Verified Construction of Fair Voting Rules

Michael Kirsten

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

November 25, 2023

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 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.

1.1.1 Definition

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

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

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

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

fun *alts-*$\mathcal{V}$ :: *'a Vote* ⇒ *'a set* **where** *alts-*$\mathcal{V}$ *V* = *fst V*

fun *pref-*$\mathcal{V}$ :: *'a Vote* ⇒ *'a Preference-Relation* **where** *pref-*$\mathcal{V}$ *V* = *snd V*

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

by *simp*

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

1.1.2 Ranking

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

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

1.1.3 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 :: *'a* **and**
b :: *'a*
assumes
a ⪯*r* *b* **and**

```
    limited A r
  shows a ∈ A ∧ b ∈ A
  using assms
  unfolding limited-def
  by auto

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

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

lemma connex-imp-refl:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation
  assumes connex A r
  shows refl-on A r
proof
  from assms
  show r ⊆ A × A
    unfolding connex-def limited-def
    by simp
next
  fix a :: 'a
  assume a ∈ A
  with assms
  have a ⪯r a
    unfolding connex-def
    by metis
  thus (a, a) ∈ r
    by simp
qed

lemma lin-ord-imp-connex:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation
  assumes linear-order-on A r
  shows connex A r
proof (unfold connex-def limited-def, safe)
  fix
    a :: 'a and
    b :: 'a
  assume (a, b) ∈ r
  moreover have refl-on A r
    using assms partial-order-onD
    unfolding linear-order-on-def
    by safe
```

```
  ultimately show a ∈ A
    by (simp add: refl-on-domain)
next
  fix
    a :: 'a and
    b :: 'a
  assume (a, b) ∈ r
  moreover have refl-on A r
    using assms partial-order-onD
    unfolding linear-order-on-def
    by safe
  ultimately show b ∈ A
    by (simp add: refl-on-domain)
next
  fix
    a :: 'a and
    b :: 'a
  assume
    a ∈ A and
    b ∈ A and
    ¬ b ≼_r a
  moreover from this
  have (b, a) ∉ r
    by simp
  moreover from this
  have refl-on A r
    using assms partial-order-onD
    unfolding linear-order-on-def
    by blast
  ultimately have (a, b) ∈ r
    using assms refl-onD
    unfolding linear-order-on-def total-on-def
    by metis
  thus a ≼_r b
    by simp
qed

lemma connex-antsym-and-trans-imp-lin-ord:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation
  assumes
    connex-r: connex A r and
    antisym-r: antisym r and
    trans-r: trans r
  shows linear-order-on A r
proof (unfold connex-def linear-order-on-def partial-order-on-def
          preorder-on-def refl-on-def total-on-def, safe)
  fix
```

```
    a :: 'a and
    b :: 'a
  assume (a, b) ∈ r
  thus a ∈ A
    using connex-r refl-on-domain connex-imp-refl
    by metis
next
  fix
    a :: 'a and
    b :: 'a
  assume (a, b) ∈ r
  thus b ∈ A
    using connex-r refl-on-domain connex-imp-refl
    by metis
next
  fix a :: 'a
  assume a ∈ A
  thus (a, a) ∈ r
    using connex-r connex-imp-refl refl-onD
    by metis
next
  from trans-r
  show trans r
    by simp
next
  from antisym-r
  show antisym r
    by simp
next
  fix
    a :: 'a and
    b :: 'a
  assume
    a ∈ A and
    b ∈ A and
    (b, a) ∉ r
  moreover from this
  have a ≼r b ∨ b ≼r a
    using connex-r
    unfolding connex-def
    by metis
  hence (a, b) ∈ r ∨ (b, a) ∈ r
    by simp
  ultimately show (a, b) ∈ r
    by metis
qed

lemma limit-to-limits:
  fixes
```

```
    A :: 'a set and
    r :: 'a Preference-Relation
  shows limited A (limit A r)
  unfolding limited-def
  by fastforce

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

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

lemma limit-presv-trans:
  fixes
```

$A :: 'a\ set$ **and**
$r :: 'a\ Preference\text{-}Relation$
assumes *trans r*
shows *trans* (*limit A r*)
unfolding *trans-def*
using *transE assms*
by *auto*

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

lemma *limit-presv-prefs*:
fixes
$A :: 'a\ set$ **and**
$r :: 'a\ Preference\text{-}Relation$ **and**
$a :: 'a$ **and**
$b :: 'a$
assumes
$a \preceq_r b$ **and**
$a \in A$ **and**
$b \in A$
shows *let s = limit A r in* $a \preceq_s b$
using *assms*
by *simp*

lemma *limit-rel-presv-prefs*:
fixes
$A :: 'a\ set$ **and**
$r :: 'a\ Preference\text{-}Relation$ **and**
$a :: 'a$ **and**
$b :: 'a$
assumes $(a, b) \in limit\ A\ r$
shows $a \preceq_r b$
using *mem-Collect-eq assms*
by *simp*

lemma *limit-trans*:
fixes

$A :: 'a\ set$ **and**
$B :: 'a\ set$ **and**
$r :: 'a\ Preference\text{-}Relation$
assumes $A \subseteq B$
shows $limit\ A\ r = limit\ A\ (limit\ B\ r)$
using *assms*
by *auto*

lemma *lin-ord-not-empty*:
fixes $r :: 'a\ Preference\text{-}Relation$
assumes $r \neq \{\}$
shows $\neg\ linear\text{-}order\text{-}on\ \{\}\ r$
using *assms connex-imp-refl lin-ord-imp-connex refl-on-domain subrelI*
by *fastforce*

lemma *lin-ord-singleton*:
fixes $a :: 'a$
shows $\forall\ r.\ linear\text{-}order\text{-}on\ \{a\}\ r \longrightarrow r = \{(a,\ a)\}$
proof (*clarify*)
fix $r :: 'a\ Preference\text{-}Relation$
assume *lin-ord-r-a*: $linear\text{-}order\text{-}on\ \{a\}\ r$
hence $a \preceq_r a$
using *lin-ord-imp-connex singletonI*
unfolding *connex-def*
by *metis*
moreover from *lin-ord-r-a*
have $\forall\ (b,\ c) \in r.\ b = a \wedge c = a$
using *connex-imp-refl lin-ord-imp-connex refl-on-domain split-beta*
by *fastforce*
ultimately show $r = \{(a,\ a)\}$
by *auto*
qed

1.1.4 Auxiliary Lemmas

lemma *above-trans*:
fixes
$r :: 'a\ Preference\text{-}Relation$ **and**
$a :: 'a$ **and**
$b :: 'a$
assumes
$trans\ r$ **and**
$(a,\ b) \in r$
shows $above\ r\ b \subseteq above\ r\ a$
using *Collect-mono assms transE*
unfolding *above-def*
by *metis*

lemma *above-refl*:

fixes
$A :: 'a\ set$ **and**
$r :: 'a\ Preference\text{-}Relation$ **and**
$a :: 'a$
assumes
refl-on $A\ r$ **and**
$a \in A$
shows $a \in above\ r\ a$
using *assms refl-onD*
unfolding *above-def*
by *simp*

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

lemma *above-connex*:
fixes
$A :: 'a\ set$ **and**
$r :: 'a\ Preference\text{-}Relation$ **and**
$a :: 'a$
assumes
connex $A\ r$ **and**
$a \in A$
shows $a \in above\ r\ a$
using *assms connex-imp-refl above-refl*
by *metis*

lemma *pref-imp-in-above*:
fixes
$r :: 'a\ Preference\text{-}Relation$ **and**
$a :: 'a$ **and**
$b :: 'a$
shows $(a \preceq_r b) = (b \in above\ r\ a)$
unfolding *above-def*
by *simp*

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

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

lemma *above-one*:
fixes
$A :: 'a\ set$ **and**
$r :: 'a\ Preference\text{-}Relation$
assumes
lin-ord-r: *linear-order-on A r* **and**
fin-A: *finite A* **and**
non-empty-A: $A \neq \{\}$
shows $\exists\ a \in A.\ above\ r\ a = \{a\} \wedge (\forall\ a' \in A.\ above\ r\ a' = \{a'\} \longrightarrow a' = a)$
proof −
obtain $n :: nat$ **where**
len-n-plus-one: $n + 1 = card\ A$
using *Suc-eq-plus1 antisym-conv2 fin-A non-empty-A card-eq-0-iff*
gr0-implies-Suc le0
by *metis*
have $linear\text{-}order\text{-}on\ A\ r \wedge finite\ A \wedge A \neq \{\} \wedge n + 1 = card\ A \longrightarrow$
$(\exists\ a.\ a \in A \wedge above\ r\ a = \{a\})$
proof (*induction n arbitrary*: *A r*)
case *0*
show *?case*
proof (*clarify*)

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

hence *b-pref-b*: $b \preceq_r' b$
using *CollectD limit-rel-presv-prefs singletonI*
by (*metis* (*lifting*))
show $\exists\ a'.\ a' \in A' \wedge above\ r'\ a' = \{a'\}$
proof (*cases*)
assume *a-pref-r-b*: $a \preceq_r' b$
have *refl-A*:
$\forall\ A''\ r''\ a'\ a''.\ refl\text{-}on\ A''\ r'' \wedge (a'::'a,\ a'') \in r'' \longrightarrow a' \in A'' \wedge a'' \in A''$
using *refl-on-domain*
by *metis*
have *connex-refl*: $\forall\ A''\ r''.\ connex\ (A''::'a\ set)\ r'' \longrightarrow refl\text{-}on\ A''\ r''$
using *connex-imp-refl*
by *metis*
have $\forall\ A''\ r''.\ linear\text{-}order\text{-}on\ (A''::'a\ set)\ r'' \longrightarrow connex\ A''\ r''$
by (*simp add*: *lin-ord-imp-connex*)
hence *refl-A'*: *refl-on A' r'*
using *connex-refl lin-ord-r*
by *metis*
hence $a \in A' \wedge b \in A'$
using *refl-A a-pref-r-b*
by *simp*
hence *b-in-r*: $\forall\ a'.\ a' \in A' \longrightarrow b = a' \vee (b,\ a') \in r' \vee (a',\ b) \in r'$
using *lin-ord-r*
unfolding *linear-order-on-def total-on-def*
by *metis*
have *b-in-lim-B-r*: $(b,\ b) \in limit\ B\ r'$
using *alt-b mem-Collect-eq singletonI*
unfolding *above-def*
by *metis*
have *b-wins*: $\{a'.\ (b,\ a') \in limit\ B\ r'\} = \{b\}$
using *alt-b*
unfolding *above-def*
by (*metis* (*no-types*))
have *b-refl*: $(b,\ b) \in \{(a',\ a'').\ (a',\ a'') \in r' \wedge a' \in B \wedge a'' \in B\}$
using *b-in-lim-B-r*
by *simp*
moreover have *b-wins-B*: $\forall\ b' \in B.\ b \in above\ r'\ b'$
using *subset-B-card b-in-r b-wins b-refl CollectI Product-Type.Collect-case-prodD*
unfolding *above-def*
by *fastforce*
moreover have $b \in above\ r'\ a$
using *a-pref-r-b pref-imp-in-above*
by *metis*
ultimately have *b-wins*: $\forall\ a' \in A'.\ b \in above\ r'\ a'$
using *Diff-iff a empty-iff insert-iff*
by (*metis* (*no-types*))
hence $\forall\ a' \in A'.\ a' \in above\ r'\ b \longrightarrow a' = b$
using *CollectD lin-ord-r lin-imp-antisym*
unfolding *above-def antisym-def*

by *metis*
hence $\forall\ a' \in A'.\ (a' \in \mathit{above}\ r'\ b) = (a' = b)$
using *b-wins*
by *blast*
moreover have *above-b-in-A*: $\mathit{above}\ r'\ b \subseteq A'$
unfolding *above-def*
using *refl-A' refl-A*
by *auto*
ultimately have $\mathit{above}\ r'\ b = \{b\}$
using *alt-b*
unfolding *above-def*
by *fastforce*
thus *?thesis*
using *above-b-in-A*
by *blast*
next
assume $\neg\ a \preceq_r{}'\ b$
hence $b \preceq_r{}'\ a$
using *subset-B-card DiffE a lin-ord-r alt-b limit-to-limits limited-dest singletonI subset-iff lin-ord-imp-connex pref-imp-in-above*
unfolding *connex-def*
by *metis*
hence *b-smaller-a*: $(b,\ a) \in r'$
by *simp*
have *lin-ord-subset-A*:
$\forall\ B'\ B''\ r''.$
$\mathit{linear\text{-}order\text{-}on}\ (B'' :: 'a\ \mathit{set})\ r'' \wedge B' \subseteq B'' \longrightarrow$
$\mathit{linear\text{-}order\text{-}on}\ B'\ (\mathit{limit}\ B'\ r'')$
using *limit-presv-lin-ord*
by *metis*
have $\{a'.\ (b,\ a') \in \mathit{limit}\ B\ r'\} = \{b\}$
using *alt-b*
unfolding *above-def*
by *metis*
hence *b-in-B*: $b \in B$
by *auto*
have *limit-B*: $\mathit{partial\text{-}order\text{-}on}\ B\ (\mathit{limit}\ B\ r') \wedge \mathit{total\text{-}on}\ B\ (\mathit{limit}\ B\ r')$
using *lin-ord-subset-A subset-B-card lin-ord-r*
unfolding *linear-order-on-def*
by *metis*
have
$\forall\ A''\ r''.$
$\mathit{total\text{-}on}\ A''\ r'' =$
$(\forall\ a'.\ (a' :: 'a) \notin A'' \vee$
$(\forall\ a''.\ a'' \notin A'' \vee a' = a'' \vee (a',\ a'') \in r'' \vee (a'',\ a') \in r''))$
unfolding *total-on-def*
by *metis*
hence $\forall\ a'\ a''.\ a' \in B \longrightarrow a'' \in B \longrightarrow$
$a' = a'' \vee (a',\ a'') \in \mathit{limit}\ B\ r' \vee (a'',\ a') \in \mathit{limit}\ B\ r'$

using *limit-B*
by *simp*
hence $\forall\ a' \in B.\ b \in above\ r'\ a'$
using *limit-rel-presv-prefs pref-imp-in-above singletonD mem-Collect-eq lin-ord-r alt-b b-above b-pref-b subset-B-card b-in-B*
by (*metis* (*lifting*))
hence $\forall\ a' \in B.\ a' \preceq_r' b$
unfolding *above-def*
by *simp*
hence *b-wins*: $\forall\ a' \in B.\ (a', b) \in r'$
by *simp*
have *trans* r'
using *lin-ord-r lin-imp-trans*
by *metis*
hence $\forall\ a' \in B.\ (a', a) \in r'$
using *transE b-smaller-a b-wins*
by *metis*
hence $\forall\ a' \in B.\ a' \preceq_r' a$
by *simp*
hence *nothing-above-a*: $\forall\ a' \in A'.\ a' \preceq_r' a$
using *a lin-ord-r lin-ord-imp-connex above-connex Diff-iff empty-iff insert-iff pref-imp-in-above*
by *metis*
have $\forall\ a' \in A'.\ (a' \in above\ r'\ a) = (a' = a)$
using *lin-ord-r lin-imp-antisym nothing-above-a pref-imp-in-above CollectD*
unfolding *antisym-def above-def*
by *metis*
moreover have *above-a-in-A*: $above\ r'\ a \subseteq A'$
using *lin-ord-r connex-imp-refl lin-ord-imp-connex mem-Collect-eq refl-on-domain*
unfolding *above-def*
by *fastforce*
ultimately have $above\ r'\ a = \{a\}$
using *a*
unfolding *above-def*
by *blast*
thus *?thesis*
using *above-a-in-A*
by *blast*
qed
qed
qed
hence $\exists\ a.\ a \in A \wedge above\ r\ a = \{a\}$
using *fin-A non-empty-A lin-ord-r len-n-plus-one*
by *blast*
thus *?thesis*
using *assms lin-ord-imp-connex pref-imp-in-above singletonD*
unfolding *connex-def*
by *metis*
qed

lemma *above-one-eq*:
fixes
$A :: 'a\ set$ **and**
$r :: 'a\ Preference\text{-}Relation$ **and**
$a :: 'a$ **and**
$b :: 'a$
assumes
lin-ord: $linear\text{-}order\text{-}on\ A\ r$ **and**
fin-A: $finite\ A$ **and**
not-empty-A: $A \neq \{\}$ **and**
above-a: $above\ r\ a = \{a\}$ **and**
above-b: $above\ r\ b = \{b\}$
shows $a = b$
proof −
have $a \preceq_r a$
using *above-a singletonI pref-imp-in-above*
by *metis*
also have $b \preceq_r b$
using *above-b singletonI pref-imp-in-above*
by *metis*
moreover have
$\exists\ a' \in A.\ above\ r\ a' = \{a'\} \wedge (\forall\ a'' \in A.\ above\ r\ a'' = \{a''\} \longrightarrow a'' = a')$
using *lin-ord fin-A not-empty-A*
by (*simp add*: *above-one*)
moreover have $connex\ A\ r$
using *lin-ord*
by (*simp add*: *lin-ord-imp-connex*)
ultimately show $a = b$
using *above-a above-b limited-dest*
unfolding *connex-def*
by *metis*
qed

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

lemma *rank-one-imp-above-one*:
fixes
$A :: 'a\ set$ **and**
$r :: 'a\ Preference\text{-}Relation$ **and**
$a :: 'a$
assumes

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

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

lemma rank-unique:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation and
    a :: 'a and
    b :: 'a
  assumes
    lin-ord: linear-order-on A r and
    fin-A: finite A and
    a-in-A: a ∈ A and
    b-in-A: b ∈ A and
    a-neq-b: a ≠ b
  shows rank r a ≠ rank r b
proof (unfold rank.simps above-def, clarify)
  assume card-eq: card {a'. (a, a') ∈ r} = card {a'. (b, a') ∈ r}
  have refl-r: refl-on A r
```

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

lemma above-presv-limit:
  fixes
    A :: 'a set and
```

$r :: 'a$ *Preference-Relation* **and**
$a :: 'a$
shows *above* (*limit* A r) $a \subseteq A$
unfolding *above-def*
by *auto*

1.1.5 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 $r \wedge$ *linear-order-on* A $r' \wedge$ $a \in A \wedge$
$(\forall\ a' \in A - \{a\}.\ \forall\ b' \in A - \{a\}.\ (a' \preceq_r b') = (a' \preceq_r' b'))$

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' $a \wedge (\exists\ a' \in A - \{a\}.\ a \preceq_r a' \wedge a' \preceq_r' a)$

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
unfolding *equiv-rel-except-a-def*
using *assms*
by *simp*

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

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

proof (*safe*)
fix $b :: 'a$
assume
b-in-A: $b \in A$ **and**
b-neq-a: $b \neq a$ **and**
b-pref-a: $b \preceq_r a$ **and**
a-pref-b: $a \preceq_r' b$
hence *b-pref-a-rel*: $(b, a) \in r$
by *simp*
have *a-pref-b-rel*: $(a, b) \in r'$
using *a-pref-b*
by *simp*
have *antisym r*
using *assms lifted-imp-equiv-rel-except-a lin-imp-antisym*
unfolding *equiv-rel-except-a-def*
by *metis*
hence $\forall\ a'\ b'.\ (a', b') \in r \longrightarrow (b', a') \in r \longrightarrow a' = b'$
unfolding *antisym-def*
by *metis*
hence *imp-b-eq-a*: $(b, a) \in r \Longrightarrow (a, b) \in r \Longrightarrow b = a$
by *simp*
have $\exists\ a' \in A - \{a\}.\ a \preceq_r a' \wedge a' \preceq_r' a$
using *assms*
unfolding *lifted-def*
by *metis*
then obtain $c :: 'a$ **where**
$c \in A - \{a\} \wedge a \preceq_r c \wedge c \preceq_r' a$
by *metis*
hence *c-eq-r-s-exc-a*: $c \in A - \{a\} \wedge (a, c) \in r \wedge (c, a) \in r'$
by *simp*
have *equiv-r-s-exc-a*: *equiv-rel-except-a A r r' a*
using *assms*
unfolding *lifted-def*
by *metis*
hence $\forall\ a' \in A - \{a\}.\ \forall\ b' \in A - \{a\}.\ (a' \preceq_r b') = (a' \preceq_r' b')$
unfolding *equiv-rel-except-a-def*
by *metis*
hence *equiv-r-s-exc-a-rel*:
$\forall\ a' \in A - \{a\}.\ \forall\ b' \in A - \{a\}.\ ((a', b') \in r) = ((a', b') \in r')$
by *simp*
have $\forall\ a'\ b'\ c'.\ (a', b') \in r \longrightarrow (b', c') \in r \longrightarrow (a', c') \in r$
using *equiv-r-s-exc-a*
unfolding *equiv-rel-except-a-def linear-order-on-def partial-order-on-def preorder-on-def trans-def*
by *metis*
hence $(b, c) \in r'$
using *b-in-A b-neq-a b-pref-a-rel c-eq-r-s-exc-a equiv-r-s-exc-a equiv-r-s-exc-a-rel insertE insert-Diff*
unfolding *equiv-rel-except-a-def*

```
    by metis
  hence (a, c) ∈ r'
    using a-pref-b-rel b-pref-a-rel imp-b-eq-a b-neq-a equiv-r-s-exc-a
          lin-imp-trans transE
    unfolding equiv-rel-except-a-def
    by metis
  thus False
   using c-eq-r-s-exc-a equiv-r-s-exc-a antisymD DiffD2 lin-imp-antisym singletonI
    unfolding equiv-rel-except-a-def
    by metis
qed

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

