Verified Construction of Fair Voting Rules

Michael Kirsten

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

November 9, 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

Chapter 1

Social-Choice Types

1.1 Auxiliary Lemmas

```
theory Auxiliary-Lemmas
  imports Main
begin

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 sum f X = sum (f ∘ g) Y
  ⟨proof⟩

lemma the-inv-comp:
  fixes
    f :: 'y ⇒ 'z and
    g :: 'x ⇒ 'y and
    s :: 'x set and
    t :: 'y set and
    u :: 'z set and
    x :: 'z
  assumes
    bij-betw f t u and
    bij-betw g s t and
    x ∈ u
  shows the-inv-into s (f ∘ g) x = ((the-inv-into s g) ∘ (the-inv-into t f)) x
⟨proof⟩

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*
      (- $\preceq_{-}$ - [*50*, *1000*, *51*] *50*) **where**
    $a \preceq_r b = ((a, b) \in r)$

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*
  ⟨*proof*⟩

lemma *lin-imp-trans*:
  fixes
    A :: *'a set* **and**
    r :: *'a Preference-Relation*
  assumes *linear-order-on A r*
  shows *trans r*
  ⟨*proof*⟩

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: $a \preceq_r a$ **and**
fin: *finite r*
shows *rank r a* ≥ *1*
⟨*proof*⟩

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
$a \preceq_r b$ **and**
limited A r
shows *a* ∈ *A* ∧ *b* ∈ *A*
⟨*proof*⟩

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*. $a \preceq_r b \lor b \preceq_r a$)

lemma *connex-imp-refl*:
fixes
A :: *'a set* **and**
r :: *'a Preference-Relation*
assumes *connex A r*
shows *refl-on A r*
⟨*proof*⟩

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⟩

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⟩

lemma limit-to-limits:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation
  shows limited A (limit A r)
  ⟨proof⟩

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⟩

lemma limit-presv-antisym:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation
  assumes antisym r
  shows antisym (limit A r)
  ⟨proof⟩

lemma limit-presv-trans:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation
  assumes trans r
  shows trans (limit A r)
  ⟨proof⟩

lemma limit-presv-lin-ord:
  fixes
```

$A\ B$ :: *'a set* **and**
r :: *'a Preference-Relation*
assumes
linear-order-on $B\ r$ **and**
$A \subseteq B$
shows *linear-order-on* A (*limit* $A\ r$)
⟨*proof*⟩

lemma *limit-presv-prefs*:
fixes
A :: *'a set* **and**
r :: *'a Preference-Relation* **and**
$a\ b$ :: *'a*
assumes
$a \preceq_r b$ **and**
$a \in A$ **and**
$b \in A$
shows *let* s = *limit* $A\ r$ *in* $a \preceq_s b$
⟨*proof*⟩

lemma *limit-rel-presv-prefs*:
fixes
A :: *'a set* **and**
r :: *'a Preference-Relation* **and**
$a\ b$ :: *'a*
assumes $(a, b) \in$ *limit* $A\ r$
shows $a \preceq_r b$
⟨*proof*⟩

lemma *limit-trans*:
fixes
$A\ B$ :: *'a set* **and**
r :: *'a Preference-Relation*
assumes $A \subseteq B$
shows *limit* $A\ r$ = *limit* A (*limit* $B\ r$)
⟨*proof*⟩

lemma *lin-ord-not-empty*:
fixes r :: *'a Preference-Relation*
assumes $r \neq \{\}$
shows $\neg$ *linear-order-on* $\{\}\ r$
⟨*proof*⟩

lemma *lin-ord-singleton*:
fixes a :: *'a*
shows $\forall\ r.$ *linear-order-on* $\{a\}\ r \longrightarrow r = \{(a, a)\}$
⟨*proof*⟩

1.2.4 Auxiliary Lemmas

lemma *above-trans*:
fixes
r :: *'a Preference-Relation* **and**
a b :: *'a*
assumes
trans r **and**
$(a,\ b) \in r$
shows *above r b* $\subseteq$ *above r a*
⟨*proof*⟩

lemma *above-refl*:
fixes
A :: *'a set* **and**
r :: *'a Preference-Relation* **and**
a :: *'a*
assumes
refl-on A r **and**
$a \in A$
shows $a \in$ *above r a*
⟨*proof*⟩

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 $\subseteq$ *above r' a* **and**
above r' a $= \{a\}$
shows *above r a* $= \{a\}$
⟨*proof*⟩

lemma *above-connex*:
fixes
A :: *'a set* **and**
r :: *'a Preference-Relation* **and**
a :: *'a*
assumes
connex A r **and**
$a \in A$
shows $a \in$ *above r a*
⟨*proof*⟩

lemma *pref-imp-in-above*:
fixes
r :: *'a Preference-Relation* **and**

a b :: *'a*
shows $(a \preceq_r b) = (b \in \textit{above } r\ a)$
⟨*proof*⟩

lemma *limit-presv-above*:
fixes
A :: *'a set* **and**
r :: *'a Preference-Relation* **and**
a b :: *'a*
assumes
$b \in \textit{above } r\ a$ **and**
$a \in A$ **and**
$b \in A$
shows $b \in \textit{above } (\textit{limit } A\ r)\ a$
⟨*proof*⟩

lemma *limit-rel-presv-above*:
fixes
A B :: *'a set* **and**
r :: *'a Preference-Relation* **and**
a b :: *'a*
assumes $b \in \textit{above } (\textit{limit } B\ r)\ a$
shows $b \in \textit{above } r\ a$
⟨*proof*⟩

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: $A \neq \{\}$
shows $\exists\ a \in A.\ \textit{above } r\ a = \{a\} \wedge (\forall\ a' \in A.\ \textit{above } r\ a' = \{a'\} \longrightarrow a' = a)$
⟨*proof*⟩

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: $A \neq \{\}$ **and**
above-a: $\textit{above } r\ a = \{a\}$ **and**
above-b: $\textit{above } r\ b = \{b\}$
shows $a = b$
⟨*proof*⟩

```
lemma above-one-imp-rank-one:
  fixes
    r :: 'a Preference-Relation and
    a :: 'a
  assumes above r a = {a}
  shows rank r a = 1
  ⟨proof⟩

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⟩

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)
  ⟨proof⟩

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⟩

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

⟨*proof*⟩

1.2.5 Lifting Property

definition *equiv-rel-except-a* :: $'a$ *set* $\Rightarrow$ $'a$ *Preference-Relation* $\Rightarrow$
    $'a$ *Preference-Relation* $\Rightarrow$ $'a \Rightarrow$ *bool* **where**
  equiv-rel-except-a $A\ r\ r'\ a \equiv$
    linear-order-on $A\ r \land$ *linear-order-on* $A\ r' \land a \in A \land$
    $(\forall\ a' \in A - \{a\}.\ \forall\ b' \in A - \{a\}.\ (a' \preceq_r b') = (a' \preceq_{r'} b'))$

definition *lifted* :: $'a$ *set* $\Rightarrow$ $'a$ *Preference-Relation* $\Rightarrow$ $'a$ *Preference-Relation* $\Rightarrow$
    $'a \Rightarrow$ *bool* **where**
  lifted $A\ r\ r'\ a \equiv$
    equiv-rel-except-a $A\ r\ r'\ a \land (\exists\ a' \in A - \{a\}.\ a \preceq_r a' \land a' \preceq_{r'} a)$

lemma *trivial-equiv-rel*:
  fixes
    A :: $'a$ *set* **and**
    r :: $'a$ *Preference-Relation*
  assumes *linear-order-on* $A\ r$
  shows $\forall\ a \in A.$ *equiv-rel-except-a* $A\ r\ r\ a$
  ⟨*proof*⟩

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* $A\ r\ r'\ a$
  ⟨*proof*⟩

lemma *lifted-imp-switched*:
  fixes
    A :: $'a$ *set* **and**
    $r\ r'$ :: $'a$ *Preference-Relation* **and**
    a :: $'a$
  assumes *lifted* $A\ r\ r'\ a$
  shows $\forall\ a' \in A - \{a\}.\ \neg\ (a' \preceq_r a \land a \preceq_{r'} a')$
⟨*proof*⟩

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: $a' \preceq_r a$

shows $a' \preceq_r' a$
⟨*proof*⟩

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* $\subseteq$ *above r a*
⟨*proof*⟩

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: $a' \in A - \{a\}$
shows *above r a'* $\subseteq$ *above r' a'* $\cup \{a\}$
⟨*proof*⟩

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: $A \subseteq A'$
shows *limit A r* = *limit A r'* $\vee$ *lifted A* (*limit A r*) (*limit A r'*) *a*
⟨*proof*⟩

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: $A \subseteq A'$ **and**
not-in-A: $a \notin A$
shows *limit A r* = *limit A r'*
⟨*proof*⟩

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⟩

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}
  ⟨proof⟩

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⟩

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**
  $norm\ n \equiv \forall\ (x{::}ereal\ list).\ n\ x \geq 0 \wedge (\forall\ i < length\ x.\ (x!i = 0) \longrightarrow n\ x = 0)$

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 $(\sum\ a \in A.\ g\ a) = (\sum\ a' \in A'.\ g\ (the\text{-}inv\text{-}into\ A\ f\ a'))$
  ⟨*proof*⟩

1.3.3 Common Norms

fun *l-one* :: *Norm* **where**
  $l\text{-}one\ x = (\sum\ i < length\ x.\ |x!i|)$

1.3.4 Properties

definition *symmetry* :: *Norm* ⇒ *bool* **where**
  $symmetry\ n \equiv \forall\ x\ y.\ x <\sim\sim> y \longrightarrow n\ x = n\ y$

1.3.5 Theorems

theorem *l-one-is-sym*: *symmetry l-one*
⟨*proof*⟩

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) limit_ℛ :: 'a set ⇒ 'r Result ⇒ 'r Result where
  limit_ℛ 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-$\mathcal{E}$* :: (*'a*, *'v*) *Election* ⇒ *'a set* **where**
  alternatives-$\mathcal{E}$ E = *fst E*

fun *voters-$\mathcal{E}$* :: (*'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 sum (λ p. vote-count p E) UNIV = card (voters-ℰ E)
⟨proof⟩
```

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⟩

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
 ⟨proof⟩

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'
 ⟨proof⟩
```

```
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⟩

lemma rename-inj:
  fixes π :: 'v ⇒ 'v
  assumes bij π
  shows inj (rename π)
⟨proof⟩

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⟩
```

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⟩

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)
⟨proof⟩

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)
  ⟨proof⟩

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⟩

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⟩

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⟩
```

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 infinity)

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 infinity)

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
  ⟨proof⟩

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

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})
  ⟨proof⟩

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⟩
```

```
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⟩

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
  ⟨proof⟩

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
  ⟨proof⟩
```

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
  ⟨proof⟩

lemma wins-irreflex:
  fixes
    p :: ('a, 'v) Profile and
    a :: 'a and
    V :: 'v set
  shows ¬ wins V a p a
  ⟨proof⟩
```

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⟩
```

```
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⟩
```

```
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
  ⟨proof⟩
```

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)
  ⟨proof⟩

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⟩
```

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⟩

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 state-
ments about the voter set.
⟨proof⟩

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⟩

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⟩
```

```
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⟩
```

```
lemma defer-subset:
  fixes
    A :: 'a set and
    r :: 'a Result
  assumes well-formed-𝒮𝒞ℱ A r
  shows defer-r r ⊆ A
⟨proof⟩
```

```
lemma elect-subset:
  fixes
    A :: 'a set and
    r :: 'a Result
  assumes well-formed-𝒮𝒞ℱ A r
  shows elect-r r ⊆ A
⟨proof⟩
```

