCF}$*-result.electoral-module copeland-rule*
  unfolding *copeland-rule.simps*
  using *elector-sound copeland-sound*
  by *metis*

7.4.3 Condorcet Consistency

theorem *copeland-condorcet*: *condorcet-consistency copeland-rule*
proof (*unfold copeland-rule.simps*)
  show *condorcet-consistency* (*elector copeland*)
    using *copeland-is-dcc dcc-imp-cc-elector*
    by *metis*
qed

end

7.5 Minimax Rule

```
theory Minimax-Rule
  imports Compositional-Structures/Basic-Modules/Minimax-Module
          Compositional-Structures/Elect-Composition
begin
```

This is the Minimax voting rule. It elects the alternatives with the highest Minimax score.

7.5.1 Definition

```
fun minimax-rule :: ('a, 'v, 'a Result) Electoral-Module where
  minimax-rule V A p = elector minimax V A p
```

7.5.2 Soundness

```
theorem minimax-rule-sound: 𝒮𝒞ℱ-result.electoral-module minimax-rule
  unfolding minimax-rule.simps
  using elector-sound minimax-sound
  by metis
```

7.5.3 Condorcet Consistency

```
theorem minimax-condorcet: condorcet-consistency minimax-rule
proof (unfold minimax-rule.simps)
  show condorcet-consistency (elector minimax)
    using minimax-is-dcc dcc-imp-cc-elector
    by metis
qed

end
```

7.6 Black's Rule

```
theory Blacks-Rule
  imports Pairwise-Majority-Rule
          Borda-Rule
begin
```

This is Black's voting rule. It is composed of a function that determines the Condorcet winner, i.e., the Pairwise Majority rule, and the Borda rule. Whenever there exists no Condorcet winner, it elects the choice made by the Borda rule, otherwise the Condorcet winner is elected.

7.6.1 Definition

```
fun black :: ('a, 'v, 'a Result) Electoral-Module where
  black A p = (condorcet ▷ borda) A p

fun blacks-rule :: ('a, 'v, 'a Result) Electoral-Module where
  blacks-rule A p = elector black A p
```

7.6.2 Soundness

```
theorem blacks-sound: 𝒮𝒞ℱ-result.electoral-module black
  unfolding black.simps
  using seq-comp-sound condorcet-sound borda-sound
  by metis

theorem blacks-rule-sound: 𝒮𝒞ℱ-result.electoral-module blacks-rule
  unfolding blacks-rule.simps
  using blacks-sound elector-sound
  by metis
```

7.6.3 Condorcet Consistency

```
theorem black-is-dcc: defer-condorcet-consistency black
  unfolding black.simps
 using condorcet-is-dcc borda-mod-non-blocking borda-mod-non-electing seq-comp-dcc
  by metis

theorem black-condorcet: condorcet-consistency blacks-rule
  unfolding blacks-rule.simps
  using black-is-dcc dcc-imp-cc-elector
  by metis

end
```

7.7 Nanson-Baldwin Rule

```
theory Nanson-Baldwin-Rule
  imports Compositional-Structures/Basic-Modules/Borda-Module
          Compositional-Structures/Defer-One-Loop-Composition
begin
```

This is the Nanson-Baldwin voting rule. It excludes alternatives with the lowest Borda score from the set of possible winners and then adjusts the Borda score to the new (remaining) set of still eligible alternatives.

7.7.1 Definition

fun *nanson-baldwin-rule* :: (*'a*, *'v*, *'a Result*) *Electoral-Module* **where**
nanson-baldwin-rule A p =
((*min-eliminator borda-score*) $\circlearrowleft_{\exists !d}$) *A p*

7.7.2 Soundness

theorem *nanson-baldwin-rule-sound*: *𝒮𝒞ℱ-result.electoral-module nanson-baldwin-rule*
using *min-elim-sound loop-comp-sound*
unfolding *nanson-baldwin-rule.simps Defer-One-Loop-Composition.iter.simps*
by *metis*

end

7.8 Classic Nanson Rule

theory *Classic-Nanson-Rule*
imports *Compositional-Structures/Basic-Modules/Borda-Module*
Compositional-Structures/Defer-One-Loop-Composition
begin

This is the classic Nanson's voting rule, i.e., the rule that was originally invented by Nanson, but not the Nanson-Baldwin rule. The idea is similar, however, as alternatives with a Borda score less or equal than the average Borda score are excluded. The Borda scores of the remaining alternatives are hence adjusted to the new set of (still) eligible alternatives.