```
    unfolding equiv-rel-except-a-def lifted-def
    by metis
  next
    case False
    thus ?thesis
    using a'-pref-a-rel a'-in-A rest-eq ex-lifted insertE insert-Diff
        lifted lin-imp-trans lifted-imp-equiv-rel-except-a
    unfolding equiv-rel-except-a-def trans-def
    by metis
  qed
qed

lemma lifted-above-subset:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation and
    r' :: 'a Preference-Relation and
    a :: 'a
  assumes lifted A r r' a
  shows above r' a ⊆ above r a
proof (unfold above-def, safe)
  fix a' :: 'a
  assume a-pref-x: (a, a') ∈ r'
  from assms
  have ∃ b ∈ A − {a}. a ≼r b ∧ b ≼r' a
    unfolding lifted-def
    by metis
  hence lifted-r: ∃ b ∈ A − {a}. (a, b) ∈ r ∧ (b, a) ∈ r'
    by simp
  from assms
  have ∀ b ∈ A − {a}. ∀ b' ∈ A − {a}. (b ≼r b') = (b ≼r' b')
    unfolding lifted-def equiv-rel-except-a-def
    by metis
  hence rest-eq: ∀ b ∈ A − {a}. ∀ b' ∈ A − {a}. ((b, b') ∈ r) = ((b, b') ∈ r')
    by simp
  from assms
  have trans-r: ∀ b c d. (b, c) ∈ r ⟶ (c, d) ∈ r ⟶ (b, d) ∈ r
    using lin-imp-trans
    unfolding trans-def lifted-def equiv-rel-except-a-def
    by metis
  from assms
  have trans-s: ∀ b c d. (b, c) ∈ r' ⟶ (c, d) ∈ r' ⟶ (b, d) ∈ r'
    using lin-imp-trans
    unfolding trans-def lifted-def equiv-rel-except-a-def
    by metis
  from assms
  have refl-r: (a, a) ∈ r
    using connex-imp-refl lin-ord-imp-connex refl-onD
    unfolding equiv-rel-except-a-def lifted-def
```

by *metis*
from *a-pref-x assms*
have $a' \in A$
using *connex-imp-refl lin-ord-imp-connex refl-onD2*
unfolding *equiv-rel-except-a-def lifted-def*
by *metis*
with *a-pref-x lifted-r rest-eq trans-r trans-s refl-r*
show $(a, a') \in r$
using *Diff-iff singletonD*
by (*metis* (*full-types*))
qed

lemma *lifted-above-mono*:
fixes
$A :: 'a$ *set* **and**
$r :: 'a$ *Preference-Relation* **and**
$r' :: 'a$ *Preference-Relation* **and**
$a :: 'a$ **and**
$a' :: 'a$
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 (*safe*, *simp*)
fix $b :: 'a$
assume
b-in-above-r: $b \in$ *above* $r\ a'$ **and**
b-not-in-above-s: $b \notin$ *above* $r'\ a'$
have $\forall\ b' \in A - \{a\}.\ (a' \preceq_r b') = (a' \preceq_r' b')$
using *a'-in-A-sub-a lifted-a*
unfolding *lifted-def equiv-rel-except-a-def*
by *metis*
hence $\forall\ b' \in A - \{a\}.\ (b' \in$ *above* $r\ a') = (b' \in$ *above* $r'\ a')$
unfolding *above-def*
by *simp*
hence $(b \in$ *above* $r\ a') = (b \in$ *above* $r'\ a')$
using *lifted-a b-not-in-above-s lifted-mono limited-dest lifted-def lin-ord-imp-connex*
member-remove pref-imp-in-above
unfolding *equiv-rel-except-a-def remove-def connex-def*
by *metis*
thus $b = a$
using *b-in-above-r b-not-in-above-s*
by *simp*
qed

lemma *limit-lifted-imp-eq-or-lifted*:
fixes
$A :: 'a$ *set* **and**
$A' :: 'a$ *set* **and**

$r :: 'a$ *Preference-Relation* **and**
$r' :: 'a$ *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 −
have $\forall\ a' \in A - \{a\}.\ \forall\ b' \in A - \{a\}.\ (a' \preceq_r b') = (a' \preceq_r' b')$
using *lifted subset*
unfolding *lifted-def equiv-rel-except-a-def*
by *auto*
hence *eql-rs*:
$\forall\ a' \in A - \{a\}.\ \forall\ b' \in A - \{a\}.$
$((a', b') \in (\textit{limit}\ A\ r)) = ((a', b') \in (\textit{limit}\ A\ r'))$
using *DiffD1 limit-presv-prefs limit-rel-presv-prefs*
by *simp*
have *lin-ord-r-s*: *linear-order-on* A (*limit* A r) $\wedge$ *linear-order-on* A (*limit* A r')
using *lifted subset lifted-def equiv-rel-except-a-def limit-presv-lin-ord*
by *metis*
show *?thesis*
proof (*cases*)
assume *a-in-A*: $a \in A$
thus *?thesis*
proof (*cases*)
assume $\exists\ a' \in A - \{a\}.\ a \preceq_r a' \wedge a' \preceq_r' a$
hence $\exists\ a' \in A - \{a\}.$
(*let* q = *limit* A r *in* $a \preceq_q a'$) $\wedge$ (*let* u = *limit* A r' *in* $a' \preceq_u a$)
using *DiffD1 limit-presv-prefs a-in-A*
by *simp*
thus *?thesis*
using *a-in-A eql-rs lin-ord-r-s*
unfolding *lifted-def equiv-rel-except-a-def*
by *simp*
next
assume $\neg\ (\exists\ a' \in A - \{a\}.\ a \preceq_r a' \wedge a' \preceq_r' a)$
hence *strict-pref-to-a*: $\forall\ a' \in A - \{a\}.\ \neg\ (a \preceq_r a' \wedge a' \preceq_r' a)$
by *simp*
moreover have *not-worse*: $\forall\ a' \in A - \{a\}.\ \neg\ (a' \preceq_r a \wedge a \preceq_r' a')$
using *lifted subset lifted-imp-switched*
by *fastforce*
moreover have *connex*: *connex* A (*limit* A r) $\wedge$ *connex* A (*limit* A r')
using *lifted subset limit-presv-lin-ord lin-ord-imp-connex*
unfolding *lifted-def equiv-rel-except-a-def*
by *metis*
moreover have
$\forall\ A''\ r''.$ *connex* A'' r'' =
(*limited* A'' r'' $\wedge$
$(\forall\ b\ b'.\ (b::'a) \in A'' \longrightarrow b' \in A'' \longrightarrow (b \preceq_r'' b' \vee b' \preceq_r'' b)))$

unfolding *connex-def*
by (*simp add*: *Ball-def-raw*)
hence *limit-rel-r*:
limited A (*limit A r*) ∧
(∀ *b b'*. *b* ∈ *A* ∧ *b'* ∈ *A* ⟶ (*b*, *b'*) ∈ *limit A r* ∨ (*b'*, *b*) ∈ *limit A r*)
using *connex*
by *simp*
have *limit-imp-rel*: ∀ *b b' A'' r''*. (*b*::'*a*, *b'*) ∈ *limit A'' r''* ⟶ *b* ≼$_{r}$'' *b'*
using *limit-rel-presv-prefs*
by *metis*
have *limit-rel-s*:
limited A (*limit A r'*) ∧
(∀ *b b'*. *b* ∈ *A* ∧ *b'* ∈ *A* ⟶ (*b*, *b'*) ∈ *limit A r'* ∨ (*b'*, *b*) ∈ *limit A r'*)
using *connex*
unfolding *connex-def*
by *simp*
ultimately have
∀ *a'* ∈ *A* − {*a*}. *a* ≼$_{r}$ *a'* ∧ *a* ≼$_{r}$' *a'* ∨ *a'* ≼$_{r}$ *a* ∧ *a'* ≼$_{r}$' *a*
using *DiffD1 limit-rel-r limit-rel-presv-prefs a-in-A*
by *metis*
have ∀ *a'* ∈ *A* − {*a*}. ((*a*, *a'*) ∈ (*limit A r*)) = ((*a*, *a'*) ∈ (*limit A r'*))
using *DiffD1 limit-imp-rel limit-rel-r limit-rel-s a-in-A*
strict-pref-to-a not-worse
by *metis*
hence
∀ *a'* ∈ *A* − {*a*}.
(*let q* = *limit A r in a* ≼$_{q}$ *a'*) = (*let q* = *limit A r' in a* ≼$_{q}$ *a'*)
by *simp*
moreover have
∀ *a'* ∈ *A* − {*a*}. ((*a'*, *a*) ∈ (*limit A r*)) = ((*a'*, *a*) ∈ (*limit A r'*))
using *a-in-A strict-pref-to-a not-worse DiffD1 limit-rel-presv-prefs*
limit-rel-s limit-rel-r
by *metis*
moreover have (*a*, *a*) ∈ (*limit A r*) ∧ (*a*, *a*) ∈ (*limit A r'*)
using *a-in-A connex connex-imp-refl refl-onD*
by *metis*
ultimately show *?thesis*
using *eql-rs*
by *auto*
qed
next
assume *a* ∉ *A*
thus *?thesis*
using *limit-to-limits limited-dest subrelI subset-antisym eql-rs*
by *auto*
qed
qed

lemma *negl-diff-imp-eq-limit*:

fixes
$A :: 'a\ set$ **and**
$A' :: 'a\ set$ **and**
$r :: 'a\ Preference\text{-}Relation$ **and**
$r' :: 'a\ Preference\text{-}Relation$ **and**
$a :: 'a$
assumes
change: $equiv\text{-}rel\text{-}except\text{-}a\ A'\ r\ r'\ a$ **and**
subset: $A \subseteq A'$ **and**
not-in-A: $a \notin A$
shows $limit\ A\ r = limit\ A\ r'$
proof −
have $A \subseteq A' - \{a\}$
unfolding *subset-Diff-insert*
using *not-in-A subset*
by *simp*
hence $\forall\ b \in A.\ \forall\ b' \in A.\ (b \preceq_r b') = (b \preceq_r' b')$
using *change in-mono*
unfolding *equiv-rel-except-a-def*
by *metis*
thus *?thesis*
by *auto*
qed

theorem *lifted-above-winner-alts*:
fixes
$A :: 'a\ set$ **and**
$r :: 'a\ Preference\text{-}Relation$ **and**
$r' :: 'a\ Preference\text{-}Relation$ **and**
$a :: 'a$ **and**
$a' :: 'a$
assumes
lifted-a: $lifted\ A\ r\ r'\ a$ **and**
a′-above-a′: $above\ r\ a' = \{a'\}$ **and**
fin-A: *finite A*
shows $above\ r'\ a' = \{a'\} \vee above\ r'\ a = \{a\}$
proof (*cases*)
assume $a = a'$
thus *?thesis*
using *above-subset-geq-one lifted-a a′-above-a′ lifted-above-subset*
unfolding *lifted-def equiv-rel-except-a-def*
by *metis*
next
assume *a-neq-a′*: $a \neq a'$
thus *?thesis*
proof (*cases*)
assume $above\ r'\ a' = \{a'\}$
thus *?thesis*
by *simp*

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

theorem lifted-above-winner-single:
  fixes
    A :: 'a set and
    r :: 'a Preference-Relation and
    r' :: 'a Preference-Relation and
    a :: 'a
```

assumes
lifted A r r′ a **and**
above r a = {*a*} **and**
finite A
shows *above r′ a* = {*a*}
using *assms lifted-above-winner-alts*
by *metis*

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

end

1.2 Norm

theory *Norm*
imports *HOL−Library.Extended-Real*
HOL−Combinatorics.List-Permutation
begin

A norm on R to n is a mapping N: $R \mapsto n$ on R that has the following properties:

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

1.2.1 Definition

type-synonym *Norm = ereal list ⇒ ereal*

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

1.2.2 Auxiliary Lemmas

lemma *sum-over-image-of-bijection*:
fixes
A :: 'a set **and**
A' :: 'b set **and**
f :: 'a ⇒ 'b **and**
g :: 'a ⇒ ereal
assumes *bij-betw f A A'*
shows *(∑ a ∈ A. g a) = (∑ a' ∈ A'. g (the-inv-into A f a'))*
using *assms*
proof (*induction card A arbitrary*: *A A'*)
case *0*
hence *card A' = 0*
using *bij-betw-same-card assms*
by *metis*
hence *(∑ a ∈ A. g a) = 0 ∧ (∑ a' ∈ A'. g (the-inv-into A f a')) = 0*
using *0 card-0-eq sum.empty sum.infinite*
by *metis*
thus *?case*
by *simp*
next

case (*Suc x*)
fix
  A :: *'a set* **and**
  A' :: *'b set* **and**
  x :: *nat*
assume
  IH: $\bigwedge$ *A A'*. *x* = *card A* $\Longrightarrow$
        bij-betw f A A' $\Longrightarrow$ *sum g A* = ($\sum$ *a* ∈ *A'*. *g* (*the-inv-into A f a*)) **and**
  suc: *Suc x* = *card A* **and**
  bij-A-A': *bij-betw f A A'*
obtain *a* **where**
  a-in-A: *a* ∈ *A*
  using *suc card-eq-SucD insertI1*
  by *metis*
have *a-compl-A*: *insert a* (*A* − {*a*}) = *A*
  using *a-in-A*
  by *blast*
have *inj-on-A-A'*: *inj-on f A* ∧ *A'* = *f* ‘ *A*
  using *bij-A-A'*
  unfolding *bij-betw-def*
  by *simp*
hence *inj-on-A*: *inj-on f A*
  by *simp*
have *img-of-A*: *A'* = *f* ‘ *A*
  using *inj-on-A-A'*
  by *simp*
have *inj-on f* (*insert a A*)
  using *inj-on-A a-compl-A*
  by *simp*
hence *A'-sub-fa*: *A'* − {*f a*} = *f* ‘ (*A* − {*a*})
  using *img-of-A*
  by *blast*
hence *bij-without-a*: *bij-betw f* (*A* − {*a*}) (*A'* − {*f a*})
  using *inj-on-A a-compl-A inj-on-insert*
  unfolding *bij-betw-def*
  by (*metis* (*no-types*))
have ∀ *f A A'*. *bij-betw f* (*A*::*'a set*) (*A'*::*'b set*) = (*inj-on f A* ∧ *f* ‘ *A* = *A'*)
  unfolding *bij-betw-def*
  by *simp*
hence *inv-without-a*:
  ∀ *a'* ∈ *A'* − {*f a*}. *the-inv-into* (*A* − {*a*}) *f a'* = *the-inv-into A f a'*
  using *inj-on-A A'-sub-fa*
  by (*simp add*: *inj-on-diff the-inv-into-f-eq*)
have *card-without-a*: *card* (*A* − {*a*}) = *x*
  using *suc a-in-A Diff-empty card-Diff-insert diff-Suc-1 empty-iff*
  by *simp*
hence *card-A'-from-x*: *card A'* = *Suc x* ∧ *card* (*A'* − {*f a*}) = *x*
  using *suc bij-A-A' bij-without-a*
  by (*simp add*: *bij-betw-same-card*)

hence $(\sum\ a \in A.\ g\ a) = (\sum\ a \in (A - \{a\}).\ g\ a) + g\ a$
using *suc add.commute card-Diff1-less-iff insert-Diff insert-Diff-single lessI sum.insert-remove card-without-a*
by *metis*
also have $\ldots = (\sum\ a' \in (A' - \{f\ a\}).\ g\ (\textit{the-inv-into}\ (A - \{a\})\ f\ a')) + g\ a$
using *IH bij-without-a card-without-a*
by *simp*
also have $\ldots = (\sum\ a' \in (A' - \{f\ a\}).\ g\ (\textit{the-inv-into}\ A\ f\ a')) + g\ a$
using *inv-without-a*
by *simp*
also have $\ldots = (\sum\ a' \in (A' - \{f\ a\}).\ g\ (\textit{the-inv-into}\ A\ f\ a')) +$
$g\ (\textit{the-inv-into}\ A\ f\ (f\ a))$
using *a-in-A bij-A-A'*
by (*simp add*: *bij-betw-imp-inj-on the-inv-into-f-f*)
also have $\ldots = (\sum\ a' \in A'.\ g\ (\textit{the-inv-into}\ A\ f\ a'))$
using *add.commute card-Diff1-less-iff insert-Diff insert-Diff-single lessI sum.insert-remove card-A'-from-x*
by *metis*
finally show $(\sum\ a \in A.\ g\ a) = (\sum\ a' \in A'.\ g\ (\textit{the-inv-into}\ A\ f\ a'))$
by *simp*
qed

1.2.3 Common Norms

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

1.2.4 Properties

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

1.2.5 Theorems

theorem *l-one-is-sym*: *symmetry l-one*
proof (*unfold symmetry-def*, *safe*)
fix
l :: *ereal list* **and**
l' :: *ereal list*
assume *perm*: *l* <~~> *l'*
from *perm* **obtain** π
where
$\textit{perm}_\pi$: π *permutes* {..< *length l*} **and**
l_π: *permute-list* π *l* = *l'*
using *mset-eq-permutation*
by *metis*
from $\textit{perm}_\pi$ l_π
have $(\sum\ i < \textit{length}\ l.\ |l'!i|) = (\sum\ i < \textit{length}\ l.\ |l!(\pi\ i)|)$
using *permute-list-nth*
by *fastforce*

also have $\ldots = (\sum\ i < \mathit{length}\ l.\ |l!(\pi\ (\mathit{inv}\ \pi\ i))|)$
using perm_π *permutes-inv-eq f-the-inv-into-f-bij-betw permutes-imp-bij sum.cong sum-over-image-of-bijection*
by (*smt* (*verit*, *ccfv-SIG*))
also have $\ldots = (\sum\ i < \mathit{length}\ l.\ |l!i|)$
using perm_π *permutes-inv-eq*
by *metis*
finally have $(\sum\ i < \mathit{length}\ l.\ |l'!i|) = (\sum\ i < \mathit{length}\ l.\ |l!i|)$
by *simp*
moreover have $\mathit{length}\ l = \mathit{length}\ l'$
using *perm perm-length*
by *metis*
ultimately show *l-one l = l-one l′*
using *l-one.elims*
by *metis*
qed

end

1.3 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.