```
lemma reject-subset:
  fixes
    A :: 'a set and
    r :: 'a Result
  assumes well-formed-𝒮𝒞ℱ A r
  shows reject-r r ⊆ A
⟨proof⟩
```

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

fun *limit-$\mathcal{SWF}$* :: *'a set* ⇒ (*'a Preference-Relation*) *set* ⇒
        (*'a Preference-Relation*) *set* **where**
  limit-$\mathcal{SWF}$ 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.

⟨*ML*⟩

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 *$\mathcal{SCF}$-result*: *result well-formed-$\mathcal{SCF}$ limit-$\mathcal{SCF}$*
⟨*proof*⟩

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*
  $\lambda\ A\ r.\ \mathit{set\text{-}equals\text{-}partition}\ (\mathit{Pow}\ A)\ r \wedge \mathit{disjoint3}\ r$
  $\lambda\ A\ rs.\ \{r \cap A \mid r.\ r \in rs\}$
$\langle \mathit{proof} \rangle$

Results from social welfare functions ($\mathcal{SWF}$*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 $\mathcal{SWF}$*-result*: *result well-formed-*$\mathcal{SWF}$ *limit-*$\mathcal{SWF}$
$\langle \mathit{proof} \rangle$

$\langle \mathit{ML} \rangle$

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

1.9.2 Relations for Symmetry Constructions

fun *restricted-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 *closed-restricted-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 *action-induced-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 *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 $\mathit{closed\text{-}rel} :: {'x}\ \mathit{set} \Rightarrow {'x}\ \mathit{rel} \Rightarrow \mathit{bool}$ **where**
$\mathit{closed\text{-}rel}\ s\ r = (\forall\ x\ y.\ (x,\ y) \in r \longrightarrow x \in s \longrightarrow y \in s)$

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

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

definition $\mathit{action\text{-}induced\text{-}equivariance} :: {'z}\ \mathit{set} \Rightarrow {'x}\ \mathit{set} \Rightarrow ({'z},\ {'x})\ \mathit{binary\text{-}fun} \Rightarrow$
$({'z},\ {'y})\ \mathit{binary\text{-}fun} \Rightarrow ({'x},{'y})\ \mathit{symmetry}$ **where**
$\mathit{action\text{-}induced\text{-}equivariance}\ t\ s\ \varphi\ \psi = \mathit{Equivariance}\ s\ \{(\varphi\ z,\ \psi\ z) \mid z.\ z \in t\}$

1.9.4 Auxiliary Lemmas

lemma *un-left-inv-singleton-set-system*: $\bigcup \circ \mathit{singleton\text{-}set\text{-}system} = \mathit{id}$
⟨*proof*⟩

lemma *preimg-comp*:
fixes
$f :: {'x} \Rightarrow {'y}$ **and**
$g :: {'x} \Rightarrow {'x}$ **and**
$s :: {'x}\ \mathit{set}$ **and**
$x :: {'y}$
shows $\mathit{preimg}\ f\ (g\ `\ s)\ x = g\ `\ \mathit{preimg}\ (f \circ g)\ s\ x$
⟨*proof*⟩

1.9.5 Rewrite Rules

theorem *rewrite-invar-as-equivar*:

fixes
$f :: 'x \Rightarrow 'y$ **and**
$s :: 'x\ \mathit{set}$ **and**
$t :: 'z\ \mathit{set}$ **and**
$\varphi :: ('z, 'x)\ \mathit{binary\text{-}fun}$
shows $\mathit{is\text{-}symmetry}\ f\ (\mathit{Invariance}\ (\mathit{action\text{-}induced\text{-}rel}\ t\ s\ \varphi)) =$
$\mathit{is\text{-}symmetry}\ f\ (\mathit{action\text{-}induced\text{-}equivariance}\ t\ s\ \varphi\ (\lambda\ g.\ \mathit{id}))$
$\langle \mathit{proof} \rangle$

lemma *rewrite-invar-ind-by-act*:
fixes
$f :: 'x \Rightarrow 'y$ **and**
$s :: 'z\ \mathit{set}$ **and**
$t :: 'x\ \mathit{set}$ **and**
$\varphi :: ('z, 'x)\ \mathit{binary\text{-}fun}$
shows $\mathit{is\text{-}symmetry}\ f\ (\mathit{Invariance}\ (\mathit{action\text{-}induced\text{-}rel}\ s\ t\ \varphi)) =$
$(\forall\ x \in s.\ \forall\ y \in t.\ f\ y = f\ (\varphi\ x\ y))$
$\langle \mathit{proof} \rangle$

lemma *rewrite-equivariance*:
fixes
$f :: 'x \Rightarrow 'y$ **and**
$s :: 'z\ \mathit{set}$ **and**
$t :: 'x\ \mathit{set}$ **and**
$\varphi :: ('z, 'x)\ \mathit{binary\text{-}fun}$ **and**
$\psi :: ('z, 'y)\ \mathit{binary\text{-}fun}$
shows $\mathit{is\text{-}symmetry}\ f\ (\mathit{action\text{-}induced\text{-}equivariance}\ s\ t\ \varphi\ \psi) =$
$(\forall\ x \in s.\ \forall\ y \in t.\ f\ (\varphi\ x\ y) = \psi\ x\ (f\ y))$
$\langle \mathit{proof} \rangle$

lemma *rewrite-group-action-img*:
fixes
$m :: 'x\ \mathit{monoid}$ **and**
$s\ t :: 'y\ \mathit{set}$ **and**
$\varphi :: ('x, 'y)\ \mathit{binary\text{-}fun}$ **and**
$x\ y :: 'x$
assumes
$t \subseteq s$ **and**
$x \in \mathit{carrier}\ m$ **and**
$y \in \mathit{carrier}\ m$ **and**
$\mathit{group\text{-}action}\ m\ s\ \varphi$
shows $\varphi\ (x \otimes_{m} y)\ `\ t = \varphi\ x\ `\ \varphi\ y\ `\ t$
$\langle \mathit{proof} \rangle$

lemma *rewrite-carrier*: $\mathit{carrier}\ (\mathit{BijGroup}\ \mathit{UNIV}) = \{f'.\ \mathit{bij}\ f'\}$
$\langle \mathit{proof} \rangle$

lemma *universal-set-carrier-imp-bij-group*:
fixes $f :: 'a \Rightarrow 'a$

assumes $f \in$ *carrier* (*BijGroup UNIV*)
shows *bij f*
⟨*proof*⟩

lemma *rewrite-sym-group*:
fixes
f g :: $'a \Rightarrow 'a$ **and**
s :: $'a$ *set*
assumes
$f \in$ *carrier* (*BijGroup s*) **and**
$g \in$ *carrier* (*BijGroup s*)
shows
rewrite-mult: $f \otimes_{\mathit{BijGroup}\ s} g =$ *extensional-continuation* $(f \circ g)$ *s* **and**
rewrite-mult-univ: $s = \mathit{UNIV} \longrightarrow f \otimes_{\mathit{BijGroup}\ s} g = f \circ g$
⟨*proof*⟩

lemma *simp-extensional-univ*:
fixes *f* :: $'a \Rightarrow 'b$
shows *extensional-continuation f UNIV* = *f*
⟨*proof*⟩

lemma *extensional-continuation-subset*:
fixes
f :: $'a \Rightarrow 'b$ **and**
s t :: $'a$ *set* **and**
x :: $'a$
assumes
$t \subseteq s$ **and**
$x \in t$
shows *extensional-continuation f s x* = *extensional-continuation f t x*
⟨*proof*⟩

lemma *rel-ind-by-coinciding-action-on-subset-eq-restr*:
fixes
$\varphi\ \psi$ :: $('a, 'b)$ *binary-fun* **and**
s :: $'a$ *set* **and**
t u :: $'b$ *set*
assumes
$u \subseteq t$ **and**
$\forall\ x \in s.\ \forall\ y \in u.\ \psi\ x\ y = \varphi\ x\ y$
shows *action-induced-rel s u* ψ = *restricted-rel* (*action-induced-rel s t* φ) *u UNIV*
⟨*proof*⟩

lemma *coinciding-actions-ind-equal-rel*:
fixes
s :: $'x$ *set* **and**
t :: $'y$ *set* **and**
$\varphi\ \psi$ :: $('x, 'y)$ *binary-fun*
assumes $\forall\ x \in s.\ \forall\ y \in t.\ \varphi\ x\ y = \psi\ x\ y$

shows *action-induced-rel s t φ* = *action-induced-rel s t ψ*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

theorem *group-action-equivar-f-imp-equivar-preimg*:
fixes
f :: *'x* ⇒ *'y* **and**
$\mathcal{D}_f$ 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*) $\mathcal{D}_f$ φ ψ
assumes
action-φ: *group-action m s* φ **and**
action-res: *group-action m UNIV* ψ **and**
dom-in-s: $\mathcal{D}_f \subseteq s$ **and**
closed-domain:
closed-restricted-rel (*action-induced-rel* (*carrier m*) *s* φ) *s* $\mathcal{D}_f$ **and**
equivar-f: *is-symmetry f equivar-prop* **and**
group-elem-x: $x \in$ *carrier m*
shows ∀ *y*. *preimg f* $\mathcal{D}_f$ (ψ *x y*) = (φ *x*) ' (*preimg f* $\mathcal{D}_f$ *y*)
⟨*proof*⟩

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* ($g \circ f$) (*Invariance r*)
⟨*proof*⟩

lemma *equivar-comp*:
fixes
f :: *'x* ⇒ *'y* **and**
g :: *'y* ⇒ *'z* **and**
s :: *'x set* **and**
t :: *'y set* **and**
τ :: ((*'x* ⇒ *'x*) × (*'y* ⇒ *'y*)) *set* **and**
υ :: ((*'y* ⇒ *'y*) × (*'z* ⇒ *'z*)) *set*
defines
transitive-acts ≡
{(φ, ψ). ∃ χ :: *'y* ⇒ *'y*. (φ, χ) ∈ τ ∧ (χ, ψ) ∈ υ ∧ χ ' *f* ' *s* ⊆ *t*}
assumes
f ' *s* ⊆ *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*⟩

```
lemma equivar-ind-by-action-comp:
  fixes
    f :: 'x ⇒ 'y and
    g :: 'y ⇒ 'z and
    s :: 'w set and
    t :: 'x set and
    u :: 'y set and
    φ :: ('w, 'x) binary-fun and
    χ :: ('w, 'y) binary-fun and
    ψ :: ('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⟩

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⟩

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))
⟨proof⟩

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 (in result) closed-elections :: ('a, 'v) Election rel ⇒ bool where
  closed-elections r =
    (∀ (e, e') ∈ r.
      limit (alternatives-ℰ e) UNIV = limit (alternatives-ℰ e') UNIV)

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

```
definition anonymity_𝒢 :: ('v ⇒ 'v) monoid where
  anonymity_𝒢 = BijGroup (UNIV :: 'v set)

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_ℛ :: ('a, 'v) Election set ⇒ ('a, 'v) Election rel where
  anonymity_ℛ ℰ = action-induced-rel (carrier anonymity_𝒢) ℰ (φ-anon 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 ℰ =
      (π ‘ (alternatives-ℰ ℰ), voters-ℰ ℰ, (rel-rename π) ∘ (profile-ℰ ℰ))

definition neutrality_𝒢 :: ('a ⇒ 'a) monoid where
  neutrality_𝒢 = BijGroup (UNIV :: 'a set)

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

fun neutrality_ℛ :: ('a, 'v) Election set ⇒ ('a, 'v) Election rel where
  neutrality_ℛ ℰ = action-induced-rel (carrier neutrality_𝒢) ℰ (φ-neutral ℰ)

fun ψ-neutral_c :: ('a ⇒ 'a, 'a) binary-fun where
  ψ-neutral_c π r = π r
```

```
fun ψ-neutral_w :: ('a ⇒ 'a, 'a rel) binary-fun where
  ψ-neutral_w π r = rel-rename π r
```

Homogeneity

```
fun homogeneity_𝓡 :: ('a, 'v) Election set ⇒ ('a, 'v) Election rel where
  homogeneity_𝓡 ℰ =
      {(E, E'). E ∈ ℰ
        ∧  alternatives-ℰ E = alternatives-ℰ E'
        ∧ finite (voters-ℰ E) ∧ finite (voters-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_𝓡' :: ('a, 'v::linorder) Election set ⇒ ('a, 'v) Election rel where
  homogeneity_𝓡' ℰ =
      {(E, E'). E ∈ 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⟩

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⟩

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
  ⟨proof⟩

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
  ⟨proof⟩

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⟩

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⟩
```

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⟩

lemma vote-count-anon-rel:
  fixes
```

$\mathcal{E}$ :: $('a, 'v)$ *Election set* **and**
$E\ E'$ :: $('a, 'v)$ *Election*
assumes
fin-voters-E: *finite* (*voters-$\mathcal{E}$ E*) **and**
fin-voters-E': *finite* (*voters-$\mathcal{E}$ E'*) **and**
default-non-v: $\forall\ v.\ v \notin$ *voters-$\mathcal{E}$ E* $\longrightarrow$ *profile-$\mathcal{E}$ E v* $= \{\}$ **and**
default-non-v': $\forall\ v.\ v \notin$ *voters-$\mathcal{E}$ E'* $\longrightarrow$ *profile-$\mathcal{E}$ E' v* $= \{\}$ **and**
eq: *alternatives-$\mathcal{E}$ E* $=$ *alternatives-$\mathcal{E}$ E'* $\wedge\ (E, E') \in \mathcal{E} \times \mathcal{E}$
$\wedge\ (\forall\ p.$ *vote-count p E* $=$ *vote-count p E'*$)$
shows $(E, E') \in$ *anonymity*$_{\mathcal{R}}$ $\mathcal{E}$
⟨*proof*⟩

lemma *rename-comp*:
fixes $\pi\ \pi'$ :: $'v \Rightarrow\ 'v$
assumes
bij π **and**
bij π'
shows *rename* $\pi \circ$ *rename* $\pi' =$ *rename* $(\pi \circ \pi')$
⟨*proof*⟩

interpretation *anonymous-group-action*:
group-action anonymity$_{\mathcal{G}}$ *well-formed-elections* φ*-anon well-formed-elections*
⟨*proof*⟩

lemma (**in** *result*) *anonymity*:
is-symmetry (λ *E. limit* (*alternatives-$\mathcal{E}$ E*) *UNIV*)
(*Invariance* (*anonymity*$_{\mathcal{R}}$ *well-formed-elections*))
⟨*proof*⟩

1.10.4 Neutrality Lemmas

lemma *rel-rename-helper*:
fixes
r :: $'a$ *rel* **and**
π :: $'a \Rightarrow\ 'a$ **and**
$a\ b$ :: $'a$
assumes *bij* π
shows $(\pi\ a, \pi\ b) \in \{(\pi\ x, \pi\ y) \mid x\ y.\ (x, y) \in r\}$
$\longleftrightarrow (a, b) \in \{(x, y) \mid x\ y.\ (x, y) \in r\}$
⟨*proof*⟩

lemma *rel-rename-comp*:
fixes $\pi\ \pi'$ :: $'a \Rightarrow\ 'a$
shows *rel-rename* $(\pi \circ \pi') =$ *rel-rename* $\pi \circ$ *rel-rename* π'
⟨*proof*⟩

lemma *rel-rename-sound*:
fixes
π :: $'a \Rightarrow\ 'a$ **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*⟩

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*⟩

lemma *rel-rename-bij*:
fixes *π* :: *'a* ⇒ *'a*
assumes *bij-π*: *bij π*
shows *bij* (*rel-rename π*)
⟨*proof*⟩

lemma *alternatives-rename-comp*:
fixes *π π'* :: *'a* ⇒ *'a*
shows *alternatives-rename π* ∘ *alternatives-rename π'* =
alternatives-rename (*π* ∘ *π'*)
⟨*proof*⟩

lemma *well-formed-elects-closed*:
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*⟩

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*⟩

interpretation φ*-neutral-action*: *group-action neutrality*$_{\mathcal{G}}$ *well-formed-elections*
φ*-neutral well-formed-elections*
⟨*proof*⟩

interpretation ψ*-neutral*$_{\mathrm{c}}$*-action*: *group-action neutrality*$_{\mathcal{G}}$ *UNIV* ψ*-neutral*$_{\mathrm{c}}$
⟨*proof*⟩

interpretation ψ*-neutral*$_{\mathrm{w}}$*-action*: *group-action neutrality*$_{\mathcal{G}}$ *UNIV* ψ*-neutral*$_{\mathrm{w}}$
⟨*proof*⟩

lemma *neutrality-*$\mathcal{SCF}$: *is-symmetry* (λ $\mathcal{E}$. *limit-*$\mathcal{SCF}$ (*alternatives-*$\mathcal{E}$ $\mathcal{E}$) *UNIV*)
(*action-induced-equivariance* (*carrier neutrality*$_{\mathcal{G}}$) *well-formed-elections*
(φ*-neutral well-formed-elections*) (*set-action* ψ*-neutral*$_{\mathrm{c}}$))
⟨*proof*⟩

lemma *neutrality-*$\mathcal{SWF}$: *is-symmetry* (λ $\mathcal{E}$. *limit-*$\mathcal{SWF}$ (*alternatives-*$\mathcal{E}$ $\mathcal{E}$) *UNIV*)
(*action-induced-equivariance* (*carrier neutrality*$_{\mathcal{G}}$) *well-formed-elections*
(φ*-neutral well-formed-elections*) (*set-action* ψ*-neutral*$_{\mathrm{w}}$))
⟨*proof*⟩

1.10.5 Homogeneity Lemmas

definition *reflp-on'* :: *'a set* $\Rightarrow$ *'a rel* $\Rightarrow$ *bool* **where**
reflp-on' A r $\longleftrightarrow$ *reflp-on A* (λ *x y*. (*x*, *y*) $\in$ *r*)

lemma *refl-homogeneity*$_{\mathcal{R}}$:
fixes $\mathcal{E}$ :: (*'a*, *'v*) *Election set*
assumes $\mathcal{E}$ $\subseteq$ *finite-elections-*$\mathcal{V}$
shows *reflp-on'* $\mathcal{E}$ (*homogeneity*$_{\mathcal{R}}$ $\mathcal{E}$)
⟨*proof*⟩

lemma (**in** *result*) *homogeneity*:
is-symmetry (λ $\mathcal{E}$. *limit* (*alternatives-*$\mathcal{E}$ $\mathcal{E}$) *UNIV*)
(*Invariance* (*homogeneity*$_{\mathcal{R}}$ *UNIV*))
⟨*proof*⟩

lemma *refl-homogeneity*$_{\mathcal{R}}$*'*:
fixes $\mathcal{E}$ :: (*'a*, *'v::linorder*) *Election set*
assumes $\mathcal{E}$ $\subseteq$ *finite-elections-*$\mathcal{V}$
shows *reflp-on'* $\mathcal{E}$ (*homogeneity*$_{\mathcal{R}}$*'* $\mathcal{E}$)
⟨*proof*⟩

lemma (**in** *result*) *homogeneity'*:
is-symmetry (λ $\mathcal{E}$. *limit* (*alternatives-*$\mathcal{E}$ $\mathcal{E}$) *UNIV*)
(*Invariance* (*homogeneity*$_{\mathcal{R}}$*'* *UNIV*))

⟨*proof*⟩

1.10.6 Reversal Symmetry Lemmas

lemma *reverse-reverse-id*: *reverse-rel* ∘ *reverse-rel* = *id*
⟨*proof*⟩

lemma *reverse-rel-limit*:
fixes
A :: *'a set* **and**
r :: *'a rel*
shows *reverse-rel* (*limit A r*) = *limit A* (*reverse-rel r*)
⟨*proof*⟩

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*)
⟨*proof*⟩

interpretation *reversal*$_{\mathcal{G}}$*-group*: *group reversal*$_{\mathcal{G}}$
⟨*proof*⟩

interpretation *φ-reverse-action*: *group-action reversal*$_{\mathcal{G}}$ *well-formed-elections*
φ-reverse well-formed-elections
⟨*proof*⟩

interpretation *ψ-reverse-action*: *group-action reversal*$_{\mathcal{G}}$ *UNIV ψ-reverse*
⟨*proof*⟩

lemma *reversal-symmetry*: *is-symmetry* (λ ℰ. *limit-𝒮𝒲ℱ* (*alternatives-ℰ* ℰ) *UNIV*)
(*action-induced-equivariance* (*carrier reversal*$_{\mathcal{G}}$) *well-formed-elections*
(*φ-reverse well-formed-elections*) (*set-action ψ-reverse*))
⟨*proof*⟩

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-neutral :: ('a ⇒ 'a, 'b) binary-fun and
        voter-type :: 'v itself
  assumes
    action-neutral: group-action neutrality_𝒢 UNIV ψ-neutral and
    neutrality:
      is-symmetry (λ ℰ :: ('a, 'v) Election. limit (alternatives-ℰ ℰ) UNIV)
              (action-induced-equivariance (carrier neutrality_𝒢)
                well-formed-elections
                (φ-neutral well-formed-elections) (set-action ψ-neutral))

sublocale result-properties ⊆ result
  ⟨proof⟩
```