7.8.1 Definition

fun *classic-nanson-rule* :: (*'a*, *'v*, *'a Result*) *Electoral-Module* **where**
classic-nanson-rule V A p =
((*leq-average-eliminator borda-score*) $\circlearrowleft_{\exists !d}$) *V A p*

7.8.2 Soundness

theorem *classic-nanson-rule-sound*: *𝒮𝒞ℱ-result.electoral-module classic-nanson-rule*
using *leq-avg-elim-sound loop-comp-sound*
unfolding *classic-nanson-rule.simps Defer-One-Loop-Composition.iter.simps*
by *metis*

end

7.9 Schwartz Rule

```
theory Schwartz-Rule
  imports Compositional-Structures/Basic-Modules/Borda-Module
          Compositional-Structures/Defer-One-Loop-Composition
begin
```

This is the Schwartz voting rule. Confusingly, it is sometimes also referred as Nanson's rule. The Schwartz rule proceeds as in the classic Nanson's rule, but excludes alternatives with a Borda score that is strictly less than the average Borda score.

7.9.1 Definition

```
fun schwartz-rule :: ('a, 'v, 'a Result) Electoral-Module where
  schwartz-rule V A p =
    ((less-average-eliminator borda-score) ↺∃!d) V A p
```

7.9.2 Soundness

```
theorem schwartz-rule-sound: SCℱ-result.electoral-module schwartz-rule
  using less-avg-elim-sound loop-comp-sound
  unfolding schwartz-rule.simps Defer-One-Loop-Composition.iter.simps
  by metis

end
```

7.10 Sequential Majority Comparison

```
theory Sequential-Majority-Comparison
  imports Plurality-Rule
          Compositional-Structures/Drop-And-Pass-Compatibility
          Compositional-Structures/Revision-Composition
          Compositional-Structures/Maximum-Parallel-Composition
          Compositional-Structures/Defer-One-Loop-Composition
begin
```

Sequential majority comparison compares two alternatives by plurality voting. The loser gets rejected, and the winner is compared to the next alternative. This process is repeated until only a single alternative is left, which is then elected.

7.10.1 Definition

```
fun smc :: 'a Preference-Relation ⇒ ('a, 'v, 'a Result) Electoral-Module where
```

```
smc x V A p =
    ((elector ((((pass-module 2 x) ▷ ((plurality-rule↓) ▷ (pass-module 1 x))) ‖↑
    (drop-module 2 x)) ↻∃!d)) V A p)
```

7.10.2 Soundness

As all base components are electoral modules (, aggregators, or termination conditions), and all used compositional structures create electoral modules, sequential majority comparison unsurprisingly is an electoral module.

```
theorem smc-sound:
  fixes x :: 'a Preference-Relation
  shows SCF-result.electoral-module (smc x)
proof (unfold SCF-result.electoral-module.simps well-formed-SCF.simps, safe)
  fix
    A :: 'a set and
    V :: 'v set and
    p :: ('a, 'v) Profile
  assume profile V A p
  thus
    disjoint3 (smc x V A p) and
    set-equals-partition A (smc x V A p)
    unfolding iter.simps smc.simps elector.simps
    using drop-mod-sound elect-mod-sound loop-comp-sound max-par-comp-sound
          pass-mod-sound plurality-rule-sound rev-comp-sound seq-comp-sound
  by (metis (no-types) seq-comp-presv-disj, metis (no-types) seq-comp-presv-alts)
qed
```