1.3.1 Definition

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

type-synonym *'a Result = 'a set * 'a set * 'a set*

1.3.2 Auxiliary Functions

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

fun *set-equals-partition* :: *'a set* ⇒*'a Result* ⇒ *bool* **where**
  set-equals-partition A (*e*, *r*, *d*) = (*e* ∪ *r* ∪ *d* = *A*)

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

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

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

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

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

1.3.3 Auxiliary Lemmas

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

```
    using assms
    by simp
  ultimately show a ∈ A
    by auto
next
  fix a :: 'a
  assume
    a ∈ r and
    a ∈ e
  moreover have
    (e ∩ r = {}) ∧ (e ∩ d = {}) ∧ (r ∩ d = {}) ∧ (e ∪ r ∪ d = A)
    using assms
    by simp
  ultimately show False
    by auto
next
  fix a :: 'a
  assume
    a ∈ r and
    a ∈ d
  moreover have
    (e ∩ r = {}) ∧ (e ∩ d = {}) ∧ (r ∩ d = {}) ∧ (e ∪ r ∪ d = A)
    using assms
    by simp
  ultimately show False
    by auto
qed

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

lemma *defer-subset*:
fixes
$A :: 'a\ set$ **and**
$r :: 'a\ Result$
assumes *well-formed A r*
shows *defer-r r* $\subseteq A$
proof (*safe*)
fix $a :: 'a$
assume $a \in$ *defer-r r*
moreover obtain
$f :: 'a\ Result \Rightarrow 'a\ set \Rightarrow 'a\ set$ **and**
$g :: 'a\ Result \Rightarrow 'a\ set \Rightarrow 'a\ Result$ **where**
$A = f\ r\ A \wedge r = g\ r\ A \wedge$ *disjoint3* $(g\ r\ A) \wedge$ *set-equals-partition* $(f\ r\ A)\ (g\ r\ A)$
using *assms*
by *simp*
moreover have
$\forall\ p.\ \exists\ E\ R\ D.$ *set-equals-partition A p* $\longrightarrow (E,\ R,\ D) = p \wedge E \cup R \cup D = A$
by *simp*
ultimately show $a \in A$
using *UnCI snd-conv*
by *metis*
qed

lemma *elect-subset*:
fixes
$A :: 'a\ set$ **and**
$r :: 'a\ Result$
assumes *well-formed A r*
shows *elect-r r* $\subseteq A$
proof (*safe*)
fix $a :: 'a$
assume $a \in$ *elect-r r*
moreover obtain
$f :: 'a\ Result \Rightarrow 'a\ set \Rightarrow 'a\ set$ **and**
$g :: 'a\ Result \Rightarrow 'a\ set \Rightarrow 'a\ Result$ **where**
$A = f\ r\ A \wedge r = g\ r\ A \wedge$ *disjoint3* $(g\ r\ A) \wedge$ *set-equals-partition* $(f\ r\ A)\ (g\ r\ A)$
using *assms*
by *simp*
moreover have
$\forall\ p.\ \exists\ E\ R\ D.$ *set-equals-partition A p* $\longrightarrow (E,\ R,\ D) = p \wedge E \cup R \cup D = A$
by *simp*
ultimately show $a \in A$
using *UnCI assms fst-conv*
by *metis*
qed

lemma *reject-subset*:
fixes

```
    A :: 'a set and
    r :: 'a Result
  assumes well-formed A r
  shows reject-r r ⊆ A
proof (safe)
  fix a :: 'a
  assume a ∈ reject-r r
  moreover obtain
    f :: 'a Result ⇒ 'a set ⇒ 'a set and
    g :: 'a Result ⇒ 'a set ⇒ 'a Result where
    A = f r A ∧ r = g r A ∧ disjoint3 (g r A) ∧ set-equals-partition (f r A) (g r A)
    using assms
    by simp
  moreover have
    ∀ p. ∃ E R D. set-equals-partition A p ⟶ (E, R, D) = p ∧ E ∪ R ∪ D = A
    by simp
  ultimately show a ∈ A
    using UnCI assms prod.sel disjoint3.cases
    by metis
qed

end
```

1.4 Preference Profile

```
theory Profile
  imports Preference-Relation
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 list of such preference relations. Unlike a the common preference profiles in the social-choice sense, the profiles described here considers only the (sub-)set of alternatives that are received.

1.4.1 Definition

A profile contains one ballot for each voter.

```
type-synonym 'a Profile = ('a Preference-Relation) list

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

fun *alts-$\mathcal{E}$* :: *'a Election* ⇒ *'a set* **where** *alts-$\mathcal{E}$ E* = *fst E*

fun *prof-$\mathcal{E}$* :: *'a Election* ⇒ *'a Profile* **where** *prof-$\mathcal{E}$ E* = *snd E*

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

definition *profile* :: *'a set* ⇒ *'a Profile* ⇒ *bool* **where**
profile A p ≡ ∀ *i*::*nat*. *i* < *length p* ⟶ *linear-order-on A* (*p*!*i*)

lemma *profile-set* :
fixes
A :: *'a set* **and**
p :: *'a Profile*
shows *profile A p* ≡ (∀ *b* ∈ *set p*. *linear-order-on A b*)
unfolding *profile-def all-set-conv-all-nth*
by *simp*

abbreviation *finite-profile* :: *'a set* ⇒ *'a Profile* ⇒ *bool* **where**
finite-profile A p ≡ *finite A* ∧ *profile A p*

1.4.2 Preference Counts and Comparisons

The win count for an alternative a in a profile p is the amount of ballots in p that rank alternative a in first position.

fun *win-count* :: *'a Profile* ⇒ *'a* ⇒ *nat* **where**
win-count p a =
card {*i*::*nat*. *i* < *length p* ∧ *above* (*p*!*i*) *a* = {*a*}}

fun *win-count-code* :: *'a Profile* ⇒ *'a* ⇒ *nat* **where**
win-count-code Nil a = *0* |
win-count-code (*r*#*p*) *a* =
(*if* (*above r a* = {*a*}) *then 1 else 0*) + *win-count-code p a*

lemma *win-count-equiv*[*code*]:
fixes
p :: *'a Profile* **and**
a :: *'a*
shows *win-count p a* = *win-count-code p a*
proof (*induction p rule*: *rev-induct*, *simp*)
case (*snoc r p*)
fix
r :: *'a Preference-Relation* **and**
p :: *'a Profile*
assume *base-case*: *win-count p a* = *win-count-code p a*
have *size-one*: *length* [*r*] = *1*
by *simp*
have *p-pos*: ∀ *i* < *length p*. *p*!*i* = (*p*@[*r*])!*i*

```
  by (simp add: nth-append)
have
  win-count [r] a =
    (let q = [r] in
      card {i::nat. i < length q ∧ (let r' = (q!i) in (above r' a = {a}))})
  by simp
hence one-ballot-equiv: win-count [r] a = win-count-code [r] a
  using size-one
  by (simp add: nth-Cons')
have pref-count-induct: win-count (p@[r]) a = win-count p a + win-count [r] a
proof (simp)
  have {i. i = 0 ∧ (above ([r]!i) a = {a})} =
        (if (above r a = {a}) then {0} else {})
    by (simp add: Collect-conv-if)
  hence shift-idx-a:
    card {i. i = length p ∧ (above ([r]!0) a = {a})} =
      card {i. i = 0 ∧ (above ([r]!i) a = {a})}
    by simp
  have set-prof-eq:
    {i. i < Suc (length p) ∧ (above ((p@[r])!i) a = {a})} =
      {i. i < length p ∧ (above (p!i) a = {a})} ∪
        {i. i = length p ∧ (above ([r]!0) a = {a})}
  proof (safe, simp-all)
    fix
      n :: nat and
      a' :: 'a
    assume
      n < Suc (length p) and
      above ((p@[r])!n) a = {a} and
      n ≠ length p and
      a' ∈ above (p!n) a
    thus a' = a
      using less-antisym p-pos singletonD
      by metis
  next
    fix n :: nat
    assume
      n < Suc (length p) and
      above ((p@[r])!n) a = {a} and
      n ≠ length p
    thus a ∈ above (p!n) a
      using less-antisym insertI1 p-pos
      by metis
  next
    fix
      n :: nat and
      a' :: 'a
    assume
      n < Suc (length p) and
```

above ((*p*@[*r*])!*n*) *a* = {*a*} **and**
a′ ∈ *above r a* **and**
a′ ≠ *a*
thus *n* < *length p*
using *less-antisym nth-append-length p-pos singletonD*
by *metis*
next
fix
n :: *nat* **and**
a′ :: ′*a* **and**
a′′ :: ′*a*
assume
n < *Suc* (*length p*) **and**
above ((*p*@[*r*])!*n*) *a* = {*a*} **and**
a′ ∈ *above r a* **and**
a′ ≠ *a* **and**
a′′ ∈ *above* (*p*!*n*) *a*
thus *a*′′ = *a*
using *less-antisym p-pos nth-append-length singletonD*
by *metis*
next
fix
n :: *nat* **and**
a′ :: ′*a*
assume
n < *Suc* (*length p*) **and**
above ((*p*@[*r*])!*n*) *a* = {*a*} **and**
a′ ∈ *above r a* **and**
a′ ≠ *a*
thus *a* ∈ *above* (*p*!*n*) *a*
using *insertI1 less-antisym nth-append nth-append-length singletonD*
by *metis*
next
fix *n* :: *nat*
assume
n < *Suc* (*length p*) **and**
above ((*p*@[*r*])!*n*) *a* = {*a*} **and**
a ∉ *above r a*
thus *n* < *length p*
using *insertI1 less-antisym nth-append-length*
by *metis*
next
fix
n :: *nat* **and**
a′ :: ′*a*
assume
n < *Suc* (*length p*) **and**
above ((*p*@[*r*])!*n*) *a* = {*a*} **and**
a ∉ *above r a* **and**

```
    a' ∈ above (p!n) a
  thus a' = a
    using insertI1 less-antisym nth-append-length p-pos singletonD
    by metis
next
  fix n :: nat
  assume
    n < Suc (length p) and
    above ((p@[r])!n) a = {a} and
    a ∉ above r a
  thus a ∈ above (p!n) a
    using insertI1 less-antisym nth-append-length p-pos
    by metis
next
  fix
    n :: nat and
    a' :: 'a
  assume
    n < length p and
    above (p!n) a = {a} and
    a' ∈ above ((p@[r])!n) a
  thus a' = a
    by (simp add: nth-append)
next
  fix n :: nat
  assume
    n < length p and
    above (p!n) a = {a}
  thus a ∈ above ((p@[r])!n) a
    by (simp add: nth-append)
qed
have finite {n. n < length p ∧ (above (p!n) a = {a})}
  by simp
moreover have finite {n. n = length p ∧ (above ([r]!0) a = {a})}
  by simp
ultimately have
  card ({i. i < length p ∧ (above (p!i) a = {a})} ∪
    {i. i = length p ∧ (above ([r]!0) a = {a})}) =
      card {i. i < length p ∧ (above (p!i) a = {a})} +
        card {i. i = length p ∧ (above ([r]!0) a = {a})}
  using card-Un-disjoint
  by blast
thus
  card {i. i < Suc (length p) ∧ (above ((p@[r])!i) a = {a})} =
    card {i. i < length p ∧ (above (p!i) a = {a})} +
      card {i. i = 0 ∧ (above ([r]!i) a = {a})}
  using set-prof-eq shift-idx-a
  by auto
qed
```

```
  have win-count-code (p@[r]) a = win-count-code p a + win-count-code [r] a
  proof (induction p, simp)
    case (Cons r' q)
    fix
      r :: 'a Preference-Relation and
      r' :: 'a Preference-Relation and
      q :: 'a Profile
    assume win-count-code (q@[r']) a =
             win-count-code q a + win-count-code [r'] a
    thus win-count-code ((r#q)@[r']) a =
           win-count-code (r#q) a + win-count-code [r'] a
      by simp
  qed
  thus win-count (p@[r]) a = win-count-code (p@[r]) a
    using pref-count-induct base-case one-ballot-equiv
    by presburger
qed

fun prefer-count :: 'a Profile ⇒ 'a ⇒ 'a ⇒ nat where
  prefer-count p x y =
      card {i::nat. i < length p ∧ (let r = (p!i) in (y ≼_r x))}

fun prefer-count-code :: 'a Profile ⇒ 'a ⇒ 'a ⇒ nat where
  prefer-count-code Nil x y = 0 |
  prefer-count-code (r#p) x y =
      (if y ≼_r x then 1 else 0) + prefer-count-code p x y

lemma pref-count-equiv[code]:
  fixes
    p :: 'a Profile and
    a :: 'a and
    b :: 'a
  shows prefer-count p a b = prefer-count-code p a b
proof (induction p rule: rev-induct, simp)
  case (snoc r p)
  fix
    r :: 'a Preference-Relation and
    p :: 'a Profile
  assume base-case: prefer-count p a b = prefer-count-code p a b
  have size-one: length [r] = 1
    by simp
  have p-pos-in-ps: ∀ i < length p. p!i = (p@[r])!i
    by (simp add: nth-append)
  have prefer-count [r] a b =
         (let q = [r] in
           card {i::nat. i < length q ∧ (let r = (q!i) in (b ≼_r a))})
    by simp
  hence one-ballot-equiv: prefer-count [r] a b = prefer-count-code [r] a b
    using size-one
```

```
  by (simp add: nth-Cons')
have pref-count-induct:
  prefer-count (p@[r]) a b = prefer-count p a b + prefer-count [r] a b
proof (simp)
  have {i. i = 0 ∧ (b, a) ∈ [r]!i} = (if ((b, a) ∈ r) then {0} else {})
    by (simp add: Collect-conv-if)
  hence shift-idx-a:
    card {i. i = length p ∧ (b, a) ∈ [r]!0} = card {i. i = 0 ∧ (b, a) ∈ [r]!i}
    by simp
  have set-prof-eq:
    {i. i < Suc (length p) ∧ (b, a) ∈ (p@[r])!i} =
      {i. i < length p ∧ (b, a) ∈ p!i} ∪ {i. i = length p ∧ (b, a) ∈ [r]!0}
  proof (safe, simp-all)
    fix i :: nat
    assume
      i < Suc (length p) and
      (b, a) ∈ (p@[r])!i and
      i ≠ length p
    thus (b, a) ∈ p!i
      using less-antisym p-pos-in-ps
      by metis
  next
    fix i :: nat
    assume
      i < Suc (length p) and
      (b, a) ∈ (p@[r])!i and
      (b, a) ∉ r
    thus i < length p
      using less-antisym nth-append-length
      by metis
  next
    fix i :: nat
    assume
      i < Suc (length p) and
      (b, a) ∈ (p@[r])!i and
      (b, a) ∉ r
    thus (b, a) ∈ p!i
      using less-antisym nth-append-length p-pos-in-ps
      by metis
  next
    fix i :: nat
    assume
      i < length p and
      (b, a) ∈ p!i
    thus (b, a) ∈ (p@[r])!i
      using less-antisym p-pos-in-ps
      by metis
  qed
  have fin-len-p: finite {n. n < length p ∧ (b, a) ∈ p!n}
```

```
    by simp
  have finite {n. n = length p ∧ (b, a) ∈ [r]!0}
    by simp
  hence
    card ({i. i < length p ∧ (b, a) ∈ p!i} ∪ {i. i = length p ∧ (b, a) ∈ [r]!0}) =
        card {i. i < length p ∧ (b, a) ∈ p!i} +
          card {i. i = length p ∧ (b, a) ∈ [r]!0}
    using fin-len-p card-Un-disjoint
    by blast
  thus
    card {i. i < Suc (length p) ∧ (b, a) ∈ (p@[r])!i} =
      card {i. i < length p ∧ (b, a) ∈ p!i} + card {i. i = 0 ∧ (b, a) ∈ [r]!i}
    using set-prof-eq shift-idx-a
    by simp
qed
have pref-count-code-induct:
  prefer-count-code (p@[r]) a b =
    prefer-count-code p a b + prefer-count-code [r] a b
proof (simp, safe)
  assume y-pref-x: (b, a) ∈ r
  show prefer-count-code (p@[r]) a b = Suc (prefer-count-code p a b)
  proof (induction p, simp-all)
    show (b, a) ∈ r
      using y-pref-x
      by simp
  qed
next
  assume not-y-pref-x: (b, a) ∉ r
  show prefer-count-code (p@[r]) a b = prefer-count-code p a b
  proof (induction p, simp-all, safe)
    assume (b, a) ∈ r
    thus False
      using not-y-pref-x
      by simp
  qed
qed
show prefer-count (p@[r]) a b = prefer-count-code (p@[r]) a b
  using pref-count-code-induct pref-count-induct base-case one-ballot-equiv
  by presburger
qed

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

lemma pref-count-set-compr:
```

fixes
$A :: 'a\ set$ **and**
$p :: 'a\ Profile$ **and**
$a :: 'a$
shows $\{prefer\text{-}count\ p\ a\ a' \mid a'.\ a' \in A - \{a\}\} = (prefer\text{-}count\ p\ a)\ `\ (A - \{a\})$
by *auto*

lemma *pref-count*:
fixes
$A :: 'a\ set$ **and**
$p :: 'a\ Profile$ **and**
$a :: 'a$ **and**
$b :: 'a$
assumes
prof: $profile\ A\ p$ **and**
a-in-A: $a \in A$ **and**
b-in-A: $b \in A$ **and**
neq: $a \neq b$
shows $prefer\text{-}count\ p\ a\ b = (length\ p) - (prefer\text{-}count\ p\ b\ a)$
proof −
have $\forall\ i{::}nat.\ i < length\ p \longrightarrow connex\ A\ (p!i)$
using *prof*
unfolding *profile-def*
by (*simp add*: *lin-ord-imp-connex*)
hence *asym*: $\forall\ i{::}nat.\ i < length\ p \longrightarrow$
$(let\ r = (p!i)\ in\ (\neg\ b \preceq_r a)) \longrightarrow (let\ r = (p!i)\ in\ (a \preceq_r b))$
using *a-in-A b-in-A*
unfolding *connex-def*
by *metis*
have $\forall\ i{::}nat.\ i < length\ p \longrightarrow (b, a) \in (p!i) \longrightarrow (a, b) \notin (p!i)$
using *antisymD neq lin-imp-antisym prof*
unfolding *profile-def*
by *metis*
hence $\{i{::}nat.\ i < length\ p \wedge (let\ r = (p!i)\ in\ (b \preceq_r a))\} =$
$\{i{::}nat.\ i < length\ p\} -$
$\{i{::}nat.\ i < length\ p \wedge (let\ r = (p!i)\ in\ (a \preceq_r b))\}$
using *asym*
by *auto*
thus *?thesis*
by (*simp add*: *card-Diff-subset Collect-mono*)
qed

lemma *pref-count-sym*:
fixes
$p :: 'a\ Profile$ **and**
$a :: 'a$ **and**
$b :: 'a$ **and**
$c :: 'a$
assumes

pref-count-ineq: *prefer-count p a c* $\geq$ *prefer-count p c b* **and**
prof: *profile A p* **and**
a-in-A: $a \in A$ **and**
b-in-A: $b \in A$ **and**
c-in-A: $c \in A$ **and**
a-neq-c: $a \neq c$ **and**
c-neq-b: $c \neq b$
shows *prefer-count p b c* $\geq$ *prefer-count p c a*
proof −
have *prefer-count p a c* = (*length p*) − (*prefer-count p c a*)
using *pref-count prof a-in-A c-in-A a-neq-c*
by *metis*
moreover have *pref-count-b-eq*:
prefer-count p c b = (*length p*) − (*prefer-count p b c*)
using *pref-count prof c-in-A b-in-A c-neq-b*
by (*metis* (*mono-tags*, *lifting*))
hence (*length p*) − (*prefer-count p b c*) $\leq$ (*length p*) − (*prefer-count p c a*)
using *calculation pref-count-ineq*
by *simp*
hence (*prefer-count p c a*) − (*length p*) $\leq$ (*prefer-count p b c*) − (*length p*)
using *a-in-A diff-is-0-eq diff-le-self a-neq-c pref-count prof c-in-A*
by (*metis* (*no-types*))
thus *?thesis*
using *pref-count-b-eq calculation pref-count-ineq*
by *linarith*
qed

lemma *empty-prof-imp-zero-pref-count*:
fixes
p :: *'a Profile* **and**
a :: *'a* **and**
b :: *'a*
assumes *p* = []
shows *prefer-count p a b* = *0*
using *assms*
by *simp*

lemma *empty-prof-imp-zero-pref-count-code*:
fixes
p :: *'a Profile* **and**
a :: *'a* **and**
b :: *'a*
assumes *p* = []
shows *prefer-count-code p a b* = *0*
using *assms*
by *simp*

lemma *pref-count-code-incr*:
fixes

```
    p :: 'a Profile and
    r :: 'a Preference-Relation and
    a :: 'a and
    b :: 'a and
    n :: nat
  assumes
    prefer-count-code p a b = n and
    b ≼r a
  shows prefer-count-code (r#p) a b = n + 1
  using assms
  by simp

lemma pref-count-code-not-smaller-imp-constant:
  fixes
    p :: 'a Profile and
    r :: 'a Preference-Relation and
    a :: 'a and
    b :: 'a and
    n :: nat
  assumes
    prefer-count-code p a b = n and
    ¬ (b ≼r a)
  shows prefer-count-code (r#p) a b = n
  using assms
  by simp

fun wins :: 'a ⇒ 'a Profile ⇒ 'a ⇒ bool where
  wins a p b =
    (prefer-count p a b > prefer-count p b a)
```

Alternative a wins against b implies that b does not win against a.