1.11.2 Interpretations

```
global-interpretation 𝒮𝒞ℱ-properties: result-properties well-formed-𝒮𝒞ℱ
        limit-𝒮𝒞ℱ ψ-neutral_c
  ⟨proof⟩

global-interpretation 𝒮𝒲ℱ-properties: result-properties well-formed-𝒮𝒲ℱ
        limit-𝒮𝒲ℱ ψ-neutral_w
  ⟨proof⟩

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* $\Rightarrow$ *bool* **where**
  well-formed-l $l \equiv$ *distinct* l

2.1.2 Auxiliary Lemmas About Lists

lemma *is-arg-min-equal*:
  fixes
    $f\ g :: 'a \Rightarrow 'b::ord$ **and**
    $S :: 'a\ set$ **and**
    $x :: 'a$
  assumes $\forall\ x \in S.\ f\ x = g\ x$
  shows *is-arg-min* $f\ (\lambda\ s.\ s \in S)\ x =$ *is-arg-min* $g\ (\lambda\ s.\ s \in S)\ x$
$\langle proof \rangle$

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⟩

lemma listset-finiteness:
  fixes l :: 'a set list
  assumes ∀ i::nat. i < length l ⟶ finite (l!i)
  shows finite (listset l)
  ⟨proof⟩

lemma all-ls-elems-same-len:
  fixes l :: 'a set list
  shows ∀ l' :: 'a list. l' ∈ listset l ⟶ length l' = length l
⟨proof⟩

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⟩

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⟩
```

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
  ⟨proof⟩

lemma rank-zero-imp-not-present:
  fixes
    p :: 'a Preference-List and
    a :: 'a
  assumes rank-l p a = 0
  shows a ∉ set p
  ⟨proof⟩
```

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*
(- $\lesssim$- - [*50*, *1000*, *51*] *50*) **where**
$a \lesssim_l b$ = (*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 $a \lesssim_l a$
shows *rank-l l a* ≥ *1*
⟨*proof*⟩

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

lemma *rel-trans*:
fixes *l* :: *'a Preference-List*
shows *trans* (*pl-α l*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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
$a \lesssim_l b$ **and**
limited A l
shows *a* ∈ *A* ∧ *b* ∈ *A*
⟨*proof*⟩

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*)
⟨*proof*⟩

2.1.6 Auxiliary Definitions

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

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

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

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

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

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

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

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

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

2.1.7 Auxiliary Lemmas

lemma *list-trans*[*simp*]:
fixes $l :: {'a}\ \textit{Preference-List}$
shows $\textit{trans}\ l$
$\langle \textit{proof} \rangle$

lemma *list-antisym*[*simp*]:
fixes $l :: {'a}\ \textit{Preference-List}$
shows $\textit{antisym-l}\ l$
$\langle \textit{proof} \rangle$

lemma *lin-order-equiv-list-of-alts*:
fixes
$A :: {'a}\ \textit{set}$ **and**
$l :: {'a}\ \textit{Preference-List}$
shows $\textit{linear-order-on-l}\ A\ l = (A = \textit{set}\ l)$
$\langle \textit{proof} \rangle$

lemma *connex-imp-refl*:
fixes

```
    A :: 'a set and
    l :: 'a Preference-List
  assumes connex-l A l
  shows refl-on-l A l
  ⟨proof⟩

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
  ⟨proof⟩

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)
  ⟨proof⟩

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
  ⟨proof⟩

theorem above-equiv:
  fixes
    l :: 'a Preference-List and
    a :: 'a
  shows set (above-l l a) = above (pl-α l) a
⟨proof⟩

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⟩

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)
⟨*proof*⟩

2.1.8 First Occurrence Indices

lemma *pos-in-list-yields-rank*:
fixes
l :: *'a Preference-List* **and**
a :: *'a* **and**
n :: *nat*
assumes
$\forall\ (j{::}nat) \leq n.\ l!j \neq a$ **and**
$l!(n - 1) = a$
shows *rank-l* $l\ a = n$
⟨*proof*⟩

lemma *ranked-alt-not-at-pos-before*:
fixes
l :: *'a Preference-List* **and**
a :: *'a* **and**
n :: *nat*
assumes
$a \in$ *set* l **and**
$n <$ (*rank-l* $l\ a$) $- 1$
shows $l!n \neq a$
⟨*proof*⟩

lemma *pos-in-list-yields-pos*:
fixes
l :: *'a Preference-List* **and**
a :: *'a*
assumes $a \in$ *set* l
shows $l!$(*rank-l* $l\ a - 1) = a$
⟨*proof*⟩

lemma *rel-of-pref-pred-for-set-eq-list-to-rel*:
fixes l :: *'a Preference-List*
shows *relation-of* $(\lambda\ y\ z.\ y \lesssim_l z)$ (*set* l) = *pl-*α l
⟨*proof*⟩

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 (Preference-List.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))
  ⟨proof⟩
```

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 refinement:
  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⟩

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⟩

typedef 'a Ordered-Preference =
  {p :: 'a :: finite Preference-Relation. linear-order-on (UNIV :: 'a set) p}
  morphisms ord2pref pref2ord
⟨proof⟩

instance Ordered-Preference :: (finite) finite
⟨proof⟩

lemma range-ord2pref: range ord2pref = {p. linear-order p}
  ⟨proof⟩

lemma card-ord-pref: card (UNIV :: 'a :: finite Ordered-Preference set) =
                        fact (card (UNIV :: 'a set))
⟨proof⟩

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⟩

quotient-type ('a, 'v) Election_Q =
  'a set × 'v set × ('a, 'v) Profile / election-equality
  ⟨proof⟩

fun fst_Q :: ('a, 'v) Election_Q ⇒ 'a set where
```

fst$_{\mathcal{Q}}$ *E* = *Product-Type.fst* (*rep-Election*$_{\mathcal{Q}}$ *E*)

fun *snd*$_{\mathcal{Q}}$ :: (*'a*, *'v*) *Election*$_{\mathcal{Q}}$ ⇒ *'v set* × (*'a*, *'v*) *Profile* **where**
snd$_{\mathcal{Q}}$ *E* = *Product-Type.snd* (*rep-Election*$_{\mathcal{Q}}$ *E*)

abbreviation *alternatives-*$\mathcal{E}_{\mathcal{Q}}$ :: (*'a*, *'v*) *Election*$_{\mathcal{Q}}$ ⇒ *'a set* **where**
alternatives-$\mathcal{E}_{\mathcal{Q}}$ *E* ≡ *fst*$_{\mathcal{Q}}$ *E*

abbreviation *voters-*$\mathcal{E}_{\mathcal{Q}}$ :: (*'a*, *'v*) *Election*$_{\mathcal{Q}}$ ⇒ *'v set* **where**
voters-$\mathcal{E}_{\mathcal{Q}}$ *E* ≡ *Product-Type.fst* (*snd*$_{\mathcal{Q}}$ *E*)

abbreviation *profile-*$\mathcal{E}_{\mathcal{Q}}$ :: (*'a*, *'v*) *Election*$_{\mathcal{Q}}$ ⇒ (*'a*, *'v*) *Profile* **where**
profile-$\mathcal{E}_{\mathcal{Q}}$ *E* ≡ *Product-Type.snd* (*snd*$_{\mathcal{Q}}$ *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 $\pi_{\mathcal{Q}}$ :: (*'x* ⇒ *'y*) ⇒ (*'x set* ⇒ *'y*) **where**
  $\pi_{\mathcal{Q}}$ *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-*$\pi_{\mathcal{Q}}$ :: (*'x* ⇒ *'x set*) ⇒ (*'x set* ⇒ *'y*) ⇒ (*'x* ⇒ *'y*) **where**
  inv-$\pi_{\mathcal{Q}}$ *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* $\neq$ *1*
shows *singleton-set s* = *undefined*
⟨*proof*⟩

lemma *singleton-set-def-if-card-one*:
fixes *s* :: *'x set*
assumes *card s* = *1*
shows $\exists!$ *x*. *x* = *singleton-set s* $\wedge$ $\{x\}$ = *s*
⟨*proof*⟩

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* $\Rightarrow$ *'y* **and**
r :: *'x rel* **and**
s :: *'x set*
assumes
f respects r **and**
equiv s r
shows $\forall$ *t* $\in$ *s* // *r*. $\forall$ *x* $\in$ *t*. $\pi_{\mathcal{Q}}$ *f t* = *f x*
⟨*proof*⟩

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* $\Rightarrow$ *'x* **and**
r :: *'x rel* **and**
s :: *'x set*
assumes *equiv s r*
defines *induced-fun* $\equiv$ (*inv*-$\pi_{\mathcal{Q}}$ (*relation-class r*) *f*)
shows
induced-fun respects r **and**
$\forall$ *A* $\in$ *s* // *r*. $\pi_{\mathcal{Q}}$ *induced-fun A* = *f A*
⟨*proof*⟩

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 $\subseteq$ *s*
shows *restricted-rel r t s* = *Restr r t*
⟨*proof*⟩

```
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⟩

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⟩

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
    N :: 'b ⇒ nat and
    B :: 'b set
  assumes
    finite A and
    finite B and
    sum N B = card A
  shows ∃ 𝒳. A = ⋃ {𝒳 i | i. i ∈ B} ∧ (∀ i ∈ B. card (𝒳 i) = N i) ∧
            (∀ i j. i ≠ j ⟶ i ∈ B ∧ j ∈ B ⟶ 𝒳 i ∩ 𝒳 j = {})
  ⟨proof⟩
```

3.2.2 Anonymity Quotient: Grid

fun $anonymity_{\mathcal{Q}}$ :: *'a set* ⇒ (*'a*, *'v*) *Election set set* **where**
  $anonymity_{\mathcal{Q}}$ *A* = *quotient* (*elections-*$\mathcal{A}$ *A*) ($anonymity_{\mathcal{R}}$ (*elections-*$\mathcal{A}$ *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 $vote\text{-}count_{\mathcal{Q}}$ :: *'a Preference-Relation* ⇒ (*'a*, *'v*) *Election set* ⇒ *nat* **where**
  $vote\text{-}count_{\mathcal{Q}}$ *p* = $\pi_{\mathcal{Q}}$ (*vote-count p*)

fun *anonymity-class* :: (*'a::finite*, *'v*) *Election set* ⇒
    (*nat*, *'a Ordered-Preference*) *vec* **where**
  anonymity-class X = (χ *p*. $vote\text{-}count_{\mathcal{Q}}$ (*ord2pref p*) *X*)

lemma *anon-rel-equiv*: *equiv* (*elections-*$\mathcal{A}$ *UNIV*) ($anonymity_{\mathcal{R}}$ (*elections-*$\mathcal{A}$ *UNIV*))
⟨*proof*⟩

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 $anonymity_{\mathcal{Q}}$*-isomorphism*:
  assumes *infinite* (*UNIV* :: *'v set*)
  shows *bij-betw* (*anonymity-class* :: (*'a* :: *finite*, *'v*) *Election set*
      ⇒ *nat*^(*'a Ordered-Preference*)) ($anonymity_{\mathcal{Q}}$ (*UNIV* :: *'a set*))
       (*UNIV* :: (*nat*^(*'a Ordered-Preference*)) *set*)
⟨*proof*⟩

3.2.3 Homogeneity Quotient: Simplex

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

fun $anonymity\text{-}homogeneity_{\mathcal{R}}$ :: (*'a*, *'v*) *Election set* ⇒ (*'a*, *'v*) *Election rel* **where**
  $anonymity\text{-}homogeneity_{\mathcal{R}}$ $\mathcal{E}$ =
   {(*E*, *E'*) | *E E'*. *E* ∈ $\mathcal{E}$ ∧ *E'* ∈ $\mathcal{E}$
        ∧ (*finite* (*voters-*$\mathcal{E}$ *E*) = *finite* (*voters-*$\mathcal{E}$ *E'*))
        ∧ (∀ *r*. *vote-fraction r E* = *vote-fraction r E'*)}

fun $anonymity\text{-}homogeneity_{\mathcal{Q}}$ :: *'a set* ⇒ (*'a*, *'v*) *Election set set* **where**
  $anonymity\text{-}homogeneity_{\mathcal{Q}}$ *A* =
   quotient (*elections-*$\mathcal{A}$ *A*) ($anonymity\text{-}homogeneity_{\mathcal{R}}$ (*elections-*$\mathcal{A}$ *A*))

fun *vote-fraction*$_{\mathcal{Q}}$ :: *'a Preference-Relation* ⇒ (*'a*, *'v*) *Election set* ⇒ *rat* **where**
vote-fraction$_{\mathcal{Q}}$ *p* = $\pi_{\mathcal{Q}}$ (*vote-fraction p*)

fun *anonymity-homogeneity-class* :: (*'a*::*finite*, *'v*) *Election set* ⇒
(*rat*, *'a Ordered-Preference*) *vec* **where**
anonymity-homogeneity-class $\mathcal{E}$ = (χ *p*. *vote-fraction*$_{\mathcal{Q}}$ (*ord2pref p*) $\mathcal{E}$)