7.10.3 Electing

The sequential majority comparison electoral module is electing. This property is needed to convert electoral modules to a social choice function. Apart from the very last proof step, it is a part of the monotonicity proof below.

```
theorem smc-electing:
  fixes x :: 'a Preference-Relation
  assumes linear-order x
  shows electing (smc x)
proof –
  let ?pass2 = pass-module 2 x
  let ?tie-breaker = (pass-module 1 x)
  let ?plurality-defer = (plurality-rule↓) ▷ ?tie-breaker
  let ?compare-two = ?pass2 ▷ ?plurality-defer
  let ?drop2 = drop-module 2 x
  let ?eliminator = ?compare-two ‖↑ ?drop2
  let ?loop =
    let t = defer-equal-condition 1 in (?eliminator ↻t)

  have 00011: non-electing (plurality-rule↓)
    using plurality-rule-sound rev-comp-non-electing
    by metis
  have 00012: non-electing ?tie-breaker
    using assms
    by simp
  have 00013: defers 1 ?tie-breaker
    using assms pass-one-mod-def-one
    by simp
  have 20000: non-blocking (plurality-rule↓)
    by simp
  have 0020: disjoint-compatibility ?pass2 ?drop2
    using assms
    by simp
  have 1000: non-electing ?pass2
    using assms
    by simp
  have 1001: non-electing ?plurality-defer
    using 00011 00012 seq-comp-presv-non-electing
    by blast
  have 2000: non-blocking ?pass2
    using assms
    by simp
  have 2001: defers 1 ?plurality-defer
    using 20000 00011 00013 seq-comp-def-one
    by blast
  have 002: disjoint-compatibility ?compare-two ?drop2
    using assms 0020 disj-compat-seq pass-mod-sound plurality-rule-sound
        rev-comp-sound seq-comp-sound voters-determine-pass-mod
        voters-determine-plurality-rule voters-determine-seq-comp
        voters-determine-rev-comp
    by metis
  have 100: non-electing ?compare-two
    using 1000 1001 seq-comp-presv-non-electing
    by simp
  have 101: non-electing ?drop2
    using assms
    by simp
  have 102: agg-conservative max-aggregator
    by simp
  have 200: defers 1 ?compare-two
    using 2000 1000 2001 seq-comp-def-one
    by simp
  have 201: rejects 2 ?drop2
    using assms
    by simp
  have 10: non-electing ?eliminator
    using 100 101 102 conserv-max-agg-presv-non-electing
    by blast
  have 20: eliminates 1 ?eliminator
    using 200 100 201 002 par-comp-elim-one
```

```
    by simp
  have 2: defers 1 ?loop
    using 10 20 iter-elim-def-n zero-less-one prod.exhaust-sel
          defer-equal-condition.simps
    by metis
  have 3: electing elect-module
    by simp
  show ?thesis
    using 2 3 assms seq-comp-electing smc-sound
    unfolding Defer-One-Loop-Composition.iter.simps
            smc.simps elector.simps electing-def
    by metis
qed
```

7.10.4 (Weak) Monotonicity

The following proof is a fully modular proof for weak monotonicity of sequential majority comparison. It is composed of many small steps.

```
theorem smc-monotone:
  fixes x :: 'a Preference-Relation
  assumes linear-order x
  shows monotonicity (smc x)
proof –
  let ?pass2 = pass-module 2 x
  let ?tie-breaker = pass-module 1 x
  let ?plurality-defer = (plurality-rule↓) ▷ ?tie-breaker
  let ?compare-two = ?pass2 ▷ ?plurality-defer
  let ?drop2 = drop-module 2 x
  let ?eliminator = ?compare-two ∥↑ ?drop2
  let ?loop =
    let t = defer-equal-condition 1 in (?eliminator ↻t)

  have 00010: defer-invariant-monotonicity (plurality-rule↓)
    by simp
  have 00011: non-electing (plurality-rule↓)
    using rev-comp-non-electing plurality-rule-sound
    by blast
  have 00012: non-electing ?tie-breaker
    using assms
    by simp
  have 00013: defers 1 ?tie-breaker
    using assms pass-one-mod-def-one
    by simp
  have 00014: defer-monotonicity ?tie-breaker
    using assms
    by simp
  have 20000: non-blocking (plurality-rule↓)
    by simp
  have 0000: defer-lift-invariance ?pass2
  using assms
  by simp
have 0001: defer-lift-invariance ?plurality-defer
  using 00010 00012 00013 00014 def-inv-mono-imp-def-lift-inv
  unfolding pass-module.simps voters-determine-election.simps
  by blast
have 0020: disjoint-compatibility ?pass2 ?drop2
  using assms
  by simp
have 1000: non-electing ?pass2
  using assms
  by simp
have 1001: non-electing ?plurality-defer
  using 00011 00012 seq-comp-presv-non-electing
  by blast
have 2000: non-blocking ?pass2
  using assms
  by simp
have 2001: defers 1 ?plurality-defer
  using 20000 00011 00013 seq-comp-def-one
  by blast
have 000: defer-lift-invariance ?compare-two
  using 0000 0001 seq-comp-presv-def-lift-inv
      voters-determine-plurality-rule voters-determine-pass-mod
      voters-determine-rev-comp voters-determine-seq-comp
  by blast
have 001: defer-lift-invariance ?drop2
  using assms
  by simp
have 002: disjoint-compatibility ?compare-two ?drop2
  using assms 0020 disj-compat-seq pass-mod-sound plurality-rule-sound
    voters-determine-pass-mod rev-comp-sound seq-comp-sound voters-determine-seq-comp
    voters-determine-plurality-rule voters-determine-pass-mod voters-determine-rev-comp
  by metis
have 100: non-electing ?compare-two
  using 1000 1001 seq-comp-presv-non-electing
  by simp
have 101: non-electing ?drop2
  using assms
  by simp
have 102: agg-conservative max-aggregator
  by simp
have 200: defers 1 ?compare-two
  using 2000 1000 2001 seq-comp-def-one
  by simp
have 201: rejects 2 ?drop2
  using assms
  by simp
have 00: defer-lift-invariance ?eliminator
```

```
    using 000 001 002 par-comp-def-lift-inv
    by blast
  have 10: non-electing ?eliminator
    using 100 101 conserv-max-agg-presv-non-electing
    by blast
  have 20: eliminates 1 ?eliminator
    using 200 100 201 002 par-comp-elim-one
    by simp
  have 0: defer-lift-invariance ?loop
    using 00 loop-comp-presv-def-lift-inv
      voters-determine-plurality-rule voters-determine-pass-mod voters-determine-drop-mod
      voters-determine-rev-comp voters-determine-seq-comp voters-determine-max-par-comp
    by metis
  have 1: non-electing ?loop
    using 10 loop-comp-presv-non-electing
    by simp
  have 2: defers 1 ?loop
   using 10 20 iter-elim-def-n prod.exhaust-sel zero-less-one defer-equal-condition.simps
    by metis
  have 3: electing elect-module
    by simp
  show ?thesis
    using 0 1 2 3 assms seq-comp-mono
    unfolding Electoral-Module.monotonicity-def elector.simps
            Defer-One-Loop-Composition.iter.simps
            smc-sound smc.simps
    by (metis (full-types))
qed

end
```