```
lemma wins-antisym:
  fixes
    p :: 'a Profile and
    a :: 'a and
    b :: 'a
  assumes wins a p b
  shows ¬ wins b p a
  using assms
  by simp

lemma wins-irreflex:
  fixes
    p :: 'a Profile and
    a :: 'a
  shows ¬ wins a p a
  using wins-antisym
  by metis
```

1.4.3 Condorcet Winner

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

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

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

using *assms*
by *presburger*
qed

1.4.4 Limited Profile

This function restricts a profile p to a set A and keeps all of A's preferences.

fun *limit-profile* :: *'a set* ⇒ *'a Profile* ⇒ *'a Profile* **where**
limit-profile A p = *map* (*limit A*) *p*

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

lemma *limit-profile-sound*:
fixes
A :: *'a set* **and**
B :: *'a set* **and**
p :: *'a Profile*
assumes
profile: *profile B p* **and**
subset: *A* ⊆ *B*
shows *profile A* (*limit-profile A p*)
proof (*clarsimp*)
have *prof-is-lin-ord*:
∀ *A' p'*.
(*profile* (*A'*::*'a set*) *p'* ⟶ (∀ *n* < *length p'*. *linear-order-on A'* (*p'*!*n*))) ∧
((∀ *n* < *length p'*. *linear-order-on A'* (*p'*!*n*)) ⟶ *profile A' p'*)
unfolding *profile-def*
by *simp*
have *limit-prof-simp*: *limit-profile A p* = *map* (*limit A*) *p*
by *simp*
obtain *n* :: *nat* **where**
prof-limit-n: *n* < *length* (*limit-profile A p*) ⟶
linear-order-on A (*limit-profile A p*!*n*) ⟶ *profile A* (*limit-profile A p*)
using *prof-is-lin-ord*
by *metis*
have *prof-n-lin-ord*: ∀ *n* < *length p*. *linear-order-on B* (*p*!*n*)
using *prof-is-lin-ord profile*
by *simp*

```
  have prof-length: length p = length (map (limit A) p)
    by simp
  have n < length p ⟶ linear-order-on B (p!n)
    using prof-n-lin-ord
    by simp
  thus profile A (map (limit A) p)
    using prof-length prof-limit-n limit-prof-simp
          limit-presv-lin-ord nth-map subset profile
    unfolding profile-def
    by (metis (no-types))
qed

lemma limit-prof-presv-size:
  fixes
    A :: 'a set and
    p :: 'a Profile
  shows length p = length (limit-profile A p)
  by simp
```

1.4.5 Lifting Property

```
definition equiv-prof-except-a :: 'a set ⇒ 'a Profile ⇒ 'a Profile ⇒ 'a ⇒ bool
where
  equiv-prof-except-a A p p' a ≡
    profile A p ∧ profile A p' ∧ a ∈ A ∧ length p = length p' ∧
      (∀ i::nat. i < length p ⟶ equiv-rel-except-a A (p!i) (p'!i) 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 :: 'a set ⇒ 'a Profile ⇒ 'a Profile ⇒ 'a ⇒ bool where
  lifted A p p' a ≡
    profile A p ∧ profile A p' ∧ finite A ∧ a ∈ A ∧ length p = length p' ∧
      (∀ i::nat. i < length p ∧ ¬ Preference-Relation.lifted A (p!i) (p'!i) a ⟶
         (p!i) = (p'!i)) ∧
      (∃ i::nat. i < length p ∧ Preference-Relation.lifted A (p!i) (p'!i) a)

lemma lifted-imp-equiv-prof-except-a:
  fixes
    A :: 'a set and
    p :: 'a Profile and
    p' :: 'a Profile and
    a :: 'a
  assumes lifted A p p' a
  shows equiv-prof-except-a A p p' a
proof (unfold equiv-prof-except-a-def, safe)
  from assms
  show profile A p
    unfolding lifted-def
    by metis
```

next
from *assms*
show *profile A p'*
unfolding *lifted-def*
by *metis*
next
from *assms*
show $a \in A$
unfolding *lifted-def*
by *metis*
next
from *assms*
show *length p = length p'*
unfolding *lifted-def*
by *metis*
next
fix $i :: nat$
assume $i <$ *length p*
with *assms*
show *equiv-rel-except-a A (p!i) (p'!i) a*
using *lifted-imp-equiv-rel-except-a trivial-equiv-rel*
unfolding *lifted-def profile-def*
by (*metis* (*no-types*))
qed

lemma *negl-diff-imp-eq-limit-prof*:
fixes
$A :: 'a$ *set* **and**
$A' :: 'a$ *set* **and**
$p :: 'a$ *Profile* **and**
$p' :: 'a$ *Profile* **and**
$a :: 'a$
assumes
change: *equiv-prof-except-a A' p q a* **and**
subset: $A \subseteq A'$ **and**
not-in-A: $a \notin A$
shows *limit-profile A p = limit-profile A q*
proof (*simp*)
have $\forall$ *i*::*nat*. $i <$ *length p* $\longrightarrow$ *equiv-rel-except-a A' (p!i) (q!i) a*
using *change equiv-prof-except-a-def*
by *metis*
hence $\forall$ *i*::*nat*. $i <$ *length p* $\longrightarrow$ *limit A (p!i) = limit A (q!i)*
using *not-in-A negl-diff-imp-eq-limit subset*
by *metis*
thus *map* (*limit A*) *p = map* (*limit A*) *q*
using *change equiv-prof-except-a-def length-map nth-equalityI nth-map*
by (*metis* (*mono-tags*, *lifting*))
qed

lemma *limit-prof-eq-or-lifted*:
fixes
$A :: 'a\ set$ **and**
$A' :: 'a\ set$ **and**
$p :: 'a\ Profile$ **and**
$p' :: 'a\ Profile$ **and**
$a :: 'a$
assumes
lifted-a: $lifted\ A'\ p\ p'\ a$ **and**
subset: $A \subseteq A'$
shows $limit\text{-}profile\ A\ p = limit\text{-}profile\ A\ p' \vee$
$lifted\ A\ (limit\text{-}profile\ A\ p)\ (limit\text{-}profile\ A\ p')\ a$
proof (*cases*)
assume *a-in-A*: $a \in A$
have $\forall\ i{::}nat.\ i < length\ p \longrightarrow$
$(Preference\text{-}Relation.lifted\ A'\ (p!i)\ (p'!i)\ a \vee (p!i) = (p'!i))$
using *lifted-a*
unfolding *lifted-def*
by *metis*
hence *one*:
$\forall\ i{::}nat.\ i < length\ p \longrightarrow$
$Preference\text{-}Relation.lifted\ A\ (limit\ A\ (p!i))\ (limit\ A\ (p'!i))\ a \vee$
$(limit\ A\ (p!i)) = (limit\ A\ (p'!i))$
using *limit-lifted-imp-eq-or-lifted subset*
by *metis*
thus *?thesis*
proof (*cases*)
assume $\forall\ i{::}nat.\ i < length\ p \longrightarrow (limit\ A\ (p!i)) = (limit\ A\ (p'!i))$
thus *?thesis*
using *length-map lifted-a nth-equalityI nth-map limit-profile.simps*
unfolding *lifted-def*
by (*metis* (*mono-tags*, *lifting*))
next
assume *forall-limit-p-q*:
$\neg\ (\forall\ i{::}nat.\ i < length\ p \longrightarrow (limit\ A\ (p!i)) = (limit\ A\ (p'!i)))$
let $?p = limit\text{-}profile\ A\ p$
let $?q = limit\text{-}profile\ A\ p'$
have $profile\ A\ ?p \wedge profile\ A\ ?q$
using *lifted-a subset limit-presv-lin-ord limit-prof-presv-size*
limit-profile.elims nth-map
unfolding *profile-def lifted-def*
by (*metis* (*mono-tags*, *lifting*))
moreover have $length\ ?p = length\ ?q$
using *lifted-a*
unfolding *lifted-def*
by *fastforce*
moreover have
$\exists\ i{::}nat.\ i < length\ ?p \wedge Preference\text{-}Relation.lifted\ A\ (?p!i)\ (?q!i)\ a$
using *forall-limit-p-q length-map lifted-a limit-profile.simps nth-map one*

unfolding *lifted-def*
by (*metis* (*no-types*, *lifting*))
moreover have
$\forall$ *i*::*nat*.
i < *length ?p* $\wedge$ $\neg$ *Preference-Relation.lifted A* (*?p*!*i*) (*?q*!*i*) *a* $\longrightarrow$
(*?p*!*i*) = (*?q*!*i*)
using *length-map lifted-a limit-profile.simps nth-map one*
unfolding *lifted-def*
by *metis*
ultimately have *lifted A ?p ?q a*
using *a-in-A lifted-a subset infinite-super*
unfolding *lifted-def*
by (*metis* (*no-types*, *lifting*))
thus *?thesis*
by *simp*
qed
next
assume *a* $\notin$ *A*
thus *?thesis*
using *lifted-a negl-diff-imp-eq-limit-prof subset lifted-imp-equiv-prof-except-a*
by *metis*
qed

end

1.5 Preference List

theory *Preference-List*
imports *../Preference-Relation*
List−Index.List-Index
begin

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

1.5.1 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*

1.5.2 Auxiliary Lemmas About Lists

lemma *is-arg-min-equal*:
fixes
f :: *'a* $\Rightarrow$ *'b*::*ord* **and**

```
    g :: 'a ⇒ 'b and
    S :: 'a set and
    x :: 'a
  assumes ∀ x ∈ S. f x = g x
  shows is-arg-min f (λ s. s ∈ S) x = is-arg-min g (λ s. s ∈ S) x
proof (unfold is-arg-min-def, cases x ∉ S, clarsimp)
  case x-in-S: False
  thus (x ∈ S ∧ (∄ y. y ∈ S ∧ f y < f x)) = (x ∈ S ∧ (∄ y. y ∈ S ∧ g y < g x))
  proof (cases ∃ y. (λ s. s ∈ S) y ∧ f y < f x)
    case y: True
    then obtain y :: 'a where
      (λ s. s ∈ S) y ∧ f y < f x
      by metis
    hence (λ s. s ∈ S) y ∧ g y < g x
      using x-in-S assms
      by metis
    thus ?thesis
      using y
      by metis
  next
    case not-y: False
    have ¬ (∃ y. (λ s. s ∈ S) y ∧ g y < g x)
    proof (safe)
      fix y :: 'a
      assume
        y-in-S: y ∈ S and
        g-y-lt-g-x: g y < g x
      have f-eq-g-for-elems-in-S: ∀ a. a ∈ S ⟶ f a = g a
        using assms
        by simp
      hence g x = f x
        using x-in-S
        by presburger
      thus False
        using f-eq-g-for-elems-in-S g-y-lt-g-x not-y y-in-S
        by (metis (no-types))
    qed
    thus ?thesis
      using x-in-S not-y
      by simp
  qed
qed

lemma list-cons-presv-finiteness:
  fixes
    A :: 'a set and
    S :: 'a list set
  assumes
    fin-A: finite A and
```

fin-B: *finite S*
shows *finite* $\{a\#l \mid a\ l.\ a \in A \wedge l \in S\}$
proof −
let *?P* $= \lambda\ A.$ *finite* $\{a\#l \mid a\ l.\ a \in A \wedge l \in S\}$
have $\forall\ a\ A'.$ *finite* $A' \longrightarrow a \notin A' \longrightarrow$ *?P* $A' \longrightarrow$ *?P* (*insert a* A')
proof (*safe*)
fix
$a :: 'a$ **and**
$A' :: 'a$ *set*
assume *finite* $\{a\#l \mid a\ l.\ a \in A' \wedge l \in S\}$
moreover have
$\{a'\#l \mid a'\ l.\ a' \in$ *insert a* $A' \wedge l \in S\} =$
$\{a\#l \mid a\ l.\ a \in A' \wedge l \in S\} \cup \{a\#l \mid l.\ l \in S\}$
by *blast*
moreover have *finite* $\{a\#l \mid l.\ l \in S\}$
using *fin-B*
by *simp*
ultimately have *finite* $\{a'\#l \mid a'\ l.\ a' \in$ *insert a* $A' \wedge l \in S\}$
by *simp*
thus *?P* (*insert a* A')
by *simp*
qed
moreover have *?P* $\{\}$
by *simp*
ultimately show *?P A*
using *finite-induct*[*of A ?P*] *fin-A*
by *simp*
qed

lemma *listset-finiteness*:
fixes $l :: 'a$ *set list*
assumes $\forall\ i{::}nat.\ i <$ *length* $l \longrightarrow$ *finite* $(l!i)$
shows *finite* (*listset l*)
using *assms*
proof (*induct l*, *simp*)
case (*Cons a l*)
fix
$a :: 'a$ *set* **and**
$l :: 'a$ *set list*
assume
elems-fin-then-set-fin: $\forall\ i{::}nat <$ *length l*. *finite* $(l!i) \Longrightarrow$ *finite* (*listset l*) **and**
fin-all-elems: $\forall\ i{::}nat <$ *length* $(a\#l)$. *finite* $((a\#l)!i)$
hence *finite a*
by *auto*
moreover from *fin-all-elems*
have $\forall\ i <$ *length l*. *finite* $(l!i)$
by *auto*
hence *finite* (*listset l*)
using *elems-fin-then-set-fin*

by *simp*
ultimately have *finite* $\{a'\#l' \mid a'\ l'.\ a' \in a \wedge l' \in (listset\ l)\}$
using *list-cons-presv-finiteness*
by *auto*
thus *finite* $(listset\ (a\#l))$
by (*simp add*: *set-Cons-def*)
qed

lemma *all-ls-elems-same-len*:
fixes $l :: 'a\ set\ list$
shows $\forall\ l'::('a\ list).\ l' \in listset\ l \longrightarrow length\ l' = length\ l$
proof (*induct l*, *simp*)
case (*Cons a l*)
fix
$a :: 'a\ set$ **and**
$l :: 'a\ set\ list$
assume $\forall\ l'.\ l' \in listset\ l \longrightarrow length\ l' = length\ l$
moreover have
$\forall\ a'\ l'::('a\ set\ list).\ listset\ (a'\#l') = \{b\#m \mid b\ m.\ b \in a' \wedge m \in listset\ l'\}$
by (*simp add*: *set-Cons-def*)
ultimately show $\forall\ l'.\ l' \in listset\ (a\#l) \longrightarrow length\ l' = length\ (a\#l)$
using *local.Cons*
by *force*
qed

lemma *all-ls-elems-in-ls-set*:
fixes $l :: 'a\ set\ list$
shows $\forall\ l'\ i::nat.\ l' \in listset\ l \wedge i < length\ l' \longrightarrow l'!i \in l!i$
proof (*induct l*, *simp*, *safe*)
case (*Cons a l*)
fix
$a :: 'a\ set$ **and**
$l :: 'a\ set\ list$ **and**
$l' :: 'a\ list$ **and**
$i :: nat$
assume *elems-in-set-then-elems-pos*:
$\forall\ l'\ i::nat.\ l' \in listset\ l \wedge i < length\ l' \longrightarrow l'!i \in l!i$ **and**
l-prime-in-set-a-l: $l' \in listset\ (a\#l)$ **and**
i-lt-len-l-prime: $i < length\ l'$
have $l' \in set\text{-}Cons\ a\ (listset\ l)$
using *l-prime-in-set-a-l*
by *simp*
hence $l' \in \{m.\ \exists\ b\ m'.\ m = b\#m' \wedge b \in a \wedge m' \in (listset\ l)\}$
unfolding *set-Cons-def*
by *simp*
hence $\exists\ b\ m.\ l' = b\#m \wedge b \in a \wedge m \in (listset\ l)$
by *simp*
thus $l'!i \in (a\#l)!i$
using *elems-in-set-then-elems-pos i-lt-len-l-prime nth-Cons-Suc*

Suc-less-eq gr0-conv-Suc length-Cons nth-non-equal-first-eq
by *metis*
qed

lemma *all-ls-in-ls-set*:
fixes $l :: 'a\ set\ list$
shows $\forall\ l'.\ length\ l' = length\ l \wedge (\forall\ i < length\ l'.\ l'!i \in l!i) \longrightarrow l' \in listset\ l$
proof (*induction l*, *safe*, *simp*)
case (*Cons a l*)
fix
$l :: 'a\ set\ list$ **and**
$l' :: 'a\ list$ **and**
$s :: 'a\ set$
assume
all-ls-in-ls-set-induct:
$\forall\ m.\ length\ m = length\ l \wedge (\forall\ i < length\ m.\ m!i \in l!i) \longrightarrow m \in listset\ l$ **and**
len-eq: $length\ l' = length\ (s\#l)$ **and**
elems-pos-in-cons-ls-pos: $\forall\ i < length\ l'.\ l'!i \in (s\#l)!i$
then obtain t **and** x **where**
l'-cons: $l' = x\#t$
using *length-Suc-conv*
by *metis*
hence $x \in s$
using *elems-pos-in-cons-ls-pos*
by *force*
moreover have $t \in listset\ l$
using *l'-cons all-ls-in-ls-set-induct len-eq diff-Suc-1 diff-Suc-eq-diff-pred*
elems-pos-in-cons-ls-pos length-Cons nth-Cons-Suc zero-less-diff
by *metis*
ultimately show $l' \in listset\ (s\#l)$
using *l'-cons*
unfolding *listset-def set-Cons-def*
by *simp*
qed

1.5.3 Ranking

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

fun *rank-l* :: $'a\ Preference\text{-}List \Rightarrow 'a \Rightarrow nat$ **where**
$rank\text{-}l\ l\ a = (if\ a \in set\ l\ then\ index\ l\ a + 1\ else\ 0)$

fun *rank-l-idx* :: $'a\ Preference\text{-}List \Rightarrow 'a \Rightarrow nat$ **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*
by (*simp add*: *ext index-size-conv member-def*)

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

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

1.5.4 Definition

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

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

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

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

1.5.5 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 :: *'a* **and**

```
    b :: 'a
  assumes
    a ≲l b and
    limited A l
  shows a ∈ A ∧ b ∈ A
  using assms
  unfolding limited-def
  by simp

lemma limit-equiv:
  fixes
    A :: 'a set and
    l :: 'a list
  assumes well-formed-l l
  shows pl-α (limit-l A l) = limit A (pl-α l)
  using assms
proof (induction l)
  case Nil
  thus pl-α (limit-l A []) = limit A (pl-α [])
    unfolding pl-α-def
    by simp
next
  case (Cons a l)
  fix
    a :: 'a and
    l :: 'a list
  assume
    wf-imp-limit: well-formed-l l ⟹ pl-α (limit-l A l) = limit A (pl-α l) and
    wf-a-l: well-formed-l (a#l)
  show pl-α (limit-l A (a#l)) = limit A (pl-α (a#l))
    using wf-imp-limit wf-a-l
  proof (clarsimp, safe)
    fix
      b :: 'a and
      c :: 'a
    assume b-less-c: (b, c) ∈ pl-α (a#(filter (λ a. a ∈ A) l))
    have limit-preference-list-assoc: pl-α (limit-l A l) = limit A (pl-α l)
      using wf-a-l wf-imp-limit
      by simp
    thus (b, c) ∈ pl-α (a#l)
    proof (unfold pl-α-def is-less-preferred-than-l.simps, safe)
      show b ∈ set (a#l)
        using b-less-c
        unfolding pl-α-def
        by fastforce
    next
      show c ∈ set (a#l)
        using b-less-c
        unfolding pl-α-def
```

by *fastforce*
next
have $\forall\ a'\ l'\ a''.\ ((a'::'a) \lesssim_{l'} a'') =$
$(a' \in set\ l' \wedge a'' \in set\ l' \wedge index\ l'\ a'' \leq index\ l'\ a')$
using *is-less-preferred-than-l.simps*
by *blast*
moreover from *this*
have $\{(a',\ b').\ a' \lesssim_{(} limit\text{-}l\ A\ l)\ b'\} =$
$\{(a',\ a'').\ a' \in set\ (limit\text{-}l\ A\ l) \wedge a'' \in set\ (limit\text{-}l\ A\ l) \wedge$
$index\ (limit\text{-}l\ A\ l)\ a'' \leq index\ (limit\text{-}l\ A\ l)\ a'\}$
by *presburger*
moreover from *this* **have**
$\{(a',\ b').\ a' \lesssim_l b'\} =$
$\{(a',\ a'').\ a' \in set\ l \wedge a'' \in set\ l \wedge index\ l\ a'' \leq index\ l\ a'\}$
using *is-less-preferred-than-l.simps*
by *auto*
ultimately have $\{(a',\ b').$
$a' \in set\ (limit\text{-}l\ A\ l) \wedge b' \in set\ (limit\text{-}l\ A\ l) \wedge$
$index\ (limit\text{-}l\ A\ l)\ b' \leq index\ (limit\text{-}l\ A\ l)\ a'\} =$
$limit\ A\ \{(a',\ b').\ a' \in set\ l \wedge b' \in set\ l \wedge index\ l\ b' \leq index\ l\ a'\}$
using *pl-α-def limit-preference-list-assoc*
by (*metis* (*no-types*))
hence *idx-imp*:
$b \in set\ (limit\text{-}l\ A\ l) \wedge c \in set\ (limit\text{-}l\ A\ l) \wedge$
$index\ (limit\text{-}l\ A\ l)\ c \leq index\ (limit\text{-}l\ A\ l)\ b \longrightarrow$
$b \in set\ l \wedge c \in set\ l \wedge index\ l\ c \leq index\ l\ b$
by *auto*
have $b \lesssim_{(} a\#(filter\ (\lambda\ a.\ a \in A)\ l))\ c$
using *b-less-c case-prodD mem-Collect-eq*
unfolding *pl-α-def*
by *metis*
moreover obtain
$f :: 'a \Rightarrow 'a\ list \Rightarrow 'a \Rightarrow 'a$ **and**
$g :: 'a \Rightarrow 'a\ list \Rightarrow 'a \Rightarrow 'a\ list$ **and**
$h :: 'a \Rightarrow 'a\ list \Rightarrow 'a \Rightarrow 'a$ **where**
$\forall\ d\ s\ e.\ d \lesssim_s e \longrightarrow$
$d = f\ e\ s\ d \wedge s = g\ e\ s\ d \wedge e = h\ e\ s\ d \wedge f\ e\ s\ d \in set\ (g\ e\ s\ d) \wedge$
$index\ (g\ e\ s\ d)\ (h\ e\ s\ d) \leq index\ (g\ e\ s\ d)\ (f\ e\ s\ d) \wedge$
$h\ e\ s\ d \in set\ (g\ e\ s\ d)$
by *fastforce*
ultimately have
$b = f\ c\ (a\#(filter\ (\lambda\ a.\ a \in A)\ l))\ b \wedge$
$a\#(filter\ (\lambda\ a.\ a \in A)\ l) = g\ c\ (a\#(filter\ (\lambda\ a.\ a \in A)\ l))\ b \wedge$
$c = h\ c\ (a\#(filter\ (\lambda\ a.\ a \in A)\ l))\ b \wedge$
$f\ c\ (a\#(filter\ (\lambda\ a.\ a \in A)\ l))\ b \in set\ (g\ c\ (a\#(filter\ (\lambda\ a.\ a \in A)\ l))\ b) \wedge$
$h\ c\ (a\#(filter\ (\lambda\ a.\ a \in A)\ l))\ b \in set\ (g\ c\ (a\#(filter\ (\lambda\ a.\ a \in A)\ l))\ b) \wedge$
$index\ (g\ c\ (a\#(filter\ (\lambda\ a.\ a \in A)\ l))\ b)$
$(h\ c\ (a\#(filter\ (\lambda\ a.\ a \in A)\ l))\ b) \leq$
$index\ (g\ c\ (a\#(filter\ (\lambda\ a.\ a \in A)\ l))\ b)$