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*.
sum f X = *1* ∧ (∀ *x* ∈ *X*. *f x* ≥ *0*)
∧ *x* = *sum* (λ *x*. (*f x*) $*_R$ *x*) *X*
shows *x* ∈ *convex hull X*
⟨*proof*⟩

lemma *standard-simplex-rewrite*: *convex hull standard-basis* =
{*v* :: (*real*^*'b*). (∀ *i*. *v*\$*i* ≥ *0*) ∧ *sum* ((\$) *v*) *UNIV* = *1*}
⟨*proof*⟩

lemma *fract-distr-helper*:
fixes *a b c* :: *int*
assumes *c* ≠ *0*
shows *Fract a c* + *Fract b c* = *Fract* (*a* + *b*) *c*
⟨*proof*⟩

lemma *anonymity-homogeneity-is-equivalence*:
fixes *X* :: (*'a*, *'v*) *Election set*
assumes ∀ *E* ∈ *X*. *finite* (*voters-*$\mathcal{E}$ *E*)
shows *equiv X* (*anonymity-homogeneity*$_{\mathcal{R}}$ *X*)

```
⟨proof⟩

lemma fract-distr:
  fixes
    A :: 'x set and
    f :: 'x ⇒ int and
    b :: int
  assumes
    finite A and
    b ≠ 0
  shows sum (λ a. Fract (f a) b) A = Fract (sum f A) b
  ⟨proof⟩
```

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⟩

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* ⇒ *'a* ⇒ *ereal*

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

fun *tup* :: *'a Distance* ⇒ (*'a* ∗ *'a* ⇒ *ereal*) **where**
  tup d = (λ *pair*. *d* (*fst pair*) (*snd pair*))

definition *distance* :: *'a set* ⇒ *'a Distance* ⇒ *bool* **where**
  distance S d ≡ ∀ *x y*. *x* ∈ *S* ∧ *y* ∈ *S* ⟶ *d x x* = *0* ∧ *0* ≤ *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
  ⟨proof⟩

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)
  ⟨proof⟩

lemma distrib:
  fixes
    A :: 'a set and
    f g :: 'a ⇒ int
```

shows $(\sum a \in A.\ f\ a) + (\sum a \in A.\ g\ a) = (\sum a \in A.\ f\ a + g\ a)$
$\langle proof \rangle$

lemma *distrib-ereal*:
fixes
$A :: 'a\ set$ **and**
$f\ g :: 'a \Rightarrow int$
shows *ereal* (*real-of-int* $((\sum a \in A.\ (f :: 'a \Rightarrow int)\ a) + (\sum a \in A.\ g\ a))) =$
ereal (*real-of-int* $((\sum a \in A.\ (f\ a) + (g\ a))))$
$\langle proof \rangle$

lemma *uneq-ereal*:
fixes $x\ y :: int$
assumes $x \leq y$
shows *ereal* (*real-of-int* x) $\leq$ *ereal* (*real-of-int* y)
$\langle proof \rangle$

4.1.5 Swap Distance

fun *neq-ord* :: *'a Preference-Relation* $\Rightarrow$ *'a Preference-Relation* $\Rightarrow$
$'a \Rightarrow 'a \Rightarrow bool$ **where**
neq-ord $r\ s\ a\ b = ((a \preceq_r b \land b \preceq_s a) \lor (b \preceq_r a \land a \preceq_s b))$

fun *pairwise-disagreements* :: $'a\ set \Rightarrow$ *'a Preference-Relation* $\Rightarrow$
'a Preference-Relation $\Rightarrow ('a \times 'a)\ set$ **where**
pairwise-disagreements $A\ r\ s = \{(a, b) \in A \times A.\ a \neq b \land$ *neq-ord* $r\ s\ a\ b\}$

fun *pairwise-disagreements'* :: $'a\ set \Rightarrow$ *'a Preference-Relation* $\Rightarrow$
'a Preference-Relation $\Rightarrow ('a \times 'a)\ set$ **where**
pairwise-disagreements' $A\ r\ s =$
Set.filter $(\lambda\ (a, b).\ a \neq b \land$ *neq-ord* $r\ s\ a\ b)\ (A \times A)$

lemma *set-eq-filter*:
fixes
$X :: 'a\ set$ **and**
$P :: 'a \Rightarrow bool$
shows $\{x \in X.\ P\ x\} =$ *Set.filter* $P\ X$
$\langle proof \rangle$

lemma *pairwise-disagreements-eq*[*code*]: *pairwise-disagreements* = *pairwise-disagreements'*
$\langle proof \rangle$

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 $alts\text{-}\mathcal{V}\ x \neq alts\text{-}\mathcal{V}\ y$
shows $swap\ x\ y = \infty$
$\langle proof \rangle$

lemma *swap-case-fin*:
fixes $x\ y :: {'a}\ Vote$
assumes $alts\text{-}\mathcal{V}\ x = alts\text{-}\mathcal{V}\ y$
shows $swap\ x\ y = card\ (pairwise\text{-}disagreements\ (alts\text{-}\mathcal{V}\ x)\ (pref\text{-}\mathcal{V}\ x)\ (pref\text{-}\mathcal{V}\ y))$
$\langle proof \rangle$

4.1.6 Spearman Distance

fun $spearman :: {'a}\ Vote\ Distance$ **where**
$spearman\ (A,\ x)\ (A',\ y) =$
$(if\ A = A'$
$then\ \sum\ a \in A.\ abs\ (int\ (rank\ x\ a) - int\ (rank\ y\ a))$
$else\ \infty)$

lemma *spearman-case-inf*:
fixes $x\ y :: {'a}\ Vote$
assumes $alts\text{-}\mathcal{V}\ x \neq alts\text{-}\mathcal{V}\ y$
shows $spearman\ x\ y = \infty$
$\langle proof \rangle$

lemma *spearman-case-fin*:
fixes $x\ y :: {'a}\ Vote$
assumes $alts\text{-}\mathcal{V}\ x = alts\text{-}\mathcal{V}\ y$
shows $spearman\ x\ y =$
$(\sum\ a \in alts\text{-}\mathcal{V}\ x.\ abs\ (int\ (rank\ (pref\text{-}\mathcal{V}\ x)\ a) - int\ (rank\ (pref\text{-}\mathcal{V}\ y)\ a)))$
$\langle proof \rangle$

4.1.7 Properties

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

Definitions

fun $total\text{-}invariance_{\mathcal{D}} :: {'x}\ Distance \Rightarrow {'x}\ rel \Rightarrow bool$ **where**
$total\text{-}invariance_{\mathcal{D}}\ d\ rel = is\text{-}symmetry\ (tup\ d)\ (Invariance\ (product\ rel))$

fun $invariance_{\mathcal{D}} :: {'y}\ Distance \Rightarrow {'x}\ set \Rightarrow {'y}\ set \Rightarrow$
$({'x},\ {'y})\ binary\text{-}fun \Rightarrow bool$ **where**
$invariance_{\mathcal{D}}\ d\ X\ Y\ \varphi = is\text{-}symmetry\ (tup\ d)\ (Invariance\ (equivariance\ X\ Y\ \varphi))$

definition $distance\text{-}anonymity :: ({'a},\ {'v})\ Election\ Distance \Rightarrow bool$ **where**
$distance\text{-}anonymity\ d \equiv$
$\forall\ A\ A'\ V\ V'\ p\ p'\ \pi :: ({'v} \Rightarrow {'v}).$

$(bij\ \pi \longrightarrow$
$(d\ (A,\ V,\ p)\ (A',\ V',\ p')) =$
$(d\ (rename\ \pi\ (A,\ V,\ p)))\ (rename\ \pi\ (A',\ V',\ p')))$

fun *distance-anonymity'* :: ('*a*, '*v*) *Election set* ⇒
('*a*, '*v*) *Election Distance* ⇒ *bool* **where**
distance-anonymity' X d = *invariance*$_{\mathcal{D}}$ *d* (*carrier anonymity*$_{\mathcal{G}}$) *X* (φ-*anon X*)

fun *distance-neutrality* :: ('*a*, '*v*) *Election set* ⇒
('*a*, '*v*) *Election Distance* ⇒ *bool* **where**
distance-neutrality X d = *invariance*$_{\mathcal{D}}$ *d* (*carrier neutrality*$_{\mathcal{G}}$) *X* (φ-*neutral X*)

fun *distance-reversal-symmetry* :: ('*a*, '*v*) *Election set* ⇒
('*a*, '*v*) *Election Distance* ⇒ *bool* **where**
distance-reversal-symmetry X d =
invariance$_{\mathcal{D}}$ *d* (*carrier reversal*$_{\mathcal{G}}$) *X* (φ-*reverse X*)

definition *distance-homogeneity'* :: ('*a*, '*v*::*linorder*) *Election set* ⇒
('*a*, '*v*) *Election Distance* ⇒ *bool* **where**
distance-homogeneity' X d = *total-invariance*$_{\mathcal{D}}$ *d* (*homogeneity*$_{\mathcal{R}}$' *X*)

definition *distance-homogeneity* :: ('*a*, '*v*) *Election set* ⇒
('*a*, '*v*) *Election Distance* ⇒ *bool* **where**
distance-homogeneity X d = *total-invariance*$_{\mathcal{D}}$ *d* (*homogeneity*$_{\mathcal{R}}$ *X*)

Auxiliary Lemmas

lemma *rewrite-total-invariance*$_{\mathcal{D}}$:
fixes
d :: '*x Distance* **and**
r :: '*x rel*
shows *total-invariance*$_{\mathcal{D}}$ *d r* = $(\forall\ (x,\ y) \in r.\ \forall\ (a,\ b) \in r.\ d\ a\ x = d\ b\ y)$
⟨*proof*⟩

lemma *rewrite-invariance*$_{\mathcal{D}}$:
fixes
d :: '*y Distance* **and**
X :: '*x set* **and**
Y :: '*y set* **and**
φ :: ('*x*, '*y*) *binary-fun*
shows *invariance*$_{\mathcal{D}}$ *d X Y* φ =
$(\forall\ x \in X.\ \forall\ y \in Y.\ \forall\ z \in Y.\ d\ y\ z = d\ (\varphi\ x\ y)\ (\varphi\ x\ z))$
⟨*proof*⟩

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⟩

lemma swap-neutral: invariance_D swap (carrier neutrality_G)
                        UNIV (λ π (A, q). (π ` A, rel-rename π q))
⟨proof⟩

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⟩

lemma permute-invariant-under-map:
  fixes l l′ :: ′a list
  assumes l <~~> l′
  shows map f l <~~> map f l′
  ⟨proof⟩

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⟩

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
  ⟨proof⟩

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⟩

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 *invar*: *invariance*$_{\mathcal{D}}$ *d* (*carrier neutrality*$_{\mathcal{G}}$) *UNIV vote-action*
  shows *distance-neutrality well-formed-elections* (*votewise-distance d n*)
⟨*proof*⟩

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*$_{\mathcal{C}}$ :: (*'a*, *'v*) *Consensus* **where**
  nonempty-set$_{\mathcal{C}}$ $(A,\ V,\ p) = (A \neq \{\})$

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-profile*$_{\mathcal{C}}$ :: (*'a*, *'v*) *Consensus* **where**
  nonempty-profile$_{\mathcal{C}}$ $(A,\ V,\ p) = (V \neq \{\})$

Equal top ranked alternatives.

fun *equal-top*$_{\mathcal{C}}$*'* :: *'a* ⇒ (*'a*, *'v*) *Consensus* **where**
  equal-top$_{\mathcal{C}}$*'* $a\ (A,\ V,\ p) = (a \in A \land (\forall\ v \in V.\ \textit{above}\ (p\ v)\ a = \{a\}))$

fun *equal-top*$_{\mathcal{C}}$ :: (*'a*, *'v*) *Consensus* **where**
  equal-top$_{\mathcal{C}}$ $c = (\exists\ a.\ \textit{equal-top}_{\mathcal{C}}'\ a\ c)$

Equal votes.

fun *equal-vote*$_{\mathcal{C}}$*'* :: *'a Preference-Relation* ⇒ (*'a*, *'v*) *Consensus* **where**
  equal-vote$_{\mathcal{C}}$*'* $r\ (A,\ V,\ p) = (\forall\ v \in V.\ (p\ v) = r)$

fun *equal-vote*$_{\mathcal{C}}$ :: (*'a*, *'v*) *Consensus* **where**
  equal-vote$_{\mathcal{C}}$ $c = (\exists\ r.\ \textit{equal-vote}_{\mathcal{C}}'\ r\ c)$

Unanimity condition.

```
fun unanimity_𝒞 :: ('a, 'v) Consensus where
  unanimity_𝒞 c = (nonempty-set_𝒞 c ∧ nonempty-profile_𝒞 c ∧ equal-top_𝒞 c)
```

Strong unanimity condition.