7.11 Kemeny Rule

```
theory Kemeny-Rule
  imports
   Compositional-Structures/Basic-Modules/Component-Types/Votewise-Distance-Rationalization
   Compositional-Structures/Basic-Modules/Component-Types/Distance-Rationalization-Symmetry
begin
```

This is the Kemeny rule. It creates a complete ordering of alternatives and evaluates each ordering of the alternatives in terms of the sum of preference reversals on each ballot that would have to be performed in order to produce that transitive ordering. The complete ordering which requires the fewest preference reversals is the final result of the method.

7.11.1 Definition

```
fun kemeny-rule :: ('a, 'v :: wellorder, 'a Result) Electoral-Module where
  kemeny-rule V A p = swap-ℛ strong-unanimity V A p
```

7.11.2 Soundness

```
theorem kemeny-rule-sound: 𝒮𝒞ℱ-result.electoral-module kemeny-rule
  unfolding kemeny-rule.simps swap-ℛ.simps
  using 𝒮𝒞ℱ-result.ℛ-sound
  by metis
```

7.11.3 Anonymity

```
theorem kemeny-rule-anonymous: 𝒮𝒞ℱ-result.anonymity kemeny-rule
proof (unfold kemeny-rule.simps swap-ℛ.simps)
  let ?swap-dist = votewise-distance swap l-one
  have distance-anonymity ?swap-dist
    using l-one-is-sym symmetric-norm-imp-distance-anonymous[of l-one]
    by simp
  thus 𝒮𝒞ℱ-result.anonymity
         (𝒮𝒞ℱ-result.distance-ℛ ?swap-dist strong-unanimity)
    using strong-unanimity-anonymous
         𝒮𝒞ℱ-result.anonymous-distance-and-consensus-imp-rule-anonymity
    by metis
qed
```

7.11.4 Neutrality

```
lemma swap-dist-neutral: distance-neutrality well-formed-elections
                           (votewise-distance swap l-one)
  using neutral-dist-imp-neutral-votewise-dist swap-neutral
  by blast

theorem kemeny-rule-neutral: 𝒮𝒞ℱ-properties.neutrality kemeny-rule
 using strong-unanimity-neutral' swap-dist-neutral strong-unanimity-closed-under-neutrality
       𝒮𝒞ℱ-properties.neutr-dist-and-cons-imp-neutr-dr
  unfolding kemeny-rule.simps swap-ℛ.simps
  by blast

end
```

Bibliography

[1] Karsten Diekhoff, Michael Kirsten, and Jonas Krämer. Formal property-oriented design of voting rules using composable modules. In Saša Pekeč and Kristen Brent Venable, editors, *6th International Conference on Algorithmic Decision Theory (ADT 2019)*, volume 11834 of *Lecture Notes in Artificial Intelligence*, pages 164–166. Springer, 2019. doi:10.1007/978-3-030-31489-7.

[2] Karsten Diekhoff, Michael Kirsten, and Jonas Krämer. Verified construction of fair voting rules. In Maurizio Gabbrielli, editor, *29th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2019), Revised Selected Papers*, volume 12042 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2020. doi:10.1007/978-3-030-45260-5_6.

[3] Benjamin Hadjibeyli and Mark C. Wilson. Distance rationalization of social rules. *Computing Research Repository (CoRR)*, abs/1610.01902, 2016. arXiv:1610.01902.