```
          (f c (a#(filter (λ a. a ∈ A) l)) b)
      by blast
    moreover have filter (λ a. a ∈ A) l = limit-l A l
      by simp
    ultimately have a ≠ c ⟶ index (a#l) c ≤ index (a#l) b
      using idx-imp
      by force
    thus index (a#l) c ≤ index (a#l) b
      by force
  qed
next
  fix
    b :: 'a and
    c :: 'a
  assume
     a ∈ A and
    (b, c) ∈ pl-α (a#(filter (λ a. a ∈ A) l))
  thus c ∈ A
    unfolding pl-α-def
    by fastforce
next
  fix
    b :: 'a and
    c :: 'a
  assume
    a ∈ A and
    (b, c) ∈ pl-α (a#(filter (λ a. a ∈ A) l))
  thus b ∈ A
    unfolding pl-α-def
    using case-prodD insert-iff mem-Collect-eq set-filter inter-set-filter IntE
    by auto
next
  fix
    b :: 'a and
    c :: 'a
  assume
    b-less-c: (b, c) ∈ pl-α (a#l) and
    b-in-A: b ∈ A and
    c-in-A: c ∈ A
  show (b, c) ∈ pl-α (a#(filter (λ a. a ∈ A) l))
  proof (unfold pl-α-def is-less-preferred-than.simps, safe)
    show b ≲(a#(filter (λ a. a ∈ A) l)) c
    proof (unfold is-less-preferred-than-l.simps, safe)
      show b ∈ set (a#(filter (λ a. a ∈ A) l))
      using b-less-c b-in-A
      unfolding pl-α-def
      by fastforce
    next
      show c ∈ set (a#(filter (λ a. a ∈ A) l))
```

using *b-less-c c-in-A*
unfolding *pl-α-def*
by *fastforce*
next
have $(b,\ c) \in$ *pl-α* $(a\#l)$
by (*simp add*: *b-less-c*)
hence $b \lesssim_{(a\#l)} c$
using *case-prodD mem-Collect-eq*
unfolding *pl-α-def*
by *metis*
moreover have
pl-α (*filter* $(\lambda\ a.\ a \in A)\ l) = \{(a,\ b).\ (a,\ b) \in$ *pl-α* $l \wedge a \in A \wedge b \in A\}$
using *wf-a-l wf-imp-limit*
by *simp*
ultimately show
index $(a\#($*filter* $(\lambda\ a.\ a \in A)\ l)) \ c \leq$ *index* $(a\#($*filter* $(\lambda\ a.\ a \in A)\ l)) \ b$
unfolding *pl-α-def*
using *add-leE add-le-cancel-right case-prodI c-in-A b-in-A index-Cons set-ConsD*
in-rel-Collect-case-prod-eq linorder-le-cases mem-Collect-eq not-one-le-zero
by *fastforce*
qed
qed
next
fix
$b :: 'a$ **and**
$c :: 'a$
assume
a-not-in-A: $a \notin A$ **and**
b-less-c: $(b,\ c) \in$ *pl-α* l
show $(b,\ c) \in$ *pl-α* $(a\#l)$
proof (*unfold pl-α-def is-less-preferred-than-l.simps*, *safe*)
show $b \in$ *set* $(a\#l)$
using *b-less-c*
unfolding *pl-α-def*
by *fastforce*
next
show $c \in$ *set* $(a\#l)$
using *b-less-c*
unfolding *pl-α-def*
by *fastforce*
next
show *index* $(a\#l)\ c \leq$ *index* $(a\#l)\ b$
proof (*unfold index-def*, *simp*, *safe*)
assume $a = b$
thus *False*
using *a-not-in-A b-less-c case-prod-conv is-less-preferred-than-l.elims mem-Collect-eq set-filter wf-a-l*
unfolding *pl-α-def*

```
        by simp
      next
        show find-index (λ x. x = c) l ≤ find-index (λ x. x = b) l
          using b-less-c case-prodD mem-Collect-eq
          unfolding pl-α-def
          by (simp add: index-def)
      qed
    qed
  next
    fix
      b :: 'a and
      c :: 'a
    assume
      a-not-in-l: a ∉ set l and
      a-not-in-A: a ∉ A and
      b-in-A: b ∈ A and
      c-in-A: c ∈ A and
      b-less-c: (b, c) ∈ pl-α (a#l)
    thus (b, c) ∈ pl-α l
    proof (unfold pl-α-def is-less-preferred-than-l.simps, safe)
      assume b ∈ set (a#l)
      thus b ∈ set l
        using a-not-in-A b-in-A
        by fastforce
    next
      assume c ∈ set (a#l)
      thus c ∈ set l
        using a-not-in-A c-in-A
        by fastforce
    next
      assume index (a#l) c ≤ index (a#l) b
      thus index l c ≤ index l b
        using a-not-in-l a-not-in-A c-in-A add-le-cancel-right
              index-Cons index-le-size size-index-conv
        by (metis (no-types, lifting))
    qed
  qed
qed
```

1.5.6 Auxiliary Definitions

```
definition total-on-l :: 'a set ⇒ 'a Preference-List ⇒ bool where
  total-on-l A l ≡ ∀ a ∈ A. a ∈ set l

definition refl-on-l :: 'a set ⇒ 'a Preference-List ⇒ bool where
  refl-on-l A l ≡ (∀ a. a ∈ set l ⟶ a ∈ A) ∧ (∀ a ∈ A. a ≲l a)

definition trans :: 'a Preference-List ⇒ bool where
  trans l ≡ ∀ (a, b, c) ∈ set l × set l × set l. a ≲l b ∧ b ≲l c ⟶ a ≲l c
```

definition *preorder-on-l* :: *'a set* ⇒ *'a Preference-List* ⇒ *bool* **where**
preorder-on-l A l ≡ *refl-on-l A l* ∧ *trans l*

definition *antisym-l* :: *'a list* ⇒ *bool* **where**
antisym-l l ≡ ∀ *a b*. *a* ≲*l* *b* ∧ *b* ≲*l* *a* ⟶ *a* = *b*

definition *partial-order-on-l* :: *'a set* ⇒ *'a Preference-List* ⇒ *bool* **where**
partial-order-on-l A l ≡ *preorder-on-l A l* ∧ *antisym-l l*

definition *linear-order-on-l* :: *'a set* ⇒ *'a Preference-List* ⇒ *bool* **where**
linear-order-on-l A l ≡ *partial-order-on-l A l* ∧ *total-on-l A l*

definition *connex-l* :: *'a set* ⇒ *'a Preference-List* ⇒ *bool* **where**
connex-l A l ≡ *limited A l* ∧ (∀ *a* ∈ *A*. ∀ *b* ∈ *A*. *a* ≲*l* *b* ∨ *b* ≲*l* *a*)

abbreviation *ballot-on* :: *'a set* ⇒ *'a Preference-List* ⇒ *bool* **where**
ballot-on A l ≡ *well-formed-l l* ∧ *linear-order-on-l A l*

1.5.7 Auxiliary Lemmas

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

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

lemma *lin-order-equiv-list-of-alts*:
fixes
A :: *'a set* **and**
l :: *'a Preference-List*
shows *linear-order-on-l A l* = (*A* = *set l*)
unfolding *linear-order-on-l-def total-on-l-def partial-order-on-l-def preorder-on-l-def*
refl-on-l-def
by *auto*

lemma *connex-imp-refl*:
fixes
A :: *'a set* **and**
l :: *'a Preference-List*
assumes *connex-l A l*
shows *refl-on-l A l*
unfolding *refl-on-l-def*

using *assms connex-l-def Preference-List.limited-def*
by *metis*

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

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

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

theorem *above-equiv*:
fixes
l :: *'a Preference-List* **and**
a :: *'a*
shows *set* (*above-l l a*) = *above* (*pl-*α *l*) *a*
proof (*safe*)
fix *b* :: *'a*
assume *b-member*: *b* $\in$ *set* (*above-l l a*)
hence *index l b* $\leq$ *index l a*
unfolding *rank-l.simps above-l-def*
using *Suc-eq-plus1 Suc-le-eq index-take linorder-not-less bot-nat-0.extremum-strict*

by (*metis* (*full-types*))
hence $a \lesssim_l b$
using *Suc-le-mono add-Suc le-antisym take-0 b-member*
in-set-takeD index-take le0 rank-l.simps
unfolding *above-l-def is-less-preferred-than-l.simps*
by *metis*
thus $b \in$ *above* (*pl-α l*) *a*
using *less-preferred-l-rel-equiv pref-imp-in-above*
by *metis*
next
fix $b :: 'a$
assume $b \in$ *above* (*pl-α l*) *a*
hence $a \lesssim_l b$
using *pref-imp-in-above less-preferred-l-rel-equiv*
by *metis*
thus $b \in$ *set* (*above-l l a*)
unfolding *above-l-def is-less-preferred-than-l.simps rank-l.simps*
using *Suc-eq-plus1 Suc-le-eq index-less-size-conv set-take-if-index le-imp-less-Suc*
by (*metis* (*full-types*))
qed

theorem *rank-equiv*:
fixes
l :: $'a$ *Preference-List* **and**
a :: $'a$
assumes *well-formed-l l*
shows *rank-l l a* = *rank* (*pl-α l*) *a*
proof (*simp*, *safe*)
assume $a \in$ *set l*
moreover have *above* (*pl-α l*) *a* = *set* (*above-l l a*)
unfolding *above-equiv*
by *simp*
moreover have *distinct* (*above-l l a*)
unfolding *above-l-def*
using *assms distinct-take*
by *blast*
moreover from *this*
have *card* (*set* (*above-l l a*)) = *length* (*above-l l a*)
using *distinct-card*
by *blast*
moreover have *length* (*above-l l a*) = *rank-l l a*
unfolding *above-l-def*
using *Suc-le-eq*
by (*simp add*: *in-set-member*)
ultimately show *Suc* (*index l a*) = *card* (*above* (*pl-α l*) *a*)
by *simp*
next
assume $a \notin$ *set l*
hence *above* (*pl-α l*) *a* = {}

unfolding *above-def*
using *less-preferred-l-rel-equiv*
by *fastforce*
thus *card* (*above* (*pl-α l*) *a*) = *0*
by *fastforce*
qed

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

1.5.8 First Occurrence Indices

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

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

lemma *pos-in-list-yields-pos*:
fixes
l :: *'a Preference-List* **and**
a :: *'a*
assumes *a* ∈ *set l*

shows $l!(\textit{rank-l}\ l\ a - 1) = a$
using *assms*
proof (*induction l*, *simp*)
fix
$l :: 'a$ *Preference-List* **and**
$b :: 'a$
case (*Cons b l*)
assume $a \in \textit{set}\ (b\#l)$
moreover from *this*
have $\textit{rank-l}\ (b\#l)\ a = 1 + \textit{index}\ (b\#l)\ a$
using *Suc-eq-plus1 add-Suc add-cancel-left-left rank-l.simps*
by *metis*
ultimately show $(b\#l)!(\textit{rank-l}\ (b\#l)\ a - 1) = a$
using *diff-add-inverse nth-index*
by *metis*
qed

lemma *rel-of-pref-pred-for-set-eq-list-to-rel*:
fixes $l :: 'a$ *Preference-List*
shows $\textit{relation-of}\ (\lambda\ y\ z.\ y \lesssim_l z)\ (\textit{set}\ l) = \textit{pl-}\alpha\ l$
proof (*unfold relation-of-def*, *safe*)
fix
$a :: 'a$ **and**
$b :: 'a$
assume $a \lesssim_l b$
moreover have $(a \lesssim_l b) = (a \preceq_{(}\textit{pl-}\alpha\ l)\ b)$
using *less-preferred-l-rel-equiv*
by (*metis* (*no-types*))
ultimately show $(a, b) \in \textit{pl-}\alpha\ l$
by *simp*
next
fix
$a :: 'a$ **and**
$b :: 'a$
assume $(a, b) \in \textit{pl-}\alpha\ l$
thus $a \lesssim_l b$
using *less-preferred-l-rel-equiv*
unfolding *is-less-preferred-than.simps*
by *metis*
thus
$a \in \textit{set}\ l$ **and**
$b \in \textit{set}\ l$
by (*simp*, *simp*)
qed

end

1.6 Preference (List) Profile

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

1.6.1 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 Profile* **where**
  pl-to-pr-α pl = *map* (*Preference-List.pl-α*) *pl*

lemma *prof-abstr-presv-size*:
  fixes *p* :: *'a Profile-List*
  shows *length p* = *length* (*pl-to-pr-α p*)
  by *simp*

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 A* (*pl-to-pr-α p*)
proof (*unfold profile-def*, *intro allI impI*)
  fix *i* :: *nat*
  assume *in-range*: *i* < *length* (*pl-to-pr-α p*)
  moreover have *well-formed-l* (*p*!*i*)
    using *assms in-range*
    unfolding *profile-l-def*
    by *simp*
  moreover have *linear-order-on-l A* (*p*!*i*)
    using *assms in-range*
    unfolding *profile-l-def*
    by *simp*
  ultimately show *linear-order-on A* ((*pl-to-pr-α p*)!*i*)
    using *lin-ord-equiv length-map nth-map*
    unfolding *pl-to-pr-α.simps*

by *metis*
qed

end

1.7 Distance

theory *Distance*
imports *HOL−Library.Extended-Real*
HOL−Combinatorics.List-Permutation
Social-Choice-Types/Profile
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:

- $d(x, y) \geq 0$ (nonnegativity);
- $d(x, y) = 0$ 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 pseudodistance, whereas a quasidistance needs to satisfy the first three conditions (and not necessarily the last one).

1.7.1 Definition

type-synonym *'a Distance* = *'a* ⇒ *'a* ⇒ *ereal*

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*

1.7.2 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 Election set* ⇒ *'a Election Distance* ⇒ *bool*) ⇒
'a Election Distance ⇒ *bool* **where**
election-distance π *d* ≡ π {(*A*, *p*). *finite-profile A p*} *d*

1.7.3 Standard Distance Property

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

1.7.4 Auxiliary Lemmas

lemma *sum-monotone*:
fixes
A :: *'a set* **and**
f :: *'a* ⇒ *int* **and**
g :: *'a* ⇒ *int*
assumes ∀ *a* ∈ *A*. *f a* ≤ *g a*
shows (∑ *a* ∈ *A*. *f a*) ≤ (∑ *a* ∈ *A*. *g a*)
using *assms*
by (*induction A rule*: *infinite-finite-induct*, *simp-all*)

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

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

lemma *uneq-ereal*:
fixes
x :: *int* **and**
y :: *int*
assumes *x* ≤ *y*

shows *ereal* (*real-of-int x*) ≤ *ereal* (*real-of-int y*)
using *assms*
by *simp*

1.7.5 Swap Distance

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

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

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

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

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

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

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

lemma *swap-case-fin*:
fixes
x :: *'a Vote* **and**
y :: *'a Vote*

assumes *alts-$\mathcal{V}$ x = alts-$\mathcal{V}$ y*
shows *swap x y = card (pairwise-disagreements (alts-$\mathcal{V}$ x) (pref-$\mathcal{V}$ x) (pref-$\mathcal{V}$ y))*
using *assms*
by (*induction rule*: *swap.induct*, *simp*)

1.7.6 Spearman Distance

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

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

lemma *spearman-case-fin*:
fixes
x :: *'a Vote* **and**
y :: *'a Vote*
assumes *alts-$\mathcal{V}$ x = alts-$\mathcal{V}$ y*
shows *spearman x y =*
($\sum$ *a ∈ alts-$\mathcal{V}$ x. abs (int (rank (pref-$\mathcal{V}$ x) a) − int (rank (pref-$\mathcal{V}$ y) a)))*
using *assms*
by (*induction rule*: *spearman.induct*, *simp*)

1.8 Properties

definition *distance-anonymity* :: *'a Election Distance ⇒ bool* **where**
distance-anonymity d ≡
∀ A A' π p p'.
(∀ n. (π n) permutes {..< n}) ⟶
d (A, p) (A', p') =
d (A, permute-list (π (length p)) p) (A', permute-list (π (length p')) p')

end

1.9 Votewise Distance

theory *Votewise-Distance*

imports *Social-Choice-Types/Norm*
Distance
begin

Votewise distances are a natural class of distances on elections distances 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 R to n.

1.9.1 Definition

fun *votewise-distance* :: *'a Vote Distance* ⇒ *Norm* ⇒ *'a Election Distance* **where**
votewise-distance d n (*A*, *p*) (*A'*, *p'*) =
(*if length p* = *length p'* ∧ (*0* < *length p* ∨ *A* = *A'*)
then n (*map2* (λ *q q'*. *d* (*A*, *q*) (*A'*, *q'*)) *p p'*)
else ∞)

1.9.2 Inference Rules

lemma *symmetric-norm-imp-distance-anonymous*:
fixes
d :: *'a Vote Distance* **and**
n :: *Norm*
assumes *symmetry n*
shows *distance-anonymity* (*votewise-distance d n*)
proof (*unfold distance-anonymity-def*, *safe*)
fix
A :: *'a set* **and**
A' :: *'a set* **and**
π :: *nat* ⇒ *nat* ⇒ *nat* **and**
p :: *'a Profile* **and**
p' :: *'a Profile*
let *?len* = λ *i*. {..< *length i*} **and**
$?len_\pi$ = λ *i*. π (*length i*) **and**
?prod = *case-prod* (λ *q q'*. *d* (*A*, *q*) (*A'*, *q'*))
let $?\pi_l$ = λ *q*. *permute-list* ($?len_\pi$ *q*) *q*
assume *perm*: ∀ *n*. π *n permutes* {..< *n*}
hence sym_π: ∀ *l*. $?\pi_l$ *l* <~~> *l*
using *mset-permute-list*
by *metis*
show *votewise-distance d n* (*A*, *p*) (*A'*, *p'*) =
votewise-distance d n (*A*, $?\pi_l$ *p*) (*A'*, $?\pi_l$ *p'*)
proof (*cases length p* = *length p'* ∧ (*0* < *length p* ∨ *A* = *A'*))
case *False*
thus *?thesis*
using *perm*
by *auto*
next
case *True*

hence *votewise-distance d n* (*A*, *p*) (*A′*, *p′*) =
n (*map2* (λ *x y*. *d* (*A*, *x*) (*A′*, *y*)) *p p′*)
by *auto*
also have ... = *n* (*?π*$_l$ (*map2* (λ *x y*. *d* (*A*, *x*) (*A′*, *y*)) *p p′*))
using *assms sym*$_\pi$
unfolding *symmetry-def*
by (*metis* (*no-types*, *lifting*))
also have ... = *n* (*map* (*case-prod* (λ *x y*. *d* (*A*, *x*) (*A′*, *y*)))
(*?π*$_l$ (*zip p p′*)))
using *permute-list-map*[*of* ‹*?len*$_\pi$ *p*› ‹*zip p p′*› *?prod*] *perm True*
by *simp*
also have ... = *n* (*map2* (λ *x y*. *d* (*A*, *x*) (*A′*, *y*)) (*?π*$_l$ *p*) (*?π*$_l$ *p′*))
using *permute-list-zip*[*of* ‹*?len*$_\pi$ *p*› ‹*?len p*› *p p′*] *perm True*
by *simp*
also have ... = *votewise-distance d n* (*A*, *?π*$_l$ *p*) (*A′*, *?π*$_l$ *p′*)
using *True*
by *auto*
finally show *?thesis*
by *simp*
qed
qed

end

1.10 Consensus

theory *Consensus*
imports *HOL−Combinatorics.List-Permutation*
Social-Choice-Types/Profile
begin

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

1.10.1 Definition

type-synonym *'a Consensus* = *'a Election* ⇒ *bool*

1.10.2 Consensus Conditions

Nonempty set.

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

Nonempty profile.

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

Equal top ranked alternatives.

fun *equal-top*$_{\mathcal{C}}'$ :: *'a* $\Rightarrow$ *'a Consensus* **where**
equal-top$_{\mathcal{C}}'$ a $(A, p) = (a \in A \wedge (\forall\ i < length\ p.\ above\ (p!i)\ a = \{a\}))$

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

Equal votes.

fun *equal-vote*$_{\mathcal{C}}'$ :: *'a Preference-Relation* $\Rightarrow$ *'a Consensus* **where**
equal-vote$_{\mathcal{C}}'$ r $(A, p) = (\forall\ i < length\ p.\ (p!i) = r)$

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

Unanimity condition.

fun *unanimity*$_{\mathcal{C}}$ :: *'a Consensus* **where**
unanimity$_{\mathcal{C}}$ $c = (nonempty\text{-}set_{\mathcal{C}}\ c \wedge nonempty\text{-}profile_{\mathcal{C}}\ c \wedge equal\text{-}top_{\mathcal{C}}\ c)$

Strong unanimity condition.

fun *strong-unanimity*$_{\mathcal{C}}$ :: *'a Consensus* **where**
strong-unanimity$_{\mathcal{C}}$ $c = (nonempty\text{-}set_{\mathcal{C}}\ c \wedge nonempty\text{-}profile_{\mathcal{C}}\ c \wedge equal\text{-}vote_{\mathcal{C}}\ c)$

1.10.3 Properties

definition *consensus-anonymity* :: *'a Consensus* $\Rightarrow$ *bool* **where**
consensus-anonymity $c \equiv$
$\forall\ A\ p\ q.\ profile\ A\ p \wedge profile\ A\ q \wedge p\ <\sim\sim>\ q \longrightarrow c\ (A, p) \longrightarrow c\ (A, q)$

1.10.4 Auxiliary Lemmas

lemma *ex-anon-cons-imp-cons-anonymous*:
fixes
b :: *'a Consensus* **and**
b':: *'b* $\Rightarrow$ *'a Consensus*
assumes
general-cond-b: $b = (\lambda\ E.\ \exists\ x.\ b'\ x\ E)$ **and**
all-cond-anon: $\forall\ x.\ consensus\text{-}anonymity\ (b'\ x)$
shows *consensus-anonymity b*
proof (*unfold consensus-anonymity-def*, *safe*)
fix
A :: *'a set* **and**
p :: *'a Profile* **and**
q :: *'a Profile*
assume
cond-b: $b\ (A, p)$ **and**
prof-p: *profile A p* **and**

prof-q: *profile* A q **and**
perm: $p <\sim\sim> q$
have $\exists$ x. b' x (A, p)
using *cond-b general-cond-b*
by *simp*
then obtain $x :: 'b$ **where**
b' x (A, p)
by *blast*
hence b' x (A, q)
using *all-cond-anon perm prof-p prof-q*
unfolding *consensus-anonymity-def*
by *blast*
hence $\exists$ x. b' x (A, q)
by *metis*
thus b (A, q)
using *general-cond-b*
by *simp*
qed

1.10.5 Theorems

lemma *nonempty-set-cons-anonymous*: *consensus-anonymity nonempty-set*$_C$
unfolding *consensus-anonymity-def*
by *simp*

lemma *nonempty-profile-cons-anonymous*: *consensus-anonymity nonempty-profile*$_C$
proof (*unfold consensus-anonymity-def*, *clarify*)
fix
$A :: 'a$ *set* **and**
$p :: 'a$ *Profile* **and**
$q :: 'a$ *Profile*
assume
perm: $p <\sim\sim> q$ **and**
not-empty-p: *nonempty-profile*$_C$ (A, p)
have *length* q = *length* p
using *perm perm-length*
by *force*
thus *nonempty-profile*$_C$ (A, q)
using *not-empty-p length-0-conv*
unfolding *nonempty-profile*$_C$*.simps*
by *metis*
qed

lemma *equal-top-cons'-anonymous*:
fixes $a :: 'a$
shows *consensus-anonymity* (*equal-top*$_C$$'$ a)
proof (*unfold consensus-anonymity-def*, *clarify*)
fix
$A :: 'a$ *set* **and**