```
fun strong-unanimity_𝒞 :: ('a, 'v) Consensus where
  strong-unanimity_𝒞 c = (nonempty-set_𝒞 c ∧ nonempty-profile_𝒞 c ∧ equal-vote_𝒞 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 (neutrality_ℛ 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⟩

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⟩

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*⟩

lemma *ex-anon-cons-imp-cons-anonymous*:
fixes
b :: ('*a*, '*v*) *Consensus* **and**
b':: '*b* ⇒ ('*a*, '*v*) *Consensus*
assumes
general-cond-b: $b = (\lambda\ E.\ \exists\ x.\ b'\ x\ E)$ **and**
all-cond-anon: ∀ *x*. *consensus-anonymity* (*b' x*)
shows *consensus-anonymity b*
⟨*proof*⟩

4.3.5 Theorems

Anonymity

lemma *nonempty-set-cons-anonymous*: *consensus-anonymity nonempty-set*$_{\mathcal{C}}$
⟨*proof*⟩

lemma *nonempty-profile-cons-anonymous*: *consensus-anonymity nonempty-profile*$_{\mathcal{C}}$
⟨*proof*⟩

lemma *equal-top-cons'-anonymous*:
fixes *a* :: '*a*
shows *consensus-anonymity* (*equal-top*$_{\mathcal{C}}$' *a*)
⟨*proof*⟩

lemma *eq-top-cons-anon*: *consensus-anonymity equal-top*$_{\mathcal{C}}$
⟨*proof*⟩

lemma *eq-vote-cons'-anonymous*:
fixes *r* :: '*a Preference-Relation*
shows *consensus-anonymity* (*equal-vote*$_{\mathcal{C}}$' *r*)
⟨*proof*⟩

lemma *eq-vote-cons-anonymous*: *consensus-anonymity equal-vote*$_{\mathcal{C}}$
⟨*proof*⟩

Neutrality

lemma *nonempty-set*$_{\mathcal{C}}$*-neutral*: *consensus-neutrality well-formed-elections nonempty-set*$_{\mathcal{C}}$
⟨*proof*⟩

lemma *nonempty-profile*$_{\mathcal{C}}$*-neutral*: *consensus-neutrality well-formed-elections nonempty-profile*$_{\mathcal{C}}$
⟨*proof*⟩

lemma *equal-vote*$_{\mathcal{C}}$*-neutral*: *consensus-neutrality well-formed-elections equal-vote*$_{\mathcal{C}}$
⟨*proof*⟩

lemma *strong-unanimity*$_{\mathcal{C}}$*-neutral*: *consensus-neutrality*
    well-formed-elections strong-unanimity$_{\mathcal{C}}$
  ⟨*proof*⟩

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* :: $('a,\ 'v,\ 'r\ \mathit{Result})\ \mathit{Electoral\text{-}Module} \Rightarrow\ 'v\ \mathit{set} \Rightarrow\ 'a\ \mathit{set} \Rightarrow$
  $('a,\ 'v)\ \mathit{Profile} \Rightarrow\ 'r\ \mathit{set}$ **where**
 $\mathit{elect}\ m\ V\ A\ p \equiv \mathit{elect\text{-}r}\ (m\ V\ A\ p)$

abbreviation *reject* :: $('a,\ 'v,\ 'r\ \mathit{Result})\ \mathit{Electoral\text{-}Module} \Rightarrow\ 'v\ \mathit{set} \Rightarrow\ 'a\ \mathit{set} \Rightarrow$
  $('a,\ 'v)\ \mathit{Profile} \Rightarrow\ 'r\ \mathit{set}$ **where**
 $\mathit{reject}\ m\ V\ A\ p \equiv \mathit{reject\text{-}r}\ (m\ V\ A\ p)$

abbreviation *defer* :: $('a,\ 'v,\ 'r\ \mathit{Result})\ \mathit{Electoral\text{-}Module} \Rightarrow\ 'v\ \mathit{set} \Rightarrow\ 'a\ \mathit{set} \Rightarrow$
  $('a,\ 'v)\ \mathit{Profile} \Rightarrow\ 'r\ \mathit{set}$ **where**
 $\mathit{defer}\ m\ V\ A\ p \equiv \mathit{defer\text{-}r}\ (m\ V\ A\ p)$

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* :: $('a,\ 'v,\ ('r\ \mathit{Result}))\ \mathit{Electoral\text{-}Module} \Rightarrow$
  bool **where**
  $\mathit{electoral\text{-}module}\ m = (\forall\ A\ V\ p.\ \mathit{profile}\ V\ A\ p \longrightarrow \mathit{well\text{-}formed}\ A\ (m\ V\ A\ p))$

fun *voters-determine-election* :: $('a,\ 'v,\ ('r\ \mathit{Result}))\ \mathit{Electoral\text{-}Module} \Rightarrow \mathit{bool}$ **where**
 $\mathit{voters\text{-}determine\text{-}election}\ m =$
  $(\forall\ A\ V\ p\ p'.\ (\forall\ v \in V.\ p\ v = p'\ v) \longrightarrow m\ V\ A\ p = m\ V\ A\ p')$

lemma (**in** *result*) *electoral-modI*:
 fixes $m :: ('a,\ 'v,\ ('r\ \mathit{Result}))\ \mathit{Electoral\text{-}Module}$
 assumes $\forall\ A\ V\ p.\ \mathit{profile}\ V\ A\ p \longrightarrow \mathit{well\text{-}formed}\ A\ (m\ V\ A\ p)$
 shows *electoral-module m*
 ⟨*proof*⟩

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' :: ('a, 'v) Election set ⇒ ('a, 'v, 'r) Electoral-Module ⇒ bool where
  anonymity' X m = is-symmetry (fun_ℰ m) (Invariance (anonymity_ℛ X))
```

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

```
lemma (in result) hom-imp-anon:
  fixes
    X :: ('a, 'v) Election set and
    m :: ('a, 'v, ('r Result)) Electoral-Module
  assumes
    homogeneity X m and
    ∀ E ∈ X. finite (voters-ℰ E)
  shows anonymity' X m
⟨proof⟩
```

Neutrality

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

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

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* :: (*'a*, *'v*) *Election set* ⇒
    (*'a*, *'v*, *'a rel Result*) *Electoral-Module* ⇒ *bool* **where**
  reversal-symmetry X m =
    is-symmetry (*fun*$_{\mathcal{E}}$ *m*) (*action-induced-equivariance* (*carrier reversal*$_{\mathcal{G}}$) *X*
      (φ*-reverse X*) (*result-action* ψ*-reverse*))

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 ≡
    $\mathcal{SCF}$*-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 ≡
    $\mathcal{SCF}$*-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 ≡
    $\mathcal{SCF}$*-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 ≡
    $\mathcal{SCF}$*-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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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⟩

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⟩

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
  ⟨proof⟩

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
  ⟨proof⟩

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*
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*⟩

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⟩

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⟩

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
  ⟨proof⟩

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⟩
```

```
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
  ⟨proof⟩

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
  ⟨proof⟩

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⟩

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⟩
```

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⟩

theorem electing-imp-non-blocking:
  fixes m :: ('a, 'v, 'a Result) Electoral-Module
  assumes electing m
  shows non-blocking m
⟨proof⟩
```

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⟩

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⟩
```

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⟩
```

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⟩
```

```
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*
$\langle proof \rangle$

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*
$\langle proof \rangle$

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*
$\langle proof \rangle$

4.4.12 Social-Choice Properties

Condorcet Consistency

definition *condorcet-consistency* :: ('*a*, '*v*, '*a Result*) *Electoral-Module* $\Rightarrow$
bool **where**
condorcet-consistency m $\equiv$
$\mathcal{SCF}$*-result.electoral-module m* $\wedge$
($\forall$ *A V p a. condorcet-winner V A p a* $\longrightarrow$
(*m V A p* = ({*e* $\in$ *A. condorcet-winner V A p e*}, *A* $-$ (*elect m V A p*), {})))

lemma *condorcet-consistency'*:
fixes *m* :: ('*a*, '*v*, '*a Result*) *Electoral-Module*
shows *condorcet-consistency m* =
($\mathcal{SCF}$*-result.electoral-module m* $\wedge$
($\forall$ *A V p a. condorcet-winner V A p a* $\longrightarrow$
(*m V A p* = ({*a*}, *A* $-$ (*elect m V A p*), {}))))
$\langle proof \rangle$

lemma *condorcet-consistency''*:
fixes *m* :: ('*a*, '*v*, '*a Result*) *Electoral-Module*
shows *condorcet-consistency m* =
($\mathcal{SCF}$*-result.electoral-module m* $\wedge$
($\forall$ *A V p a.*
condorcet-winner V A p a $\longrightarrow$ *m V A p* = ({*a*}, *A* $-$ {*a*}, {})))
$\langle proof \rangle$

(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* $\Rightarrow$ *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)
  ⟨proof⟩

lemma invariance-is-congruence':
  fixes
    f :: 'x ⇒ 'y and
    r :: 'x rel
  shows (is-symmetry f (Invariance r)) = (f respects r)
  ⟨proof⟩

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
  ⟨proof⟩

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* $=$
$'v\ set \Rightarrow\ 'a \Rightarrow\ 'a\ set \Rightarrow ('a,\ 'v)\ Profile \Rightarrow enat$

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* $\Rightarrow$ *bool* **where**
condorcet-rating $f \equiv$
$\forall\ A\ V\ p\ w\ .\ condorcet\text{-}winner\ V\ A\ p\ w \longrightarrow$
$(\forall\ l \in A\ .\ l \neq w \longrightarrow f\ V\ l\ A\ p < f\ V\ w\ A\ p)$

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* $\Rightarrow$ *bool* **where**
voters-determine-evaluation $f =$
$(\forall\ A\ V\ p\ p'.\ (\forall\ v \in V.\ p\ v = p'\ v) \longrightarrow (\forall\ a \in A.\ f\ V\ a\ A\ p = f\ V\ a\ A\ p'))$

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 $e\ V\ a\ A\ p = Max\ \{e\ V\ b\ A\ p \mid b.\ b \in A\}$
⟨*proof*⟩

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⟩

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 = {a ∈ A . r (e V a A p) t}

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

lemma *less-elim-sound*[*simp*]:
fixes
e :: (*'a*, *'v*) *Evaluation-Function* **and**
t :: *Threshold-Value*
shows $\mathcal{SCF}$*-result.electoral-module* (*less-eliminator e t*)
⟨*proof*⟩

lemma *leq-elim-sound*[*simp*]:
fixes
e :: (*'a*, *'v*) *Evaluation-Function* **and**
t :: *Threshold-Value*
shows $\mathcal{SCF}$*-result.electoral-module* (*leq-eliminator e t*)
⟨*proof*⟩

lemma *max-elim-sound*[*simp*]:
fixes *e* :: (*'a*, *'v*) *Evaluation-Function*
shows $\mathcal{SCF}$*-result.electoral-module* (*max-eliminator e*)
⟨*proof*⟩

lemma *min-elim-sound*[*simp*]:
fixes *e* :: (*'a*, *'v*) *Evaluation-Function*
shows $\mathcal{SCF}$*-result.electoral-module* (*min-eliminator e*)
⟨*proof*⟩

lemma *less-avg-elim-sound*[*simp*]:
fixes *e* :: (*'a*, *'v*) *Evaluation-Function*
shows $\mathcal{SCF}$*-result.electoral-module* (*less-average-eliminator e*)
⟨*proof*⟩

lemma *leq-avg-elim-sound*[*simp*]:
fixes *e* :: (*'a*, *'v*) *Evaluation-Function*
shows $\mathcal{SCF}$*-result.electoral-module* (*leq-average-eliminator e*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *less-elim-non-blocking*:
fixes
e :: (*'a*, *'v*) *Evaluation-Function* **and**

```
    t :: Threshold-Value
  shows non-blocking (less-eliminator e t)
  ⟨proof⟩

lemma leq-elim-non-blocking:
  fixes
    e :: ('a, 'v) Evaluation-Function and
    t :: Threshold-Value
  shows non-blocking (leq-eliminator e t)
  ⟨proof⟩

lemma max-elim-non-blocking:
  fixes e :: ('a, 'v) Evaluation-Function
  shows non-blocking (max-eliminator e)
  ⟨proof⟩

lemma min-elim-non-blocking:
  fixes e :: ('a, 'v) Evaluation-Function
  shows non-blocking (min-eliminator e)
  ⟨proof⟩

lemma less-avg-elim-non-blocking:
  fixes e :: ('a, 'v) Evaluation-Function
  shows non-blocking (less-average-eliminator e)
  ⟨proof⟩

lemma leq-avg-elim-non-blocking:
  fixes e :: ('a, 'v) Evaluation-Function
  shows non-blocking (leq-average-eliminator e)
  ⟨proof⟩
```

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)
  ⟨proof⟩

lemma less-elim-non-electing:
  fixes
    e :: ('a, 'v) Evaluation-Function and
    t :: Threshold-Value
  shows non-electing (less-eliminator e t)
  ⟨proof⟩

lemma leq-elim-non-electing:
```

fixes
  e :: (*'a*, *'v*) *Evaluation-Function* **and**
  t :: *Threshold-Value*
shows *non-electing* (*leq-eliminator e t*)
⟨*proof*⟩

lemma *max-elim-non-electing*:
fixes *e* :: (*'a*, *'v*) *Evaluation-Function*
shows *non-electing* (*max-eliminator e*)
⟨*proof*⟩

lemma *min-elim-non-electing*:
fixes *e* :: (*'a*, *'v*) *Evaluation-Function*
shows *non-electing* (*min-eliminator e*)
⟨*proof*⟩

lemma *less-avg-elim-non-electing*:
fixes *e* :: (*'a*, *'v*) *Evaluation-Function*
shows *non-electing* (*less-average-eliminator e*)
⟨*proof*⟩

lemma *leq-avg-elim-non-electing*:
fixes *e* :: (*'a*, *'v*) *Evaluation-Function*
shows *non-electing* (*leq-average-eliminator e*)
⟨*proof*⟩

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*⟩

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*⟩

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') =
    $(e \cup e',$
     $A - (e \cup e' \cup d \cup d'),$
     $(d \cup d') - (e \cup e'))$

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-$\mathcal{SCF}$ A* (e, r, d) **and**
    wf-second-mod: *well-formed-$\mathcal{SCF}$ A* (e', r', d')
  shows *reject-r* (*max-aggregator A* (e, r, d) (e', r', d')) $= r \cap r'$
⟨*proof*⟩

4.9.3 Soundness

theorem *max-agg-sound*[*simp*]: *aggregator max-aggregator*
⟨*proof*⟩

4.9.4 Properties

The max-aggregator is conservative.

theorem *max-agg-consv*[*simp*]: *agg-conservative max-aggregator*
⟨*proof*⟩

The max-aggregator is commutative.

theorem *max-agg-comm*[*simp*]: *agg-commutative max-aggregator*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

5.1.3 Properties

theorem *def-mod-non-electing*: *non-electing defer-module*
  ⟨*proof*⟩

theorem *def-mod-def-lift-inv*: *defer-lift-invariance defer-module*
  ⟨*proof*⟩

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⟩

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⟩
```

```
lemma strong-unanimity'consensus-imp-elect-fst-mod-completely-determined:
  fixes r :: 'a Preference-Relation
```

shows *well-formed*
(λ *c. nonempty-set*$_{\mathcal{C}}$ *c* $\wedge$ *nonempty-profile*$_{\mathcal{C}}$ *c* $\wedge$ *equal-vote*$_{\mathcal{C}}{}'$ *r c*) *elect-first-module*
⟨*proof*⟩

lemma *strong-unanimity'consensus-imp-elect-fst-mod-well-formed*:
fixes *r* :: *'a Preference-Relation*
shows *well-formed*
(λ *c. nonempty-set*$_{\mathcal{C}}$ *c* $\wedge$ *nonempty-profile*$_{\mathcal{C}}$ *c*
$\wedge$ *equal-vote*$_{\mathcal{C}}{}'$ *r c*) *elect-first-module*
⟨*proof*⟩

lemma *cons-domain-valid*:
fixes *C* :: (*'a*, *'v*, *'r Result*) *Consensus-Class*
shows *elections-*$\mathcal{K}$ *C* $\subseteq$ *well-formed-elections*
⟨*proof*⟩

lemma *cons-domain-finite*:
fixes *C* :: (*'a*, *'v*, *'r Result*) *Consensus-Class*
shows
finite: *elections-*$\mathcal{K}$ *C* $\subseteq$ *finite-elections* **and**
finite-voters: *elections-*$\mathcal{K}$ *C* $\subseteq$ *finite-elections-*$\mathcal{V}$
⟨*proof*⟩

5.3.4 Consensus Rules

definition *non-empty-set* :: (*'a*, *'v*, *'r*) *Consensus-Class* $\Rightarrow$ *bool* **where**
non-empty-set c $\equiv$ $\exists$ *K*. *consensus-*$\mathcal{K}$ *c K*

Unanimity condition.

definition *unanimity* :: (*'a*, *'v::wellorder*, *'a Result*) *Consensus-Class* **where**
unanimity = *consensus-choice unanimity*$_{\mathcal{C}}$ *elect-first-module*

Strong unanimity condition.

definition *strong-unanimity* :: (*'a*, *'v::wellorder*, *'a Result*) *Consensus-Class* **where**
strong-unanimity = *consensus-choice strong-unanimity*$_{\mathcal{C}}$ *elect-first-module*

5.3.5 Properties

definition *consensus-rule-anonymity* :: (*'a*, *'v*, *'r*) *Consensus-Class* $\Rightarrow$ *bool* **where**
consensus-rule-anonymity c $\equiv$
($\forall$ *A V p* π :: (*'v* $\Rightarrow$ *'v*).
bij π $\longrightarrow$
(*let* (*A'*, *V'*, *q*) = (*rename* π (*A*, *V*, *p*)) *in*
profile V A p $\longrightarrow$ *profile V' A' q*
$\longrightarrow$ *consensus-*$\mathcal{K}$ *c* (*A*, *V*, *p*)
$\longrightarrow$ (*consensus-*$\mathcal{K}$ *c* (*A'*, *V'*, *q*) $\wedge$ (*rule-*$\mathcal{K}$ *c V A p* = *rule-*$\mathcal{K}$ *c V' A' q*))))

fun *consensus-rule-anonymity'* :: (*'a*, *'v*) *Election set* $\Rightarrow$
(*'a*, *'v*, *'r Result*) *Consensus-Class* $\Rightarrow$ *bool* **where**

consensus-rule-anonymity′ X C =
is-symmetry (*elect-r* ∘ $fun_{\mathcal{E}}$ (*rule-*$\mathcal{K}$ *C*)) (*Invariance* ($anonymity_{\mathcal{R}}$ *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* ∘ $fun_{\mathcal{E}}$ (*rule-*$\mathcal{K}$ *C*))
(*action-induced-equivariance*
(*carrier* $neutrality_{\mathcal{G}}$) *X* (*φ-neutral X*) (*set-action ψ-neutral*))

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* ∘ $fun_{\mathcal{E}}$ (*rule-*$\mathcal{K}$ *C*))
(*action-induced-equivariance* (*carrier* $reversal_{\mathcal{G}}$) *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* ∘ $fun_{\mathcal{E}}$ *m*) *equivar* **and**
equivar-n: *is-symmetry* (*f* ∘ $fun_{\mathcal{E}}$ *n*) *equivar* **and**
invar-cons: *is-symmetry c* (*Invariance* (*action-induced-rel G X φ*))
shows *is-symmetry* (*f* ∘ $fun_{\mathcal{E}}$ (*rule-*$\mathcal{K}$ *if-else-cons*))
(*action-induced-equivariance G X φ* (*set-action ψ*))
⟨*proof*⟩

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*⟩

5.3.7 Theorems

Anonymity

lemma *unanimity-anonymous*: *consensus-rule-anonymity unanimity*
⟨*proof*⟩

lemma *strong-unanimity-anonymous*: *consensus-rule-anonymity strong-unanimity*
⟨*proof*⟩

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* ∘ $\mathit{fun}_{\mathcal{E}}$ *defer-module*)
(*action-induced-equivariance* G E φ (*set-action* ψ))
⟨*proof*⟩

lemma *elect-first-winners-neutral*: *is-symmetry* (*elect-r* ∘ $\mathit{fun}_{\mathcal{E}}$ *elect-first-module*)
(*action-induced-equivariance* (*carrier* $\mathit{neutrality}_{\mathcal{G}}$)
well-formed-elections (φ*-neutral well-formed-elections*)
(*set-action* $\psi\text{-}\mathit{neutral}_{\mathrm{c}}$))
⟨*proof*⟩

lemma *strong-unanimity-neutral*:
defines *domain* ≡ *well-formed-elections* ∩ *Collect* $\mathit{strong\text{-}unanimity}_{\mathcal{C}}$
— We want to show neutrality on a set as general as possible, as this implies subset neutrality.
shows $\mathcal{SCF}$*-properties.consensus-rule-neutrality domain strong-unanimity*
⟨*proof*⟩

lemma *strong-unanimity-neutral'*: $\mathcal{SCF}$*-properties.consensus-rule-neutrality*
(*elections-*$\mathcal{K}$ *strong-unanimity*) *strong-unanimity*
⟨*proof*⟩

lemma *strong-unanimity-closed-under-neutrality*: *closed-restricted-rel*
($\mathit{neutrality}_{\mathcal{R}}$ *well-formed-elections*) *well-formed-elections*
(*elections-*$\mathcal{K}$ *strong-unanimity*)
⟨*proof*⟩

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 score :: ('a, 'v) Election Distance ⇒ ('a, 'v, 'r Result) Consensus-Class ⇒
        ('a, 'v) Election ⇒ 'r ⇒ ereal where
  score d K E w = Inf (d E ` (𝒦_ℰ K w))

fun (in result) ℛ_𝒲 :: ('a, 'v) Election Distance ⇒
        ('a, 'v, 'r Result) Consensus-Class ⇒ 'v set ⇒ 'a set ⇒ ('a, 'v) Profile ⇒
        'r set where
  ℛ_𝒲 d K V A p = arg-min-set (score d K (A, V, p)) (limit A UNIV)

fun (in result) distance-ℛ :: ('a, 'v) Election Distance ⇒
        ('a, 'v, 'r Result) Consensus-Class ⇒
        ('a, 'v, 'r Result) Electoral-Module where
  distance-ℛ d K V A p =
    (ℛ_𝒲 d K V A p, (limit A UNIV) − ℛ_𝒲 d K V A p, {})
```

5.4.2 Standard Definitions

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

definition voters-determine-distance :: ('a, 'v) Election Distance ⇒ bool where
  voters-determine-distance d ≡
   ∀ A A' V V' p q p'.
     (∀ v ∈ V. p v = q v)
       ⟶ (d (A, V, p) (A', V', p') = d (A, V, q) (A', V', p')
         ∧ (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 profiles :: 'v set ⇒ 'a set ⇒ (('a, 'v) Profile) set where
  profiles V A =
    (if (infinite A ∨ infinite V)
      then {} else {p. p ` V ⊆ (pl-α ` permutations-of-set A)})
```

```
fun 𝒦_ℰ-std :: ('a, 'v, 'r Result) Consensus-Class ⇒ 'r ⇒ 'a set ⇒ 'v set ⇒
        ('a, 'v) Election set where
  𝒦_ℰ-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 𝒦_ℰ-std K w (alternatives-ℰ E) (voters-ℰ E) = {}
          then ∞ else Min (d E ` (𝒦_ℰ-std K w (alternatives-ℰ E) (voters-ℰ E))))
```

```
fun (in result) ℛ_𝒲-std :: ('a, 'v) Election Distance ⇒
        ('a, 'v, 'r Result) Consensus-Class ⇒ 'v set ⇒ 'a set ⇒ ('a, 'v) Profile ⇒
        'r set where
  ℛ_𝒲-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 =
    (ℛ_𝒲-std d K V A p, (limit A UNIV) − ℛ_𝒲-std d K V A p, {})
```

5.4.3 Auxiliary Lemmas

```
lemma fin-𝒦_ℰ:
  fixes C :: ('a, 'v, 'r Result) Consensus-Class
  shows elections-𝒦 C ⊆ finite-elections
⟨proof⟩
```

```
lemma univ-𝒦_ℰ:
  fixes C :: ('a, 'v, 'r Result) Consensus-Class
  shows elections-𝒦 C ⊆ UNIV
  ⟨proof⟩
```

```
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*⟩

lemma *listset-finiteness*:
fixes *l* :: *'a set list*
assumes ∀ *i*::*nat*. *i* < *length l* ⟶ *finite* (*l*!*i*)
shows *finite* (*listset l*)
⟨*proof*⟩

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* = {}
⟨*proof*⟩

lemma *all-ls-elems-same-len*:
fixes *l* :: *'a set list*
shows ∀ *l'* :: *'a list*. *l'* ∈ *listset l* ⟶ *length l'* = *length l*
⟨*proof*⟩

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*⟩

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*⟩

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⟩
```

5.4.5 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⟩

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⟩

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*⟩

lemma *swap-standard*: *standard* (*votewise-distance swap l-one*)
⟨*proof*⟩

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*)
  ⟨*proof*⟩

lemma *swap-$\mathcal{R}$-code*[*code*]: *swap-$\mathcal{R}$* =
    (*$\mathcal{SCF}$-result.distance-$\mathcal{R}$-std* :: (*'a*, *'v* :: *linorder*) *dist-rat-std-type*)
      (*votewise-distance swap l-one*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*$_{\mathcal{D}}$ *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*⟩

Invariance

lemma *closest-dist-invar-under-refl-rel-and-tot-invar-dist*:
fixes
f :: *'a* ⇒ *'b* **and**
domain$_f$ :: *'a set* **and**
d :: *'a Distance* **and**
rel :: *'a rel*
assumes
r-refl: *reflp-on' domain*$_f$ (*Restrp rel domain*$_f$) **and**
tot-invar-d: *total-invariance*$_{\mathcal{D}}$ *d rel*
shows *is-symmetry* (*closest-preimg-distance f domain*$_f$ *d*) (*Invariance rel*)
⟨*proof*⟩

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
r-refl: *reflp-on' domain*$_f$ (*Restrp rel domain*$_f$) **and**
tot-invar-d: *total-invariance*$_{\mathcal{D}}$ *d rel*
shows *is-symmetry* (*minimizer f domain*$_f$ *d img*) (*Invariance rel*)
⟨*proof*⟩

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**
domain$_f$ ⊆ *A* **and**
closed-domain: *closed-restricted-rel rel A domain*$_f$ **and**
invar-d: *invariance*$_{\mathcal{D}}$ *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*⟩

5.6.2 Minimizer Translation

lemma $\mathcal{K}_\mathcal{E}$*-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* ∘ $fun_\mathcal{E}$ (*rule-*$\mathcal{K}$ *C*)) (*elections-*$\mathcal{K}$ *C*) {*w*} = $\mathcal{K}_\mathcal{E}$ *C w*
⟨*proof*⟩

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* ∘ $fun_\mathcal{E}$ (*rule-*$\mathcal{K}$ *C*)) (*elections-*$\mathcal{K}$ *C*) *d E* {*w*}
⟨*proof*⟩

lemma (**in** *result*) $\mathcal{R}_\mathcal{W}$*-is-minimizer*:
fixes
d :: (*'a*, *'v*) *Election Distance* **and**
C :: (*'a*, *'v*, *'r Result*) *Consensus-Class*
shows $fun_\mathcal{E}$ ($\mathcal{R}_\mathcal{W}$ *d C*) =
(λ *E*. ⋃ (*minimizer* (*elect-r* ∘ $fun_\mathcal{E}$ (*rule-*$\mathcal{K}$ *C*)) (*elections-*$\mathcal{K}$ *C*) *d*
(*singleton-set-system* (*limit* (*alternatives-*$\mathcal{E}$ *E*) *UNIV*)) *E*))
⟨*proof*⟩

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-*$\mathcal{K}$ *C*) (*Restrp rel* (*elections-*$\mathcal{K}$ *C*)) **and**
tot-invar-d: $total\text{-}invariance_\mathcal{D}$ *d rel* **and**
invar-res:
is-symmetry (λ *E*. *limit* (*alternatives-*$\mathcal{E}$ *E*) *UNIV*) (*Invariance rel*)
shows *is-symmetry* ($fun_\mathcal{E}$ (*distance-*$\mathcal{R}$ *d C*)) (*Invariance rel*)
⟨*proof*⟩

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 $\equiv$ *action-induced-rel* (*carrier G*) *B* φ **and**
rel' $\equiv$ *action-induced-rel* (*carrier G*) (*elections-*$\mathcal{K}$ *C*) φ
assumes
action-φ: *group-action G B* φ **and**
consensus-C-in-B: *elections-*$\mathcal{K}$ *C* $\subseteq$ *B* **and**
closed-domain:
closed-restricted-rel rel B (*elections-*$\mathcal{K}$ *C*) **and**
invar-res:
is-symmetry (λ *E. limit* (*alternatives-*$\mathcal{E}$ *E*) *UNIV*) (*Invariance rel*) **and**
invar-d: *invariance*$_\mathcal{D}$ *d* (*carrier G*) *B* φ **and**
invar-C-winners: *is-symmetry* (*elect-r* $\circ$ *fun*$_\mathcal{E}$ (*rule-*$\mathcal{K}$ *C*)) (*Invariance rel'*)
shows *is-symmetry* (*fun*$_\mathcal{E}$ (*distance-*$\mathcal{R}$ *d C*)) (*Invariance rel*)
⟨*proof*⟩

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 $\equiv$ *action-induced-rel* (*carrier G*) *B* φ **and**
rel' $\equiv$ *action-induced-rel* (*carrier G*) (*elections-*$\mathcal{K}$ *C*) φ **and**
equivar-prop $\equiv$
action-induced-equivariance (*carrier G*) (*elections-*$\mathcal{K}$ *C*)
φ (*set-action* ψ) **and**
equivar-prop-global-set-valued $\equiv$
action-induced-equivariance (*carrier G*) *B* φ (*set-action* ψ) **and**
equivar-prop-global-result-valued $\equiv$
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-*$\mathcal{K}$ *C* $\subseteq$ *B* **and**
closed-domain: *closed-restricted-rel rel B* (*elections-*$\mathcal{K}$ *C*) **and**
equivar-res:
is-symmetry (λ *E. limit* (*alternatives-*$\mathcal{E}$ *E*) *UNIV*)
equivar-prop-global-set-valued **and**

invar-d: *invariance*$_{\mathcal{D}}$ *d* (*carrier G*) *B* φ **and**
equivar-C-winners: *is-symmetry* (*elect-r* ∘ *fun*$_{\mathcal{E}}$ (*rule-*$\mathcal{K}$ *C*)) *equivar-prop*
shows *is-symmetry* (*fun*$_{\mathcal{E}}$ (*distance-*$\mathcal{R}$ *d C*)) *equivar-prop-global-result-valued*
⟨*proof*⟩

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-*$\mathcal{K}$ *C*) *C* **and**
closed-C: *closed-restricted-rel* (*anonymity*$_{\mathcal{R}}$ *well-formed-elections*)
well-formed-elections (*elections-*$\mathcal{K}$ *C*)
shows *anonymity*' *well-formed-elections* (*distance-*$\mathcal{R}$ *d C*)
⟨*proof*⟩

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-*$\mathcal{K}$ *C*) *C* **and**
closed-C: *closed-restricted-rel* (*neutrality*$_{\mathcal{R}}$ *well-formed-elections*)
well-formed-elections (*elections-*$\mathcal{K}$ *C*)
shows *neutrality well-formed-elections* (*distance-*$\mathcal{R}$ *d C*)
⟨*proof*⟩

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-*$\mathcal{K}$ *C*) *C* **and**
closed-C: *closed-restricted-rel* (*reversal*$_{\mathcal{R}}$ *well-formed-elections*)
well-formed-elections (*elections-*$\mathcal{K}$ *C*)
shows *reversal-symmetry well-formed-elections* ($\mathcal{SWF}$*-result.distance-*$\mathcal{R}$ *d C*)
⟨*proof*⟩

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-*$\mathcal{V}$ *d*
shows *homogeneity finite-elections-*$\mathcal{V}$ (*distance-*$\mathcal{R}$ *d C*)

```
⟨proof⟩

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' finite-elections-𝒱 (distance-ℛ d C)
⟨proof⟩
```

5.6.4 Properties