```
    p :: 'a Profile and
    q :: 'a Profile
  assume
    perm: p <~~> q and
    top-cons-a: equal-top_C' a (A, p)
  from perm obtain π where
    perm_π: π permutes {..< length p} and
    list-pq_π: permute-list π p = q
    using mset-eq-permutation
    by metis
  have l: length p = length q
    using perm perm-length
    by force
  hence ∀ i < length q. π i < length p
    using perm_π permutes-in-image
    by fastforce
  moreover have ∀ i < length q. q!i = p!(π i)
    using list-pq_π
    unfolding permute-list-def
    by auto
  moreover have winner: ∀ i < length p. above (p!i) a = {a}
    using top-cons-a
    by simp
  ultimately have ∀ i < length p. above (q!i) a = {a}
    using l
    by metis
  moreover have a ∈ A
    using top-cons-a
    by simp
  ultimately show equal-top_C' a (A, q)
    using l
    unfolding equal-top_C'.simps
    by metis
qed

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

lemma eq-vote-cons'-anonymous:
  fixes r :: 'a Preference-Relation
  shows consensus-anonymity (equal-vote_C' r)
proof (unfold consensus-anonymity-def, clarify)
  fix
    A :: 'a set and
    p :: 'a Profile and
    q :: 'a Profile
  assume
```

```
    perm: p <~~> q and
    equal-votes-pref: equal-vote_C' r (A, p)
  from perm obtain π where
    perm_π: π permutes {..< length p} and
    list-pq_π: permute-list π p = q
    using mset-eq-permutation
    by metis
  have l: length p = length q
    using perm perm-length
    by force
  hence ∀ i < length q. π i < length p
    using perm_π permutes-in-image
    by fastforce
  moreover have ∀ i < length q. q!i = p!(π i)
    using list-pq_π
    unfolding permute-list-def
    by auto
  moreover have winner: ∀ i < length p. p!i = r
    using equal-votes-pref
    by simp
  ultimately have ∀ i < length p. q!i = r
    using l
    by metis
  thus equal-vote_C' r (A, q)
    using l
    unfolding equal-vote_C'.simps
    by metis
qed

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

end
```

Chapter 2

Component Types

2.1 Electoral Module

theory *Electoral-Module*
  imports *Social-Choice-Types/Profile*
    Social-Choice-Types/Result
    HOL−Combinatorics.List-Permutation
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.

2.1.1 Definition

An electoral module maps a set of alternatives and a profile to a result.

type-synonym *'a Electoral-Module = 'a set ⇒ 'a Profile ⇒ 'a Result*

2.1.2 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 alterna-

tives 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.

definition *electoral-module* :: *'a Electoral-Module* ⇒ *bool* **where**
electoral-module m ≡ ∀ *A p*. *profile A p* ⟶ *well-formed A* (*m A p*)

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 Electoral-Module* ⇒ *'a set* ⇒ *'a Profile* ⇒ *'a set* **where**
elect m A p ≡ *elect-r* (*m A p*)

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

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

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

definition *defers* :: *nat* ⇒ *'a Electoral-Module* ⇒ *bool* **where**
defers n m ≡
electoral-module m ∧
(∀ *A p*. *card A* ≥ *n* ∧ *finite-profile A p* ⟶ *card* (*defer m 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 Electoral-Module* ⇒ *bool* **where**
rejects n m ≡
electoral-module m ∧
(∀ *A p*. *card A* ≥ *n* ∧ *finite-profile A p* ⟶ *card* (*reject m A p*) = *n*)

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

definition *eliminates* :: *nat* ⇒ *'a Electoral-Module* ⇒ *bool* **where**
eliminates n m ≡
electoral-module m ∧
(∀ *A p*. *card A* > *n* ∧ *profile A p* ⟶ *card* (*reject m 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 Electoral-Module* ⇒ *bool* **where**
elects n m ≡
electoral-module m ∧
(∀ *A p*. *card A* ≥ *n* ∧ *profile A p* ⟶ *card* (*elect m 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 Electoral-Module* ⇒ *'a set* ⇒ *'a* ⇒ *bool* **where**
indep-of-alt m A a ≡
electoral-module m ∧ (∀ *p q*. *equiv-prof-except-a A p q a* ⟶ *m A p* = *m A q*)

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

2.1.3 Equivalence Definitions

definition *prof-contains-result* :: *'a Electoral-Module* ⇒ *'a set* ⇒ *'a Profile* ⇒
'a Profile ⇒ *'a* ⇒ *bool* **where**
prof-contains-result m A p q a ≡
electoral-module m ∧ *profile A p* ∧ *profile A q* ∧ *a* ∈ *A* ∧
(*a* ∈ *elect m A p* ⟶ *a* ∈ *elect m A q*) ∧
(*a* ∈ *reject m A p* ⟶ *a* ∈ *reject m A q*) ∧
(*a* ∈ *defer m A p* ⟶ *a* ∈ *defer m A q*)

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

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

definition *mod-contains-result* :: *'a Electoral-Module* ⇒ *'a Electoral-Module* ⇒
'a set ⇒ *'a Profile* ⇒ *'a* ⇒ *bool* **where**
mod-contains-result m n A p a ≡
electoral-module m ∧ *electoral-module n* ∧ *profile A p* ∧ *a* ∈ *A* ∧
(*a* ∈ *elect m A p* ⟶ *a* ∈ *elect n A p*) ∧
(*a* ∈ *reject m A p* ⟶ *a* ∈ *reject n A p*) ∧
(*a* ∈ *defer m A p* ⟶ *a* ∈ *defer n A p*)

definition *mod-contains-result-sym* :: *'a Electoral-Module* ⇒
'a Electoral-Module ⇒ *'a set* ⇒ *'a Profile* ⇒ *'a* ⇒ *bool* **where**
mod-contains-result-sym m n A p a ≡
electoral-module m ∧ *electoral-module n* ∧ *profile A p* ∧ *a* ∈ *A* ∧
(*a* ∈ *elect m A p* ⟷ *a* ∈ *elect n A p*) ∧
(*a* ∈ *reject m A p* ⟷ *a* ∈ *reject n A p*) ∧
(*a* ∈ *defer m A p* ⟷ *a* ∈ *defer n A p*)

2.1.4 Auxiliary Lemmas

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

lemma *par-comp-result-sound*:
fixes
$m :: 'a$ *Electoral-Module* **and**
$A :: 'a$ *set* **and**
$p :: 'a$ *Profile*
assumes
electoral-module m **and**
profile A p
shows *well-formed* A $(m\ A\ p)$
using *assms*
unfolding *electoral-module-def*
by *simp*

lemma *result-presv-alts*:
fixes
$m :: 'a$ *Electoral-Module* **and**
$A :: 'a$ *set* **and**
$p :: 'a$ *Profile*
assumes
electoral-module m **and**
profile A p
shows $(elect\ m\ A\ p) \cup (reject\ m\ A\ p) \cup (defer\ m\ A\ p) = A$
proof (*safe*)
fix $a :: 'a$
assume $a \in elect\ m\ A\ p$
moreover have
$\forall\ p'.\ set\text{-}equals\text{-}partition\ A\ p' \longrightarrow$
$(\exists\ E\ R\ D.\ p' = (E, R, D) \wedge E \cup R \cup D = A)$
by *simp*
moreover have *set-equals-partition* A $(m\ A\ p)$
using *assms*
unfolding *electoral-module-def*

by *simp*
ultimately show $a \in A$
using *UnI1 fstI*
by (*metis* (*no-types*))
next
fix $a :: 'a$
assume $a \in$ *reject* $m\ A\ p$
moreover have
$\forall\ p'.$ *set-equals-partition* $A\ p' \longrightarrow$
$(\exists\ E\ R\ D.\ p' = (E,\ R,\ D) \wedge E \cup R \cup D = A)$
by *simp*
moreover have *set-equals-partition* A $(m\ A\ p)$
using *assms*
unfolding *electoral-module-def*
by *simp*
ultimately show $a \in A$
using *UnI1 fstI sndI subsetD sup-ge2*
by *metis*
next
fix $a :: 'a$
assume $a \in$ *defer* $m\ A\ p$
moreover have
$\forall\ p'.$ *set-equals-partition* $A\ p' \longrightarrow$
$(\exists\ E\ R\ D.\ p' = (E,\ R,\ D) \wedge E \cup R \cup D = A)$
by *simp*
moreover have *set-equals-partition* A $(m\ A\ p)$
using *assms*
unfolding *electoral-module-def*
by *simp*
ultimately show $a \in A$
using *sndI subsetD sup-ge2*
by *metis*
next
fix $a :: 'a$
assume
$a \in A$ **and**
$a \notin$ *defer* $m\ A\ p$ **and**
$a \notin$ *reject* $m\ A\ p$
moreover have
$\forall\ p'.$ *set-equals-partition* $A\ p' \longrightarrow$
$(\exists\ E\ R\ D.\ p' = (E,\ R,\ D) \wedge E \cup R \cup D = A)$
by *simp*
moreover have *set-equals-partition* A $(m\ A\ p)$
using *assms*
unfolding *electoral-module-def*
by *simp*
ultimately show $a \in$ *elect* $m\ A\ p$
using *prod.sel Un-iff*
by *metis*

qed

lemma *result-disj*:
fixes
m :: *'a Electoral-Module* **and**
A :: *'a set* **and**
p :: *'a Profile*
assumes
electoral-module m **and**
profile A p
shows
(*elect m A p*) ∩ (*reject m A p*) = {} ∧
(*elect m A p*) ∩ (*defer m A p*) = {} ∧
(*reject m A p*) ∩ (*defer m A p*) = {}
proof (*safe*, *simp-all*)
fix *a* :: *'a*
assume
a ∈ *elect m A p* **and**
a ∈ *reject m A p*
moreover have *well-formed A* (*m A p*)
using *assms*
unfolding *electoral-module-def*
by *metis*
ultimately show *False*
using *prod.exhaust-sel DiffE UnCI result-imp-rej*
by (*metis* (*no-types*))
next
fix *a* :: *'a*
assume
elect-a: *a* ∈ *elect m A p* **and**
defer-a: *a* ∈ *defer m A p*
have *disj*:
∀ *p'*. *disjoint3 p'* ⟶
(∃ *B C D*. *p'* = (*B*, *C*, *D*) ∧ *B* ∩ *C* = {} ∧ *B* ∩ *D* = {} ∧ *C* ∩ *D* = {})
by *simp*
have *well-formed A* (*m A p*)
using *assms*
unfolding *electoral-module-def*
by *metis*
hence *disjoint3* (*m A p*)
by *simp*
then obtain
e :: *'a Result* ⇒ *'a set* **and**
r :: *'a Result* ⇒ *'a set* **and**
d :: *'a Result* ⇒ *'a set*
where
m A p =
(*e* (*m A p*), *r* (*m A p*), *d* (*m A p*)) ∧
e (*m A p*) ∩ *r* (*m A p*) = {} ∧

```
        e (m A p) ∩ d (m A p) = {} ∧
        r (m A p) ∩ d (m A p) = {}
    using elect-a defer-a disj
    by metis
  hence ((elect m A p) ∩ (reject m A p) = {}) ∧
        ((elect m A p) ∩ (defer m A p) = {}) ∧
        ((reject m A p) ∩ (defer m A p) = {})
    using eq-snd-iff fstI
    by metis
  thus False
    using elect-a defer-a disjoint-iff-not-equal
    by (metis (no-types))
next
  fix a :: 'a
  assume
    a ∈ reject m A p and
    a ∈ defer m A p
  moreover have well-formed A (m A p)
    using assms
    unfolding electoral-module-def
    by simp
  ultimately show False
    using prod.exhaust-sel DiffE UnCI result-imp-rej
    by (metis (no-types))
qed

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

lemma reject-in-alts:
  fixes
    m :: 'a Electoral-Module and
    A :: 'a set and
    p :: 'a Profile
  assumes
    electoral-module m and
    profile A p
  shows reject m A p ⊆ A
  using le-supI1 assms result-presv-alts sup-ge2
  by fastforce
```

lemma *defer-in-alts*:
fixes
m :: *'a Electoral-Module* **and**
A :: *'a set* **and**
p :: *'a Profile*
assumes
electoral-module m **and**
profile A p
shows *defer m A p* ⊆ *A*
using *assms result-presv-alts*
by *auto*

lemma *def-presv-prof*:
fixes
m :: *'a Electoral-Module* **and**
A :: *'a set* **and**
p :: *'a Profile*
assumes
electoral-module m **and**
profile A p
shows *profile* (*defer m A p*) (*limit-profile* (*defer m A p*) *p*)
using *defer-in-alts limit-profile-sound assms*
by *metis*

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

lemma *upper-card-bounds-for-result*:
fixes
m :: *'a Electoral-Module* **and**
A :: *'a set* **and**
p :: *'a Profile*
assumes
electoral-module m **and**
finite-profile A p
shows
upper-card-bound-for-elect: *card* (*elect m A p*) ≤ *card A* **and**
upper-card-bound-for-reject: *card* (*reject m A p*) ≤ *card A* **and**
upper-card-bound-for-defer: *card* (*defer m A p*) ≤ *card A*
proof −
show *card* (*elect m A p*) ≤ *card A*
by (*simp add*: *assms card-mono elect-in-alts*)
next
show *card* (*reject m A p*) ≤ *card A*
by (*simp add*: *assms card-mono reject-in-alts*)
next
show *card* (*defer m A p*) ≤ *card A*
by (*simp add*: *assms card-mono defer-in-alts*)
qed

lemma *reject-not-elec-or-def*:
fixes
m :: *'a Electoral-Module* **and**
A :: *'a set* **and**
p :: *'a Profile*
assumes
electoral-module m **and**
profile A p
shows *reject m A p* = *A* − (*elect m A p*) − (*defer m A p*)
proof −
have *well-formed A* (*m A p*)
using *assms*
unfolding *electoral-module-def*
by *simp*
hence (*elect m A p*) ∪ (*reject m A p*) ∪ (*defer m A p*) = *A*
using *assms result-presv-alts*
by *simp*
moreover have
(*elect m A p*) ∩ (*reject m A p*) = {} ∧ (*reject m A p*) ∩ (*defer m A p*) = {}
using *assms result-disj*
by *blast*
ultimately show *?thesis*
by *blast*
qed

lemma *elec-and-def-not-rej*:
fixes
m :: *'a Electoral-Module* **and**
A :: *'a set* **and**
p :: *'a Profile*
assumes
electoral-module m **and**
profile A p
shows *elect m A p* ∪ *defer m A p* = *A* − (*reject m A p*)
proof −
have (*elect m A p*) ∪ (*reject m A p*) ∪ (*defer m A p*) = *A*
using *assms result-presv-alts*
by *blast*
moreover have
(*elect m A p*) ∩ (*reject m A p*) = {} ∧ (*reject m A p*) ∩ (*defer m A p*) = {}
using *assms result-disj*
by *blast*
ultimately show *?thesis*
by *blast*
qed

lemma *defer-not-elec-or-rej*:
fixes

$m :: 'a$ *Electoral-Module* **and**
$A :: 'a$ *set* **and**
$p :: 'a$ *Profile*
assumes
electoral-module m **and**
profile A p
shows *defer* m A $p = A - (elect\ m\ A\ p) - (reject\ m\ A\ p)$
proof −
have *well-formed* A $(m\ A\ p)$
using *assms*
unfolding *electoral-module-def*
by *simp*
hence $(elect\ m\ A\ p) \cup (reject\ m\ A\ p) \cup (defer\ m\ A\ p) = A$
using *assms result-presv-alts*
by *simp*
moreover have
$(elect\ m\ A\ p) \cap (defer\ m\ A\ p) = \{\} \wedge (reject\ m\ A\ p) \cap (defer\ m\ A\ p) = \{\}$
using *assms result-disj*
by *blast*
ultimately show *?thesis*
by *blast*
qed

lemma *electoral-mod-defer-elem*:
fixes
$m :: 'a$ *Electoral-Module* **and**
$A :: 'a$ *set* **and**
$p :: 'a$ *Profile* **and**
$a :: 'a$
assumes
electoral-module m **and**
profile A p **and**
$a \in A$ **and**
$a \notin elect\ m\ A\ p$ **and**
$a \notin reject\ m\ A\ p$
shows $a \in defer\ m\ A\ p$
using *DiffI assms reject-not-elec-or-def*
by *metis*

lemma *mod-contains-result-comm*:
fixes
$m :: 'a$ *Electoral-Module* **and**
$n :: 'a$ *Electoral-Module* **and**
$A :: 'a$ *set* **and**
$p :: 'a$ *Profile* **and**
$a :: 'a$
assumes *mod-contains-result* m n A p a
shows *mod-contains-result* n m A p a
proof (*unfold mod-contains-result-def*, *safe*)

from *assms*
show *electoral-module n*
unfolding *mod-contains-result-def*
by *safe*
next
from *assms*
show *electoral-module m*
unfolding *mod-contains-result-def*
by *safe*
next
from *assms*
show *profile A p*
unfolding *mod-contains-result-def*
by *safe*
next
from *assms*
show $a \in A$
unfolding *mod-contains-result-def*
by *safe*
next
assume $a \in$ *elect n A p*
thus $a \in$ *elect m A p*
using *IntI assms electoral-mod-defer-elem empty-iff mod-contains-result-def result-disj*
by (*metis* (*mono-tags*, *lifting*))
next
assume $a \in$ *reject n A p*
thus $a \in$ *reject m A p*
using *IntI assms electoral-mod-defer-elem empty-iff mod-contains-result-def result-disj*
by (*metis* (*mono-tags*, *lifting*))
next
assume $a \in$ *defer n A p*
thus $a \in$ *defer m A p*
using *IntI assms electoral-mod-defer-elem empty-iff mod-contains-result-def result-disj*
by (*metis* (*mono-tags*, *lifting*))
qed

lemma *not-rej-imp-elec-or-def*:
fixes
$m :: 'a$ *Electoral-Module* **and**
$A :: 'a$ *set* **and**
$p :: 'a$ *Profile* **and**
$a :: 'a$
assumes
electoral-module m **and**
profile A p **and**
$a \in A$ **and**