```
fun decisiveness :: ('a, 'v) Election set ⇒ ('a, 'v) Election Distance ⇒
        ('a, 'v, 'r Result) Electoral-Module ⇒ bool where
  decisiveness X d m =
    (∄ E. E ∈ X
    ∧ (∃ δ > 0. ∀ E' ∈ X. d E E' < δ ⟶ card (elect-r (fun_ℰ m E')) > 1))

end
```

5.7 Distance Rationalization on Election Quotients

```
theory Quotient-Distance-Rationalization
  imports Quotient-Module
          Distance-Rationalization-Symmetry
begin
```

5.7.1 Distances

```
fun distance_𝒬 :: 'x Distance ⇒ 'x set Distance where
  distance_𝒬 d A B = (if (A = {} ∧ B = {}) then 0 else
                 (if (A = {} ∨ B = {}) then ∞ else
                   π_𝒬 (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 =
    $\mathit{Inf}\ (\bigcup\ \{\{\mathit{path\text{-}length}\ p\ d \mid p.\ p \in \mathit{admissible\text{-}paths}\ r\ A\ B\}\})$

fun $\mathit{distance\text{-}infimum}_{\mathcal{Q}}$ :: *'x Distance* ⇒ *'x set Distance* **where**
  $\mathit{distance\text{-}infimum}_{\mathcal{Q}}\ d\ A\ B = \mathit{Inf}\ \{d\ a\ b \mid a\ b.\ a \in A \wedge b \in B\}$

fun *simple* :: *'x rel* ⇒ *'x set* ⇒ *'x Distance* ⇒ *bool* **where**
  simple r X d =
    $(\forall\ A \in X\ //\ r.$
      $(\exists\ a \in A.\ \forall\ B \in X\ //\ r.$
        $\mathit{distance\text{-}infimum}_{\mathcal{Q}}\ d\ A\ B = \mathit{Inf}\ \{d\ a\ b \mid b.\ b \in B\}))$

— 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**
  $\mathit{product'}\ r = \{(p_1,\ p_2).\ ((\mathit{fst}\ p_1,\ \mathit{fst}\ p_2) \in r \wedge \mathit{snd}\ p_1 = \mathit{snd}\ p_2)$
      $\vee\ ((\mathit{snd}\ p_1,\ \mathit{snd}\ p_2) \in r \wedge \mathit{fst}\ p_1 = \mathit{fst}\ p_2)\}$

Auxiliary Lemmas

lemma *tot-dist-invariance-is-congruence*:
  fixes
    d :: *'x Distance* **and**
    r :: *'x rel*
  shows $(\mathit{total\text{-}invariance}_{\mathcal{D}}\ d\ r) = (\mathit{tup}\ d\ \mathit{respects}\ (\mathit{product}\ r))$
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

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: $\mathit{total\text{-}invariance}_{\mathcal{D}}\ d\ r$
  shows $\forall\ A\ B.\ A \in X\ //\ r \wedge B \in X\ //\ r$

$\longrightarrow (\forall\ a\ b.\ a \in A \wedge b \in B \longrightarrow \mathit{distance}_{\mathcal{Q}}\ d\ A\ B = d\ a\ b)$
$\langle \mathit{proof} \rangle$

lemma *relation-paths-subset*:
fixes
$n :: \mathit{nat}$ **and**
$p :: {}'x\ \mathit{list}$ **and**
$r :: {}'x\ \mathit{rel}$ **and**
$X :: {}'x\ \mathit{set}$
assumes $r \subseteq X \times X$
shows $\forall\ p.\ p \in \mathit{relation\text{-}paths}\ r \longrightarrow (\forall\ i < \mathit{length}\ p.\ p!i \in X)$
$\langle \mathit{proof} \rangle$

lemma *admissible-path-len*:
fixes
$d :: {}'x\ \mathit{Distance}$ **and**
$r :: {}'x\ \mathit{rel}$ **and**
$X :: {}'x\ \mathit{set}$ **and**
$a\ b :: {}'x$ **and**
$p :: {}'x\ \mathit{list}$
assumes $\mathit{refl\text{-}on}\ X\ r$
shows $\mathit{triangle\text{-}ineq}\ X\ d \wedge p \in \mathit{relation\text{-}paths}\ r \wedge \mathit{total\text{-}invariance}_{\mathcal{D}}\ d\ r$
$\wedge\ a \in X \wedge b \in X \longrightarrow \mathit{path\text{-}length}\ (a\#p@[b])\ d \geq d\ a\ b$
$\langle \mathit{proof} \rangle$

lemma *quotient-dist-coincides-with-dist*$_{\mathcal{Q}}$:
fixes
$d :: {}'x\ \mathit{Distance}$ **and**
$r :: {}'x\ \mathit{rel}$ **and**
$X :: {}'x\ \mathit{set}$
assumes
equiv: $\mathit{equiv}\ X\ r$ **and**
tri: $\mathit{triangle\text{-}ineq}\ X\ d$ **and**
invar: $\mathit{total\text{-}invariance}_{\mathcal{D}}\ d\ r$
shows $\forall\ A \in X\ //\ r.\ \forall\ B \in X\ //\ r.\ \mathit{quotient\text{-}dist}\ r\ d\ A\ B = \mathit{distance}_{\mathcal{Q}}\ d\ A\ B$
$\langle \mathit{proof} \rangle$

lemma *inf-dist-coincides-with-dist*$_{\mathcal{Q}}$:
fixes
$d :: {}'x\ \mathit{Distance}$ **and**
$r :: {}'x\ \mathit{rel}$ **and**
$X :: {}'x\ \mathit{set}$
assumes
equiv-X-r: $\mathit{equiv}\ X\ r$ **and**
tot-inv-d-r: $\mathit{total\text{-}invariance}_{\mathcal{D}}\ d\ r$
shows $\forall\ A \in X\ //\ r.\ \forall\ B \in X\ //\ r.$
$\mathit{distance\text{-}infimum}_{\mathcal{Q}}\ d\ A\ B = \mathit{distance}_{\mathcal{Q}}\ d\ A\ B$
$\langle \mathit{proof} \rangle$

```
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⟩

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⟩

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⟩
```

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⟩

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⟩

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⟩

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'))))
  ⟨proof⟩

lemma rewr-singleton-set-system-union:
  fixes
    Y :: 'x set set and
```

$X :: {'x}\ set$
assumes $Y \subseteq \mathit{singleton\text{-}set\text{-}system}\ X$
shows
singleton-set-union: $x \in \bigcup\ Y \longleftrightarrow \{x\} \in Y$ **and**
obtain-singleton: $A \in \mathit{singleton\text{-}set\text{-}system}\ X \longleftrightarrow (\exists\ x \in X.\ A = \{x\})$
$\langle proof \rangle$

lemma *union-inf*:
fixes $X :: \mathit{ereal\ set\ set}$
shows $\mathit{Inf}\ \{\mathit{Inf}\ A \mid A.\ A \in X\} = \mathit{Inf}\ (\bigcup\ X)$
$\langle proof \rangle$

5.7.3 Distance Rationalization

fun (**in** *result*) $\mathcal{R}_\mathcal{Q} :: ({'a}, {'v})\ \mathit{Election\ rel} \Rightarrow ({'a}, {'v})\ \mathit{Election\ Distance} \Rightarrow$
$({'a}, {'v}, {'r}\ \mathit{Result})\ \mathit{Consensus\text{-}Class} \Rightarrow ({'a}, {'v})\ \mathit{Election\ set} \Rightarrow {'r}\ \mathit{set}$ **where**
$\mathcal{R}_\mathcal{Q}\ r\ d\ C\ A =$
$\bigcup\ (\mathit{minimizer}\ (\pi_\mathcal{Q}\ (\mathit{elect\text{-}r} \circ \mathit{fun}_\mathcal{E}\ (\mathit{rule\text{-}}\mathcal{K}\ C)))\ (\mathit{elections\text{-}}\mathcal{K}_\mathcal{Q}\ r\ C)$
$(\mathit{distance\text{-}infimum}_\mathcal{Q}\ d)\ (\mathit{singleton\text{-}set\text{-}system}\ (\mathit{limit}_\mathcal{Q}\ A\ \mathit{UNIV}))\ A)$

fun (**in** *result*) $\mathit{distance\text{-}}\mathcal{R}_\mathcal{Q} :: ({'a}, {'v})\ \mathit{Election\ rel} \Rightarrow ({'a}, {'v})\ \mathit{Election\ Distance} \Rightarrow$
$({'a}, {'v}, {'r}\ \mathit{Result})\ \mathit{Consensus\text{-}Class} \Rightarrow ({'a}, {'v})\ \mathit{Election\ set} \Rightarrow {'r}\ \mathit{Result}$ **where**
$\mathit{distance\text{-}}\mathcal{R}_\mathcal{Q}\ r\ d\ C\ A =$
$(\mathcal{R}_\mathcal{Q}\ r\ d\ C\ A,$
$\pi_\mathcal{Q}\ (\lambda\ E.\ \mathit{limit}\ (\mathit{alternatives\text{-}}\mathcal{E}\ E)\ \mathit{UNIV})\ A - \mathcal{R}_\mathcal{Q}\ r\ d\ C\ A,$
$\{\})$

Proposition 4.17 by Hadjibeyli and Wilson [3].

theorem (**in** *result*) *invar-dr-simple-dist-imp-quotient-dr-winners*:
fixes
$d :: ({'a}, {'v})\ \mathit{Election\ Distance}$ **and**
$C :: ({'a}, {'v}, {'r}\ \mathit{Result})\ \mathit{Consensus\text{-}Class}$ **and**
$r :: ({'a}, {'v})\ \mathit{Election\ rel}$ **and**
$X\ A :: ({'a}, {'v})\ \mathit{Election\ set}$
assumes
simple: $\mathit{simple}\ r\ X\ d$ **and**
closed-domain: $\mathit{closed\text{-}restricted\text{-}rel}\ r\ X\ (\mathit{elections\text{-}}\mathcal{K}\ C)$ **and**
invar-res:
$\mathit{is\text{-}symmetry}\ (\lambda\ E.\ \mathit{limit}\ (\mathit{alternatives\text{-}}\mathcal{E}\ E)\ \mathit{UNIV})\ (\mathit{Invariance}\ r)$ **and**
invar-C: $\mathit{is\text{-}symmetry}\ (\mathit{elect\text{-}r} \circ \mathit{fun}_\mathcal{E}\ (\mathit{rule\text{-}}\mathcal{K}\ C))$
$(\mathit{Invariance}\ (\mathit{Restr}\ r\ (\mathit{elections\text{-}}\mathcal{K}\ C)))$ **and**
invar-dr: $\mathit{is\text{-}symmetry}\ (\mathit{fun}_\mathcal{E}\ (\mathcal{R}_\mathcal{W}\ d\ C))\ (\mathit{Invariance}\ r)$ **and**
quot-class: $A \in X\ //\ r$ **and**
equiv-rel: $\mathit{equiv}\ X\ r$ **and**
cons-subset: $\mathit{elections\text{-}}\mathcal{K}\ C \subseteq X$
shows $\pi_\mathcal{Q}\ (\mathit{fun}_\mathcal{E}\ (\mathcal{R}_\mathcal{W}\ d\ C))\ A = \mathcal{R}_\mathcal{Q}\ r\ d\ C\ A$
$\langle proof \rangle$

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⟩

end
```

5.8 Code Generation Interpretations for Results and Properties

```
theory Interpretation-Code
  imports Electoral-Module
          Distance-Rationalization
begin
⟨ML⟩
```

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))
  ⟨proof⟩

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.ℛ𝒲 d K V A p =
            arg-min-set (score d K (A, V, p)) (limit-𝒮𝒞ℱ A UNIV)
  ⟨proof⟩

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 𝒮𝒞ℱ-result.distance-ℛ d K V A p =
      (𝒮𝒞ℱ-result.ℛ𝒲 d K V A p,
        (limit-𝒮𝒞ℱ A UNIV) − 𝒮𝒞ℱ-result.ℛ𝒲 d K V A p,
        {})
  ⟨proof⟩

lemma ℛ𝒲-std-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.ℛ𝒲-std d K V A p =
      arg-min-set (score-std d K (A, V, p)) (limit-𝒮𝒞ℱ A UNIV)
  ⟨proof⟩

lemma distance-ℛ-std-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.distance-ℛ-std d K V A p =
      (𝒮𝒞ℱ-result.ℛ𝒲-std d K V A p,
        (limit-𝒮𝒞ℱ A UNIV) − 𝒮𝒞ℱ-result.ℛ𝒲-std d K V A p,
        {})
  ⟨proof⟩

lemma anonymity-𝒮𝒞ℱ-code-lemma: 𝒮𝒞ℱ-result.anonymity =
    (λ m :: ('a, 'v, 'a Result) Electoral-Module.
      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)))
```

⟨*proof*⟩

5.8.2 Interpretation Declarations and Constants

Declarations for replacing interpreted abstract functions from locales by their explicit instantiations.

declare [[*lc-add* $\mathcal{SCF}$*-result.electoral-module electoral-module-*$\mathcal{SCF}$*-code-lemma*]]
declare [[*lc-add* $\mathcal{SCF}$*-result.*$\mathcal{R}_{\mathcal{W}}$ $\mathcal{R}_{\mathcal{W}}$*-*$\mathcal{SCF}$*-code-lemma*]]
declare [[*lc-add* $\mathcal{SCF}$*-result.*$\mathcal{R}_{\mathcal{W}}$*-std* $\mathcal{R}_{\mathcal{W}}$*-std-*$\mathcal{SCF}$*-code-lemma*]]
declare [[*lc-add* $\mathcal{SCF}$*-result.distance-*$\mathcal{R}$ *distance-*$\mathcal{R}$*-*$\mathcal{SCF}$*-code-lemma*]]
declare [[*lc-add* $\mathcal{SCF}$*-result.distance-*$\mathcal{R}$*-std distance-*$\mathcal{R}$*-std-*$\mathcal{SCF}$*-code-lemma*]]
declare [[*lc-add* $\mathcal{SCF}$*-result.anonymity anonymity-*$\mathcal{SCF}$*-code-lemma*]]

Constant aliases to use instead of the interpreted functions.

definition $\mathcal{R}_{\mathcal{W}}$*-*$\mathcal{SCF}$*-code* = $\mathcal{SCF}$*-result.*$\mathcal{R}_{\mathcal{W}}$
definition $\mathcal{R}_{\mathcal{W}}$*-std-*$\mathcal{SCF}$*-code* = $\mathcal{SCF}$*-result.*$\mathcal{R}_{\mathcal{W}}$*-std*
definition *distance-*$\mathcal{R}$*-*$\mathcal{SCF}$*-code* = $\mathcal{SCF}$*-result.distance-*$\mathcal{R}$
definition *distance-*$\mathcal{R}$*-std-*$\mathcal{SCF}$*-code* = $\mathcal{SCF}$*-result.distance-*$\mathcal{R}$*-std*
definition *electoral-module-*$\mathcal{SCF}$*-code* = $\mathcal{SCF}$*-result.electoral-module*
definition *anonymity-*$\mathcal{SCF}$*-code* = $\mathcal{SCF}$*-result.anonymity*

⟨*ML*⟩

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 =

$(\{\},$
$\{a \in A.\ \mathit{rank}\ (\mathit{limit}\ A\ r)\ a \leq n\},$
$\{a \in A.\ \mathit{rank}\ (\mathit{limit}\ A\ r)\ a > n\})$

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*⟩

lemma *voters-determine-drop-mod*:
fixes
r :: *'a Preference-Relation* **and**
n :: *nat*
shows *voters-determine-election* (*drop-module n r*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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⟩

lemma voters-determine-pass-mod:
  fixes
    r :: 'a Preference-Relation and
    n :: nat
  shows voters-determine-election (pass-module n r)
  ⟨proof⟩
```

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⟩
```

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)
  ⟨proof⟩
```

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)
  ⟨proof⟩
```

```
theorem pass-zero-mod-def-zero[simp]:
  fixes r :: 'a Preference-Relation
  assumes linear-order r
  shows defers 0 (pass-module 0 r)
⟨proof⟩
```

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⟩
```

```
theorem pass-two-mod-def-two:
  fixes r :: 'a Preference-Relation
  assumes linear-order r
  shows defers 2 (pass-module 2 r)
⟨proof⟩
```

```
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*
  ⟨*proof*⟩

lemma *elect-mod-only-voters*: *voters-determine-election elect-module*
  ⟨*proof*⟩

5.11.3 Electing

theorem *elect-mod-electing*[*simp*]: *electing elect-module*
  ⟨*proof*⟩

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 =
    ({},
     {$a \in A$. ∃ $x \in A$. *win-count V p x* > *win-count V p a*},
     {$a \in A$. ∀ $x \in A$. *win-count V p x* ≤ *win-count V p a*})

lemma *enat-leq-enat-set-max*:
  fixes
    x :: *enat* **and**
    X :: *enat set*
  assumes
    $x \in X$ **and**
    finite X
  shows $x \leq$ *Max X*
  ⟨*proof*⟩

lemma *plurality-mod-elim-equiv*:
  fixes
    A :: '*a set* **and**
    V :: '*v set* **and**
    p :: ('*a*, '*v*) *Profile*
  assumes
    non-empty-A: $A \neq \{\}$ **and**
    fin-A: *finite A* **and**
    prof: *profile V A p*
  shows *plurality V A p* = *plurality' V A p*
⟨*proof*⟩

5.12.2 Soundness

theorem *plurality-sound*[*simp*]: *𝒮𝒞ℱ-result.electoral-module plurality*
  ⟨*proof*⟩

theorem *plurality'-sound*[*simp*]: *𝒮𝒞ℱ-result.electoral-module plurality'*
⟨*proof*⟩

lemma *voters-determine-plurality-score*: *voters-determine-evaluation plurality-score*
⟨*proof*⟩

lemma *voters-determine-plurality*: *voters-determine-election plurality*
  ⟨*proof*⟩

5.12.3 Non-Blocking

The plurality module is non-blocking.

theorem *plurality-mod-non-blocking*[*simp*]: *non-blocking plurality*
⟨*proof*⟩

5.12.4 Non-Electing

The plurality module is non-electing.

theorem *plurality-non-electing*[*simp*]: *non-electing plurality*
⟨*proof*⟩

theorem *plurality′-non-electing*[*simp*]: *non-electing plurality′*
⟨*proof*⟩

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: $a \in$ *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
$\vee$ *defer plurality* V A q = $\{a\}$
⟨*proof*⟩

The plurality rule is invariant-monotone.

theorem *plurality-mod-def-inv-mono*[*simp*]: *defer-invariant-monotonicity plurality*
⟨*proof*⟩

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*
  ⟨*proof*⟩

5.13.3 Non-Blocking

The Borda module is non-blocking.

theorem *borda-mod-non-blocking*[*simp*]: *non-blocking borda*
  ⟨*proof*⟩

5.13.4 Non-Electing

The Borda module is non-electing.

theorem *borda-mod-non-electing*[*simp*]: *non-electing borda*
  ⟨*proof*⟩

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
  ⟨proof⟩
```

5.14.3 Property

```
theorem condorcet-score-is-condorcet-rating: condorcet-rating condorcet-score
⟨proof⟩
```

```
theorem condorcet-is-dcc: defer-condorcet-consistency condorcet
⟨proof⟩
```

```
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*: *$\mathcal{SCF}$-result.electoral-module copeland*
  ⟨*proof*⟩

5.15.3 Lemmas

lemma *voters-determine-copeland-score*: *voters-determine-evaluation copeland-score*
⟨*proof*⟩

theorem *voters-determine-copeland*: *voters-determine-election copeland*
  ⟨*proof*⟩

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* $\{a \in A.\ wins\ V\ w\ p\ a\} = card\ A - 1$
⟨*proof*⟩

For a Condorcet winner w, we have: "$|\{y \in A \ . \ wins \ V \ y \ p \ w\}| = 0$".

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* $\{a \in A.\ wins\ V\ a\ p\ w\} = 0$
  ⟨*proof*⟩

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*⟩

For a non-Condorcet winner l, we have: "$|\{y \in A \ . \ wins \ V \ l \ p \ y\}| = |A| - 2$".

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⟩
```

5.15.4 Property

The Copeland score is Condorcet rating.

```
theorem copeland-score-is-cr: condorcet-rating copeland-score
⟨proof⟩

theorem copeland-is-dcc: defer-condorcet-consistency copeland
⟨proof⟩

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*: *$\mathcal{SCF}$-result.electoral-module minimax*
  ⟨*proof*⟩

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: $l \in A$ **and**
    l-neq-w: $l \neq w$
  shows *minimax-score V l A p* $\leq$ *prefer-count V p l w*
⟨*proof*⟩

5.16.4 Property

theorem *minimax-score-cond-rating*: *condorcet-rating minimax-score*
⟨*proof*⟩

theorem *minimax-is-dcc*: *defer-condorcet-consistency minimax*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*)
  ⟨*proof*⟩

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*↓)
  ⟨*proof*⟩

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⟩
```

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⟩

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*⟩

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*⟩

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*⟩

lemma *seq-comp-alt-eq*[*fundef-cong*, *code*]: *sequential-composition* = *sequential-composition'*
⟨*proof*⟩

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**

$\mathcal{SCF}$-result.electoral-module n
shows *$\mathcal{SCF}$-result.electoral-module* (*m* $\triangleright$ *n*)
⟨*proof*⟩

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: *$\mathcal{SCF}$-result.electoral-module m* **and**
module-n: *$\mathcal{SCF}$-result.electoral-module n* **and**
prof: *profile V A p* **and**
empty-defer: *defer m V A p* = {}
shows (*m* $\triangleright$ *n*) *V A p* = *m V A p*
⟨*proof*⟩

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* $\triangleright$ *n*) *V A p* = *defer m V A p*
⟨*proof*⟩

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
$\mathcal{SCF}$-result.electoral-module m **and**
$\mathcal{SCF}$-result.electoral-module n **and**
finite-profile V A p
shows *card* (*defer* (*m* $\triangleright$ *n*) *V A p*) ≤ *card* (*defer m V A p*)
⟨*proof*⟩

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
  ⟨proof⟩

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)
  ⟨proof⟩

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)
  ⟨proof⟩

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
  ⟨proof⟩

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
  ⟨proof⟩
```

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⟩
```

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⟩
```

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⟩
```

```
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: $a \in (defer\ (m \triangleright n)\ V\ A\ p) \wedge lifted\ V\ A\ p\ q\ a$
shows $(m \triangleright n)\ V\ A\ p = (m \triangleright n)\ V\ A\ q$
⟨*proof*⟩

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* $(m \triangleright n)$
⟨*proof*⟩

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* $(m \triangleright n)$
⟨*proof*⟩

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': $\mathcal{SCF}$*-result.electoral-module m'* **and**
voters-determine-m': *voters-determine-election m'*
shows *disjoint-compatibility* $(m \triangleright m')$ *n*
⟨*proof*⟩

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⟩
```

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⟩
```

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⟩
```

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⟩

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⟩
```

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⟩

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))
  ⟨proof⟩
```

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*⟩
termination
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩
termination
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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* $(m \circlearrowleft_t)$
⟨*proof*⟩

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*
⟨*proof*⟩

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
$\forall$ *q a*. *a* $\in$ (*defer* (*loop-comp-helper acc m t*) *V A p*) $\wedge$ *lifted V A p q a* $\longrightarrow$
(*loop-comp-helper acc m t*) *V A p* = (*loop-comp-helper acc m t*) *V A q*
⟨*proof*⟩

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 $\in$ *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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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* = {}
⟨*proof*⟩

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*
⟨*proof*⟩

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**
$\forall$ *r*. (*t r*) = (*card* (*defer-r r*) = *x*) **and**
x > *0* **and**
profile V A p **and**
card (*defer acc V A p*) $\geq$ *x* **and**
non-electing acc
shows *card* (*defer* (*loop-comp-helper acc m t*) *V A p*) = *x*
⟨*proof*⟩

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: $\forall$ *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* $\geq$ *x*
shows *card* (*defer* (*m* $\circlearrowleft_t$) *V A p*) = *x*
⟨*proof*⟩

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* $\circlearrowleft_t$)
⟨*proof*⟩

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⟩

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⟩

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)
  ⟨proof⟩

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)
  ⟨proof⟩
```

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⟩

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⟩

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
  ⟨proof⟩

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
  ⟨proof⟩

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
  ⟨proof⟩

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
  ⟨proof⟩

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⟩

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)
  ⟨proof⟩
```

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)
  ⟨proof⟩
```

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⟩

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⟩
```

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⟩

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))
  ⟨proof⟩
```

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)
  ⟨proof⟩

lemma voters-determine-elector:
  fixes m :: ('a, 'v, 'a Result) Electoral-Module
  assumes voters-determine-election m
  shows voters-determine-election (elector m)
  ⟨proof⟩
```

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⟩
```

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⟩

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 $\circlearrowleft_{\exists\,!d}$)

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 =
    ($\{a \in A.\ \forall\ x \in A.$ *win-count V p x* $\leq$ *win-count V p a*$\}$,
     $\{a \in A.\ \exists\ x \in A.$ *win-count V p x* $>$ *win-count V p 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*
⟨*proof*⟩

lemma *plurality-elim-equiv*:
  fixes
    A :: *'a set* **and**

V :: *'v set* **and**
p :: (*'a*, *'v*) *Profile*
assumes
$A \neq \{\}$ **and**
finite A **and**
profile V A p
shows *plurality V A p* = (*plurality-rule'*↓) *V A p*
⟨*proof*⟩

7.1.2 Soundness

theorem *plurality-rule-sound*[*simp*]: *𝒮𝒞ℱ-result.electoral-module plurality-rule*
⟨*proof*⟩

theorem *plurality-rule'-sound*[*simp*]: *𝒮𝒞ℱ-result.electoral-module plurality-rule'*
⟨*proof*⟩

lemma *voters-determine-plurality-rule*: *voters-determine-election plurality-rule*
⟨*proof*⟩

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-non-empty: $A \neq \{\}$ **and**
prof-A: *profile V A p* **and**
fin-A: *finite A*
shows *elect plurality-rule V A p* $\neq \{\}$
⟨*proof*⟩

The plurality module is electing.

theorem *plurality-rule-electing*[*simp*]: *electing plurality-rule*
⟨*proof*⟩

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: $a \in$ *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*⟩

The plurality rule is invariant-monotone.

theorem *plurality-rule-inv-mono*[*simp*]: *invariant-monotonicity plurality-rule*
⟨*proof*⟩

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*$_{\mathcal{R}}$ :: (*'a*, *'v*::*wellorder*, *'a Result*) *Electoral-Module* **where**
  borda-rule$_{\mathcal{R}}$ *V A p* = *swap-*$\mathcal{R}$ *unanimity V A p*

7.2.2 Soundness

theorem *borda-rule-sound*: $\mathcal{SCF}$*-result.electoral-module borda-rule*
  ⟨*proof*⟩

theorem *borda-rule*$_{\mathcal{R}}$*-sound*: $\mathcal{SCF}$*-result.electoral-module borda-rule*$_{\mathcal{R}}$
  ⟨*proof*⟩

7.2.3 Anonymity

theorem *borda-rule*$_{\mathcal{R}}$*-anonymous*: $\mathcal{SCF}$*-result.anonymity borda-rule*$_{\mathcal{R}}$
⟨*proof*⟩

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*) $\circlearrowleft_{\exists !d}$) *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*: *$\mathcal{SCF}$-result.electoral-module pairwise-majority-rule*
  ⟨*proof*⟩

theorem *condorcet'-rule-sound*: *$\mathcal{SCF}$-result.electoral-module condorcet'*
  ⟨*proof*⟩

theorem *pairwise-majority-rule'-sound*: *$\mathcal{SCF}$-result.electoral-module pairwise-majority-rule'*
  ⟨*proof*⟩

7.3.3 Condorcet Consistency

theorem *condorcet-condorcet*: *condorcet-consistency pairwise-majority-rule*
⟨*proof*⟩

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*
⟨*proof*⟩

7.4.3 Condorcet Consistency

theorem *copeland-condorcet*: *condorcet-consistency copeland-rule*
⟨*proof*⟩

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*: $\mathcal{SCF}$*-result.electoral-module minimax-rule*
⟨*proof*⟩

7.5.3 Condorcet Consistency

theorem *minimax-condorcet*: *condorcet-consistency minimax-rule*
⟨*proof*⟩

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*: *$\mathcal{SCF}$-result.electoral-module black*
  ⟨*proof*⟩

theorem *blacks-rule-sound*: *$\mathcal{SCF}$-result.electoral-module blacks-rule*
  ⟨*proof*⟩

7.6.3 Condorcet Consistency

theorem *black-is-dcc*: *defer-condorcet-consistency black*
  ⟨*proof*⟩

theorem *black-condorcet*: *condorcet-consistency blacks-rule*
  ⟨*proof*⟩

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) ↺∃!d) A p
```

7.7.2 Soundness

```
theorem nanson-baldwin-rule-sound: SCF-result.electoral-module nanson-baldwin-rule
  ⟨proof⟩

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) ↺∃!d) V A p
```

7.8.2 Soundness

theorem *classic-nanson-rule-sound*: *𝒮𝒞ℱ-result.electoral-module classic-nanson-rule*
⟨*proof*⟩

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*) $\circlearrowleft_{\exists !d}$) *V A p*

7.9.2 Soundness

theorem *schwartz-rule-sound*: *𝒮𝒞ℱ-result.electoral-module schwartz-rule*
⟨*proof*⟩

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⟩
```

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⟩
```

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⟩

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-$\mathcal{R}$ strong-unanimity V A p*

7.11.2 Soundness

theorem *kemeny-rule-sound*: *$\mathcal{SCF}$-result.electoral-module kemeny-rule*
⟨*proof*⟩

7.11.3 Anonymity

theorem *kemeny-rule-anonymous*: *$\mathcal{SCF}$-result.anonymity kemeny-rule*
⟨*proof*⟩

7.11.4 Neutrality

lemma *swap-dist-neutral*: *distance-neutrality well-formed-elections*
(*votewise-distance swap l-one*)
⟨*proof*⟩

theorem *kemeny-rule-neutral*: *$\mathcal{SCF}$-properties.neutrality*
well-formed-elections kemeny-rule
⟨*proof*⟩

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.