```
    a ∉ reject m A p
  shows a ∈ elect m A p ∨ a ∈ defer m A p
  using assms electoral-mod-defer-elem
  by metis

lemma single-elim-imp-red-def-set:
  fixes
    m :: 'a Electoral-Module and
    A :: 'a set and
    p :: 'a Profile
  assumes
    eliminates 1 m and
    card A > 1 and
    profile A p
  shows defer m A p ⊂ A
  using Diff-eq-empty-iff Diff-subset card-eq-0-iff defer-in-alts eliminates-def
      eq-iff not-one-le-zero psubsetI reject-not-elec-or-def assms
  by metis

lemma eq-alts-in-profs-imp-eq-results:
  fixes
    m :: 'a Electoral-Module and
    A :: 'a set and
    p :: 'a Profile and
    q :: 'a Profile
  assumes
    eq: ∀ a ∈ A. prof-contains-result m A p q a and
    mod-m: electoral-module m and
    prof-p: profile A p and
    prof-q: profile A q
  shows m A p = m A q
proof −
  have elected-in-A: elect m A q ⊆ A
    using elect-in-alts mod-m prof-q
    by metis
  have rejected-in-A: reject m A q ⊆ A
    using reject-in-alts mod-m prof-q
    by metis
  have deferred-in-A: defer m A q ⊆ A
    using defer-in-alts mod-m prof-q
    by metis
  have ∀ a ∈ elect m A p. a ∈ elect m A q
    using elect-in-alts eq prof-contains-result-def mod-m prof-p in-mono
    by metis
  moreover have ∀ a ∈ elect m A q. a ∈ elect m A p
  proof
    fix a :: 'a
    assume q-elect-a: a ∈ elect m A q
    hence a ∈ A
```

```
    using elected-in-A
    by blast
  moreover have a ∉ defer m A q
    using q-elect-a prof-q mod-m result-disj
    by blast
  moreover have a ∉ reject m A q
    using q-elect-a disjoint-iff-not-equal prof-q mod-m result-disj
    by metis
  ultimately show a ∈ elect m A p
    using electoral-mod-defer-elem eq prof-contains-result-def
    by metis
qed
moreover have ∀ a ∈ reject m A p. a ∈ reject m A q
  using reject-in-alts eq prof-contains-result-def mod-m prof-p
  by fastforce
moreover have ∀ a ∈ reject m A q. a ∈ reject m A p
proof
  fix a :: 'a
  assume q-rejects-a: a ∈ reject m A q
  hence a ∈ A
    using rejected-in-A
    by blast
  moreover have a-not-deferred-q: a ∉ defer m A q
    using q-rejects-a prof-q mod-m result-disj
    by blast
  moreover have a-not-elected-q: a ∉ elect m A q
    using q-rejects-a disjoint-iff-not-equal prof-q mod-m result-disj
    by metis
  ultimately show a ∈ reject m A p
    using electoral-mod-defer-elem eq prof-contains-result-def
    by metis
qed
moreover have ∀ a ∈ defer m A p. a ∈ defer m A q
  using defer-in-alts eq prof-contains-result-def mod-m prof-p
  by fastforce
moreover have ∀ a ∈ defer m A q. a ∈ defer m A p
proof
  fix a :: 'a
  assume q-defers-a: a ∈ defer m A q
  moreover have a ∈ A
    using q-defers-a deferred-in-A
    by blast
  moreover have a ∉ elect m A q
    using q-defers-a prof-q mod-m result-disj
    by blast
  moreover have a ∉ reject m A q
    using q-defers-a prof-q disjoint-iff-not-equal mod-m result-disj
    by metis
  ultimately show a ∈ defer m A p
```

```
    using electoral-mod-defer-elem eq prof-contains-result-def
    by metis
  qed
  ultimately show ?thesis
    using prod.collapse subsetI subset-antisym
    by (metis (no-types))
qed

lemma eq-def-and-elect-imp-eq:
  fixes
    m :: 'a Electoral-Module and
    n :: 'a Electoral-Module and
    A :: 'a set and
    p :: 'a Profile and
    q :: 'a Profile
  assumes
    mod-m: electoral-module m and
    mod-n: electoral-module n and
    prof-p: profile A p and
    prof-q: profile A q and
    elec-eq: elect m A p = elect n A q and
    def-eq: defer m A p = defer n A q
  shows m A p = n A q
proof −
  have reject m A p = A − ((elect m A p) ∪ (defer m A p))
    using mod-m prof-p elect-rej-def-combination result-imp-rej
    unfolding electoral-module-def
    by metis
  moreover have reject n A q = A − ((elect n A q) ∪ (defer n A q))
    using mod-n prof-q elect-rej-def-combination result-imp-rej
    unfolding electoral-module-def
    by metis
  ultimately show ?thesis
    using elec-eq def-eq prod-eqI
    by metis
qed
```

2.1.5 Non-Blocking

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

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

2.1.6 Electing

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

definition *electing* :: *'a Electoral-Module* ⇒ *bool* **where**
electing m ≡
electoral-module m ∧
(∀ *A p*. *A* ≠ {} ∧ *finite-profile A p* ⟶ *elect m A p* ≠ {})

lemma *electing-for-only-alt*:
fixes
m :: *'a Electoral-Module* **and**
A :: *'a set* **and**
p :: *'a Profile*
assumes
one-alt: *card A* = *1* **and**
electing: *electing m* **and**
prof-p: *profile A p*
shows *elect m A p* = *A*
proof (*safe*)
fix *a* :: *'a*
assume *elect-a*: *a* ∈ *elect m A p*
have *electoral-module m* ⟶ *elect m A p* ⊆ *A*
using *prof-p*
by (*simp add*: *elect-in-alts*)
hence *elect m A p* ⊆ *A*
using *electing*
unfolding *electing-def*
by *metis*
thus *a* ∈ *A*
using *elect-a*
by *blast*
next
fix *a* :: *'a*
assume *a* ∈ *A*
thus *a* ∈ *elect m A p*
using *electing prof-p one-alt One-nat-def Suc-leI card-seteq card-gt-0-iff*
elect-in-alts infinite-super lessI
unfolding *electing-def*
by *metis*
qed

theorem *electing-imp-non-blocking*:
fixes *m* :: *'a Electoral-Module*
assumes *electing m*
shows *non-blocking m*
proof (*unfold non-blocking-def*, *safe*)
from *assms*
show *electoral-module m*
unfolding *electing-def*
by *simp*
next
fix

```
    A :: 'a set and
    p :: 'a Profile and
    a :: 'a
  assume
    finite A and
    profile A p and
    reject m A p = A and
    a ∈ A
  moreover have
    electoral-module m ∧
      (∀ A q. A ≠ {} ∧ finite-profile A q ⟶ elect m A q ≠ {})
    using assms
    unfolding electing-def
    by metis
  ultimately show a ∈ {}
    using Diff-cancel Un-empty elec-and-def-not-rej
    by (metis (no-types))
qed
```

2.1.7 Properties

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

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

lemma single-rej-decr-def-card:
  fixes
    m :: 'a Electoral-Module and
    A :: 'a set and
    p :: 'a Profile
  assumes
    rejecting: rejects 1 m and
    non-electing: non-electing m and
    f-prof: finite-profile A p
  shows card (defer m A p) = card A − 1
proof −
  have no-elect:
    electoral-module m ∧ (∀ A q. profile A q ⟶ elect m A q = {})
    using non-electing
    unfolding non-electing-def
    by (metis (no-types))
  hence reject m A p ⊆ A
    using f-prof reject-in-alts
    by metis
  moreover have A = A − elect m A p
    using no-elect f-prof
    by blast
  ultimately show ?thesis
```

```
    using f-prof no-elect rejecting card-Diff-subset card-gt-0-iff
        defer-not-elec-or-rej less-one order-less-imp-le Suc-leI
        bot.extremum-unique card.empty diff-is-0-eq' One-nat-def
    unfolding rejects-def
    by metis
qed

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

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

```
definition defer-deciding :: 'a Electoral-Module ⇒ bool where
  defer-deciding m ≡
    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 Electoral-Module ⇒ bool where
  decrementing m ≡
    electoral-module m ∧
```

$(\forall\ A\ p.\ \mathit{profile}\ A\ p \wedge \mathit{card}\ A > 1 \longrightarrow \mathit{card}\ (\mathit{reject}\ m\ A\ p) \geq 1)$

definition *defer-condorcet-consistency* :: *'a Electoral-Module* ⇒ *bool* **where**
defer-condorcet-consistency m ≡
electoral-module m ∧
(∀ *A p a. condorcet-winner A p a* ⟶
(*m A p* = ({}, *A* − (*defer m A p*), {*d* ∈ *A. condorcet-winner A p d*})))

definition *condorcet-compatibility* :: *'a Electoral-Module* ⇒ *bool* **where**
condorcet-compatibility m ≡
electoral-module m ∧
(∀ *A p a. condorcet-winner A p a* ⟶
a ∉ *reject m A p* ∧
(∀ *b.* ¬ *condorcet-winner A p b* ⟶ *b* ∉ *elect m A p*) ∧
(*a* ∈ *elect m A p* ⟶
(∀ *b* ∈ *A.* ¬ *condorcet-winner A p b* ⟶ *b* ∈ *reject m A p*)))

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

definition *defer-monotonicity* :: *'a Electoral-Module* ⇒ *bool* **where**
defer-monotonicity m ≡
electoral-module m ∧
(∀ *A p q a. a* ∈ *defer m A p* ∧ *lifted A p q a* ⟶ *a* ∈ *defer m A q*)

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

definition *defer-lift-invariance* :: *'a Electoral-Module* ⇒ *bool* **where**
defer-lift-invariance m ≡
electoral-module m ∧
(∀ *A p q a. a* ∈ (*defer m A p*) ∧ *lifted A p q a* ⟶ *m A p* = *m A q*)

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 Electoral-Module* ⇒
'a Electoral-Module ⇒ *bool* **where**
disjoint-compatibility m n ≡
electoral-module m ∧ *electoral-module n* ∧
(∀ *A.*
(∃ *B* ⊆ *A.*
(∀ *a* ∈ *B. indep-of-alt m A a* ∧
(∀ *p. profile A p* ⟶ *a* ∈ *reject m A p*)) ∧
(∀ *a* ∈ *A* − *B. indep-of-alt n A a* ∧
(∀ *p. profile A p* ⟶ *a* ∈ *reject n 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 Electoral-Module* ⇒ *bool* **where**

invariant-monotonicity m ≡
electoral-module m ∧
(∀ *A p q a*. *a* ∈ *elect m A p* ∧ *lifted A p q a* ⟶
elect m A q = *elect m A p* ∨ *elect m 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 Electoral-Module* ⇒ *bool* **where**
defer-invariant-monotonicity m ≡
electoral-module m ∧ *non-electing m* ∧
(∀ *A p q a*. *a* ∈ *defer m A p* ∧ *lifted A p q a* ⟶
defer m A q = *defer m A p* ∨ *defer m A q* = {*a*})

2.1.8 Inference Rules

lemma *ccomp-and-dd-imp-def-only-winner*:
fixes
m :: *'a Electoral-Module* **and**
A :: *'a set* **and**
p :: *'a Profile* **and**
a :: *'a*
assumes
ccomp: *condorcet-compatibility m* **and**
dd: *defer-deciding m* **and**
winner: *condorcet-winner A p a*
shows *defer m A p* = {*a*}
proof (*rule ccontr*)
assume *not-w*: *defer m A p* ≠ {*a*}
have *def-one*: *defers 1 m*
using *dd*
unfolding *defer-deciding-def*
by *metis*
hence *c-win*: *finite-profile A p* ∧ *a* ∈ *A* ∧ (∀ *b* ∈ *A* − {*a*}. *wins a p b*)
using *winner*
by *simp*
hence *card* (*defer m A p*) = *1*
using *Suc-leI card-gt-0-iff def-one equals0D*
unfolding *One-nat-def defers-def*
by *metis*
hence ∃ *b* ∈ *A*. *defer m A p* = {*b*}
using *card-1-singletonE dd defer-in-alts insert-subset c-win*
unfolding *defer-deciding-def*
by *metis*
hence ∃ *b* ∈ *A*. *b* ≠ *a* ∧ *defer m A p* = {*b*}
using *not-w*
by *metis*
hence *not-in-defer*: *a* ∉ *defer m A p*
by *auto*

```
  have non-electing m
    using dd
    unfolding defer-deciding-def
    by simp
  hence a ∉ elect m A p
    using c-win equals0D
    unfolding non-electing-def
    by simp
  hence a ∈ reject m A p
    using not-in-defer ccomp c-win electoral-mod-defer-elem
    unfolding condorcet-compatibility-def
    by metis
  moreover have a ∉ reject m A p
    using ccomp c-win winner
    unfolding condorcet-compatibility-def
    by simp
  ultimately show False
    by simp
qed

theorem ccomp-and-dd-imp-dcc[simp]:
  fixes m :: 'a Electoral-Module
  assumes
    ccomp: condorcet-compatibility m and
    dd: defer-deciding m
  shows defer-condorcet-consistency m
proof (unfold defer-condorcet-consistency-def, auto)
  show electoral-module m
    using dd
    unfolding defer-deciding-def
    by metis
next
  fix
    A :: 'a set and
    p :: 'a Profile and
    a :: 'a
  assume
    fin-A: finite A and
    prof-A: profile A p and
    a-in-A: a ∈ A and
    c-winner: ∀ b ∈ A − {a}.
                card {i. i < length p ∧ (a, b) ∈ (p!i)} <
                  card {i. i < length p ∧ (b, a) ∈ (p!i)}
  hence winner: condorcet-winner A p a
    by simp
  hence elect-empty: elect m A p = {}
    using dd
    unfolding defer-deciding-def non-electing-def
    by simp
```

```
  have cond-winner-a: {a} = {c ∈ A. condorcet-winner A p c}
    using cond-winner-unique winner
    by metis
  have defer-a: defer m A p = {a}
    using winner dd ccomp ccomp-and-dd-imp-def-only-winner winner
    by simp
  hence reject m A p = A − defer m A p
    using Diff-empty dd reject-not-elec-or-def winner elect-empty
    unfolding defer-deciding-def
    by fastforce
  hence m A p = ({}, A − defer m A p, {a})
    using elect-empty defer-a elect-rej-def-combination
    by metis
  hence m A p = ({}, A − defer m A p, {c ∈ A. condorcet-winner A p c})
    using cond-winner-a
    by simp
  thus m A p =
        ({},
          A − defer m A p,
          {c ∈ A. ∀ b ∈ A − {c}.
            card {i. i < length p ∧ (c, b) ∈ (p!i)} <
              card {i. i < length p ∧ (b, c) ∈ (p!i)}})
    using prof-A winner Collect-cong
    by simp
qed
```

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

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

```
    (∀ a ∈ A − B.
      indep-of-alt n A a ∧ (∀ p. profile A p ⟶ a ∈ reject n A p))
  using assms
  unfolding disjoint-compatibility-def
  by metis
hence
  ∃ B ⊆ A.
    (∀ a ∈ A − B.
      indep-of-alt n A a ∧ (∀ p. profile A p ⟶ a ∈ reject n A p)) ∧
    (∀ a ∈ B.
      indep-of-alt m A a ∧ (∀ p. profile A p ⟶ a ∈ reject m A p))
  by auto
hence ∃ B ⊆ A.
      (∀ a ∈ A − B.
        indep-of-alt n A a ∧ (∀ p. profile A p ⟶ a ∈ reject n A p)) ∧
      (∀ a ∈ A − (A − B).
        indep-of-alt m A a ∧ (∀ p. profile A p ⟶ a ∈ reject m A p))
  using double-diff order-refl
  by metis
thus ∃ B ⊆ A.
      (∀ a ∈ B.
        indep-of-alt n A a ∧ (∀ p. profile A p ⟶ a ∈ reject n A p)) ∧
      (∀ a ∈ A − B.
        indep-of-alt m A a ∧ (∀ p. profile A p ⟶ a ∈ reject m A p))
  by fastforce
qed
```

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

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

2.1.9 Social Choice Properties

Condorcet Consistency

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

lemma condorcet-consistency':
  fixes m :: 'a Electoral-Module
  shows condorcet-consistency m =
        (electoral-module m ∧
```

```
            (∀ A p a. condorcet-winner A p a ⟶
              (m A p = ({a}, A − (elect m A p), {}))))
proof (safe)
  assume condorcet-consistency m
  thus electoral-module m
    unfolding condorcet-consistency-def
    by metis
next
  fix
    A :: 'a set and
    p :: 'a Profile and
    a :: 'a
  assume
    condorcet-consistency m and
    condorcet-winner A p a
  thus m A p = ({a}, A − elect m A p, {})
    using cond-winner-unique
    unfolding condorcet-consistency-def
    by (metis (mono-tags, lifting))
next
  assume
    electoral-module m and
    ∀ A p a. condorcet-winner A p a ⟶ m A p = ({a}, A − elect m A p, {})
  moreover have
    ∀ m'. condorcet-consistency m' =
      (electoral-module m' ∧
        (∀ A p a. condorcet-winner A p a ⟶
          m' A p = ({a ∈ A. condorcet-winner A p a}, A − elect m' A p, {})))
    unfolding condorcet-consistency-def
    by blast
  moreover have
    ∀ A p a. condorcet-winner A p (a::'a) ⟶
       {b ∈ A. condorcet-winner A p b} = {a}
    using cond-winner-unique
    by (metis (full-types))
  ultimately show condorcet-consistency m
    unfolding condorcet-consistency-def
    using cond-winner-unique
    by presburger
qed

lemma condorcet-consistency'':
  fixes m :: 'a Electoral-Module
  shows condorcet-consistency m =
          (electoral-module m ∧
            (∀ A p a.
              condorcet-winner A p a ⟶ m A p = ({a}, A − {a}, {})))
proof (simp only: condorcet-consistency', safe)
  fix
```

```
    A :: 'a set and
    p :: 'a Profile and
    a :: 'a
  assume
    e-mod: electoral-module m and
    cc: ∀ A p a'. condorcet-winner A p a' ⟶
      m A p = ({a'}, A − elect m A p, {}) and
    c-win: condorcet-winner A p a
  show m A p = ({a}, A − {a}, {})
    using cc c-win fst-conv
    by (metis (mono-tags, lifting))
next
  fix
    A :: 'a set and
    p :: 'a Profile and
    a :: 'a
  assume
    e-mod: electoral-module m and
    cc: ∀ A p a'. condorcet-winner A p a' ⟶ m A p = ({a'}, A − {a'}, {}) and
    c-win: condorcet-winner A p a
  show m A p = ({a}, A − elect m A p, {})
    using cc c-win fst-conv
    by (metis (mono-tags, lifting))
qed
```

(Weak) Monotonicity

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

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

Homogeneity

```
fun times :: nat ⇒ 'a list ⇒ 'a list where
  times n l = concat (replicate n l)

definition homogeneity :: 'a Electoral-Module ⇒ bool where
  homogeneity m ≡
    electoral-module m ∧
      (∀ A p n. profile A p ∧ n > 0 ⟶ m A p = m A (times n p))
```

Anonymity

```
definition anonymity :: 'a Electoral-Module ⇒ bool where
  anonymity m ≡
    electoral-module m ∧
```

$(\forall\ A\ p\ q.\ profile\ A\ p \wedge profile\ A\ q \wedge p <^{\sim}{}^{\sim}> q \longrightarrow m\ A\ p = m\ A\ q)$

end

2.2 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.

2.2.1 Definition

type-synonym *'a Evaluation-Function* = *'a* ⇒ *'a set* ⇒ *'a Profile* ⇒ *nat*

2.2.2 Property

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

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

2.2.3 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 Evaluation-Function* **and**
    A :: *'a set* **and**
    p :: *'a Profile* **and**
    a :: *'a*
  assumes
    rating: *condorcet-rating e* **and**
    f-prof: *finite-profile A p* **and**
    winner: *condorcet-winner A p a*
  shows *e a A p* = *Max* {*e b A p* | *b*. *b* ∈ *A*}
proof −
  let *?set* = {*e b A p* | *b*. *b* ∈ *A*} **and**

```
    ?eMax = Max {e b A p | b. b ∈ A} and
    ?eW = e a A p
  have ?eW ∈ ?set
    using CollectI condorcet-winner.simps winner
    by (metis (mono-tags, lifting))
  moreover have ∀ e ∈ ?set. e ≤ ?eW
  proof (safe)
    fix b :: 'a
    assume b ∈ A
    moreover have ∀ n n'. (n::nat) = n' ⟶ n ≤ n'
      by simp
    ultimately show e b A p ≤ e a A p
      using less-imp-le rating winner
      unfolding condorcet-rating-def
      by (metis (no-types))
  qed
  ultimately have ?eW ∈ ?set ∧ (∀ e ∈ ?set. e ≤ ?eW)
    by blast
  moreover have finite ?set
    using f-prof
    by simp
  moreover have ?set ≠ {}
    using condorcet-winner.simps winner
    by fastforce
  ultimately show ?thesis
    using Max-eq-iff
    by (metis (no-types, lifting))
qed
```

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

```
theorem non-cond-winner-not-max-eval:
  fixes
    e :: 'a Evaluation-Function and
    A :: 'a set and
    p :: 'a Profile and
    a :: 'a and
    b :: 'a
  assumes
    rating: condorcet-rating e and
    f-prof: finite-profile A p and
    winner: condorcet-winner A p a and
    lin-A: b ∈ A and
    loser: a ≠ b
  shows e b A p < Max {e c A p | c. c ∈ A}
proof −
  have e b A p < e a A p
    using lin-A loser rating winner
```

unfolding *condorcet-rating-def*
by *metis*
also have *e a A p = Max* {*e c A p* | *c*. *c* ∈ *A*}
using *cond-winner-imp-max-eval-val f-prof rating winner*
by *fastforce*
finally show *?thesis*
by *simp*
qed

end

2.3 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.

2.3.1 Definition

type-synonym *Threshold-Value* = *nat*

type-synonym *Threshold-Relation* = *nat* ⇒ *nat* ⇒ *bool*

type-synonym *'a Electoral-Set* = *'a set* ⇒ *'a Profile* ⇒ *'a set*

fun *elimination-set* :: *'a Evaluation-Function* ⇒ *Threshold-Value* ⇒
Threshold-Relation ⇒ *'a Electoral-Set* **where**
elimination-set e t r A p = {*a* ∈ *A* . *r* (*e a A p*) *t* }

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

2.3.2 Common Eliminators

fun *less-eliminator* :: *'a Evaluation-Function* ⇒ *Threshold-Value* ⇒

'a Electoral-Module **where**
less-eliminator e t A p = elimination-module e t (<) A p

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

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

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

fun *average* :: *'a Evaluation-Function ⇒ 'a set ⇒ 'a Profile ⇒ Threshold-Value*
where
average e A p = (∑ x ∈ A. e x A p) div (card A)

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

fun *leq-average-eliminator* :: *'a Evaluation-Function ⇒ 'a Electoral-Module* **where**
leq-average-eliminator e A p = leq-eliminator e (average e A p) A p

2.3.3 Auxiliary Lemmas

lemma *score-bounded*:
fixes
e :: *'a ⇒ nat* **and**
A :: *'a set* **and**
a :: *'a*
assumes
a-in-A: *a ∈ A* **and**
fin-A: *finite A*
shows *e a ≤ Max {e x | x. x ∈ A}*
proof −
have *e a ∈ {e x |x. x ∈ A}*
using *a-in-A*
by *blast*
thus *?thesis*
using *fin-A Max-ge*
by *simp*
qed

lemma *max-score-contained*:
fixes
e :: *'a ⇒ nat* **and**
A :: *'a set* **and**

a :: *'a*
assumes
A-not-empty: $A \neq \{\}$ **and**
fin-A: *finite A*
shows $\exists\ b \in A.\ e\ b = Max\ \{e\ x \mid x.\ x \in A\}$
proof −
have *finite* $\{e\ x \mid x.\ x \in A\}$
using *fin-A*
by *simp*
hence $Max\ \{e\ x \mid x.\ x \in A\} \in \{e\ x \mid x.\ x \in A\}$
using *A-not-empty Max-in*
by *blast*
thus *?thesis*
by *auto*
qed

lemma *elimset-in-alts*:
fixes
e :: *'a Evaluation-Function* **and**
t :: *Threshold-Value* **and**
r :: *Threshold-Relation* **and**
A :: *'a set* **and**
p :: *'a Profile*
shows *elimination-set e t r A p* $\subseteq A$
unfolding *elimination-set.simps*
by *safe*

2.3.4 Soundness

lemma *elim-mod-sound*[*simp*]:
fixes
e :: *'a Evaluation-Function* **and**
t :: *Threshold-Value* **and**
r :: *Threshold-Relation*
shows *electoral-module* (*elimination-module e t r*)
unfolding *electoral-module-def*
by *auto*

lemma *less-elim-sound*[*simp*]:
fixes
e :: *'a Evaluation-Function* **and**
t :: *Threshold-Value*
shows *electoral-module* (*less-eliminator e t*)
unfolding *electoral-module-def*
by *auto*

lemma *leq-elim-sound*[*simp*]:
fixes
e :: *'a Evaluation-Function* **and**

t :: *Threshold-Value*
shows *electoral-module* (*leq-eliminator e t*)
unfolding *electoral-module-def*
by *auto*

lemma *max-elim-sound*[*simp*]:
fixes *e* :: *'a Evaluation-Function*
shows *electoral-module* (*max-eliminator e*)
unfolding *electoral-module-def*
by *auto*

lemma *min-elim-sound*[*simp*]:
fixes *e* :: *'a Evaluation-Function*
shows *electoral-module* (*min-eliminator e*)
unfolding *electoral-module-def*
by *auto*

lemma *less-avg-elim-sound*[*simp*]:
fixes *e* :: *'a Evaluation-Function*
shows *electoral-module* (*less-average-eliminator e*)
unfolding *electoral-module-def*
by *auto*

lemma *leq-avg-elim-sound*[*simp*]:
fixes *e* :: *'a Evaluation-Function*
shows *electoral-module* (*leq-average-eliminator e*)
unfolding *electoral-module-def*
by *auto*

2.3.5 Non-Blocking

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

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

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

lemma *max-elim-non-blocking*:
fixes *e* :: *'a Evaluation-Function*
shows *non-blocking* (*max-eliminator e*)
unfolding *non-blocking-def*
using *electoral-module-def*
by *auto*

lemma *min-elim-non-blocking*:
fixes *e* :: *'a Evaluation-Function*
shows *non-blocking* (*min-eliminator e*)
unfolding *non-blocking-def*
using *electoral-module-def*
by *auto*

lemma *less-avg-elim-non-blocking*:
fixes *e* :: *'a Evaluation-Function*
shows *non-blocking* (*less-average-eliminator e*)
unfolding *non-blocking-def*
using *electoral-module-def*
by *auto*

lemma *leq-avg-elim-non-blocking*:
fixes *e* :: *'a Evaluation-Function*
shows *non-blocking* (*leq-average-eliminator e*)
unfolding *non-blocking-def*
using *electoral-module-def*
by *auto*

2.3.6 Non-Electing

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

lemma *less-elim-non-electing*:

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

lemma *leq-elim-non-electing*:
fixes
e :: *'a Evaluation-Function* **and**
t :: *Threshold-Value*
shows *non-electing* (*leq-eliminator e t*)
unfolding *non-electing-def*
by *simp*

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

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

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

lemma *leq-avg-elim-non-electing*:
fixes *e* :: *'a Evaluation-Function*
shows *non-electing* (*leq-average-eliminator e*)
unfolding *non-electing-def*
by *simp*

2.3.7 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 Evaluation-Function*
assumes *condorcet-rating e*
shows *condorcet-compatibility* (*max-eliminator e*)
proof (*unfold condorcet-compatibility-def*, *safe*)

```
  show electoral-module (max-eliminator e)
    by simp
next
  fix
    A :: 'a set and
    p :: 'a Profile and
    a :: 'a
  assume
    c-win: condorcet-winner A p a and
    rej-a: a ∈ reject (max-eliminator e) A p
  have e a A p = Max {e b A p | b. b ∈ A}
    using c-win cond-winner-imp-max-eval-val assms
    by fastforce
  hence a ∉ reject (max-eliminator e) A p
    by simp
  thus False
    using rej-a
    by linarith
next
  fix
    A :: 'a set and
    p :: 'a Profile and
    a :: 'a
  assume a ∈ elect (max-eliminator e) A p
  moreover have a ∉ elect (max-eliminator e) A p
    by simp
  ultimately show False
    by linarith
next
  fix
    A :: 'a set and
    p :: 'a Profile and
    a :: 'a and
    a' :: 'a
  assume
    condorcet-winner A p a and
    a ∈ elect (max-eliminator e) A p
  thus a' ∈ reject (max-eliminator e) A p
    using empty-iff max-elim-non-electing
    unfolding condorcet-winner.simps non-electing-def
    by metis
qed

lemma cr-eval-imp-dcc-max-elim-helper:
  fixes
    A :: 'a set and
    p :: 'a Profile and
    e :: 'a Evaluation-Function and
    a :: 'a
```

```
  assumes
    finite-profile A p and
    condorcet-rating e and
    condorcet-winner A p a
  shows elimination-set e (Max {e b A p | b. b ∈ A}) (<) A p = A − {a}
proof (safe, simp-all, safe)
  assume e a A p < Max {e b A p | b. b ∈ A}
  thus False
    using cond-winner-imp-max-eval-val assms
    by fastforce
next
  fix a' :: 'a
  assume
    a' ∈ A and
    ¬ e a' A p < Max {e b A p | b. b ∈ A}
  thus a' = a
    using non-cond-winner-not-max-eval assms
    by (metis (mono-tags, lifting))
qed
```

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

```
theorem cr-eval-imp-dcc-max-elim[simp]:
  fixes e :: 'a Evaluation-Function
  assumes condorcet-rating e
  shows defer-condorcet-consistency (max-eliminator e)
proof (unfold defer-condorcet-consistency-def, safe, simp)
  fix
    A :: 'a set and
    p :: 'a Profile and
    a :: 'a
  assume winner: condorcet-winner A p a
  hence profile: finite-profile A p
    by simp
  let ?trsh = Max {e b A p | b. b ∈ A}
  show
    max-eliminator e A p =
      ({},
        A − defer (max-eliminator e) A p,
        {b ∈ A. condorcet-winner A p b})
  proof (cases elimination-set e (?trsh) (<) A p ≠ A)
    have elim-set: (elimination-set e ?trsh (<) A p) = A − {a}
      using profile assms winner cr-eval-imp-dcc-max-elim-helper
      by (metis (mono-tags, lifting))
    case True
    hence
      max-eliminator e A p =
        ({},
          (elimination-set e ?trsh (<) A p),
```

```
        A − (elimination-set e ?trsh (<) A p))
    by simp
  also have ... = ({}, A − {a}, {a})
    using elim-set winner
    by auto
  also have ... = ({},A − defer (max-eliminator e) A p, {a})
    using calculation
    by simp
  also have
    ... = ({},
          A − defer (max-eliminator e) A p,
          {b ∈ A. condorcet-winner A p b})
    using cond-winner-unique winner Collect-cong
    by (metis (no-types, lifting))
  finally show ?thesis
    using winner
    by metis
 next
  case False
  moreover have ?trsh = e a A p
    using assms winner
    by (simp add: cond-winner-imp-max-eval-val)
  ultimately show ?thesis
    using winner
    by auto
 qed
qed

end
```

2.4 Aggregator

```
theory Aggregator
 imports Social-Choice-Types/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.

2.4.1 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 A (e, r, d) ∧ well-formed A (e', r', d') ⟶
well-formed A (agg A (e, r, d) (e', r', d'))

2.4.2 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 A (e, r, d) ∧ well-formed 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

2.5 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.

2.5.1 Definition

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

2.5.2 Auxiliary Lemma

lemma *max-agg-rej-set*:
fixes
$A :: 'a\ set$ **and**
$e :: 'a\ set$ **and**
$e' :: 'a\ set$ **and**
$d :: 'a\ set$ **and**
$d' :: 'a\ set$ **and**
$r :: 'a\ set$ **and**
$r' :: 'a\ set$ **and**
$a :: 'a$
assumes
wf-first-mod: *well-formed* A $(e,\ r,\ d)$ **and**
wf-second-mod: *well-formed* A $(e',\ r',\ d')$
shows *reject-r* $(max\text{-}aggregator\ A\ (e,\ r,\ d)\ (e',\ r',\ d')) = r \cap r'$
proof −
have $A - (e \cup d) = r$
using *wf-first-mod*
by (*simp add*: *result-imp-rej*)
moreover have $A - (e' \cup d') = r'$
using *wf-second-mod*
by (*simp add*: *result-imp-rej*)
ultimately have $A - (e \cup e' \cup d \cup d') = r \cap r'$
by *blast*
moreover have $\{l \in A.\ l \notin e \cup e' \cup d \cup d'\} = A - (e \cup e' \cup d \cup d')$
unfolding *set-diff-eq*
by *simp*
ultimately show *reject-r* $(max\text{-}aggregator\ A\ (e,\ r,\ d)\ (e',\ r',\ d')) = r \cap r'$
by *simp*
qed

2.5.3 Soundness

theorem *max-agg-sound*[*simp*]: *aggregator max-aggregator*
proof (*unfold aggregator-def*, *simp*, *safe*)
fix
$A :: 'a\ set$ **and**
$e :: 'a\ set$ **and**
$e' :: 'a\ set$ **and**
$d :: 'a\ set$ **and**
$d' :: 'a\ set$ **and**
$r :: 'a\ set$ **and**
$r' :: 'a\ set$ **and**
$a :: 'a$
assume
$e' \cup r' \cup d' = e \cup r \cup d$ **and**
$a \notin d$ **and**
$a \notin r$ **and**
$a \in e'$

thus $a \in e$
by *auto*
next
fix
$A :: 'a\ set$ **and**
$e :: 'a\ set$ **and**
$e' :: 'a\ set$ **and**
$d :: 'a\ set$ **and**
$d' :: 'a\ set$ **and**
$r :: 'a\ set$ **and**
$r' :: 'a\ set$ **and**
$a :: 'a$
assume
$e' \cup r' \cup d' = e \cup r \cup d$ **and**
$a \notin d$ **and**
$a \notin r$ **and**
$a \in d'$
thus $a \in e$
by *auto*
qed

2.5.4 Properties

The max-aggregator is conservative.

theorem *max-agg-consv*[*simp*]: *agg-conservative max-aggregator*
proof (*unfold agg-conservative-def*, *safe*)
show *aggregator max-aggregator*
using *max-agg-sound*
by *metis*
next
fix
$A :: 'a\ set$ **and**
$e :: 'a\ set$ **and**
$e' :: 'a\ set$ **and**
$d :: 'a\ set$ **and**
$d' :: 'a\ set$ **and**
$r :: 'a\ set$ **and**
$r' :: 'a\ set$ **and**
$a :: 'a$
assume
elect-a: $a \in$ *elect-r* (*max-aggregator* A (e, r, d) (e', r', d')) **and**
a-not-in-e': $a \notin e'$
have $a \in e \cup e'$
using *elect-a*
by *simp*
thus $a \in e$
using *a-not-in-e'*
by *simp*
next

fix
$A :: 'a\ set$ **and**
$e :: 'a\ set$ **and**
$e' :: 'a\ set$ **and**
$d :: 'a\ set$ **and**
$d' :: 'a\ set$ **and**
$r :: 'a\ set$ **and**
$r' :: 'a\ set$ **and**
$a :: 'a$
assume
wf-result: *well-formed* A (e', r', d') **and**
reject-a: $a \in$ *reject-r* (*max-aggregator* A (e, r, d) (e', r', d')) **and**
a-not-in-r': $a \notin r'$
have $a \in r \cup r'$
using *wf-result reject-a*
by *force*
thus $a \in r$
using *a-not-in-r'*
by *simp*
next
fix
$A :: 'a\ set$ **and**
$e :: 'a\ set$ **and**
$e' :: 'a\ set$ **and**
$d :: 'a\ set$ **and**
$d' :: 'a\ set$ **and**
$r :: 'a\ set$ **and**
$r' :: 'a\ set$ **and**
$a :: 'a$
assume
defer-a: $a \in$ *defer-r* (*max-aggregator* A (e, r, d) (e', r', d')) **and**
a-not-in-d': $a \notin d'$
have $a \in d \cup d'$
using *defer-a*
by *force*
thus $a \in d$
using *a-not-in-d'*
by *simp*
qed

The max-aggregator is commutative.

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

end

2.6 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.

2.6.1 Definition

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

end

2.7 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.

2.7.1 Definition

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

end

Chapter 3

Basic Modules

3.1 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.

3.1.1 Definition

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

3.1.2 Soundness

theorem *def-mod-sound*[*simp*]: *electoral-module defer-module*
  unfolding *electoral-module-def*
  by *simp*

3.1.3 Properties

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

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

end

3.2 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.

3.2.1 Definition

fun *elect-first-module* :: *'a Electoral-Module* **where**
  elect-first-module A p =
    ($\{a \in A.\ above\ (p!0)\ a = \{a\}\}$,
    $\{a \in A.\ above\ (p!0)\ a \neq \{a\}\}$,
    $\{\}$)

3.2.2 Soundness

theorem *elect-first-mod-sound*: *electoral-module elect-first-module*
proof (*unfold electoral-module-def*, *safe*)
  fix
    A :: *'a set* **and**
    p :: *'a Profile*
  have $\{a \in A.\ above\ (p!0)\ a = \{a\}\} \cup \{a \in A.\ above\ (p!0)\ a \neq \{a\}\} = A$
    by *blast*
  hence *set-equals-partition A* (*elect-first-module A p*)
    by *simp*
  moreover have
    $\forall\ a \in A.\ a \notin \{a' \in A.\ above\ (p!0)\ a' = \{a'\}\} \vee$
              $a \notin \{a' \in A.\ above\ (p!0)\ a' \neq \{a'\}\}$
    by *simp*
  hence $\{a \in A.\ above\ (p!0)\ a = \{a\}\} \cap \{a \in A.\ above\ (p!0)\ a \neq \{a\}\} = \{\}$
    by *blast*
  hence *disjoint3* (*elect-first-module A p*)
    by *simp*
  ultimately show *well-formed A* (*elect-first-module A p*)
    by *simp*
qed

end

3.3 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.

3.3.1 Definition

type-synonym *'a Consensus-Class = 'a Consensus × 'a Electoral-Module*

fun *consensus-𝒦 :: 'a Consensus-Class ⇒ 'a Consensus* **where**
consensus-𝒦 K = fst K

fun *rule-𝒦 :: 'a Consensus-Class ⇒ 'a Electoral-Module* **where**
rule-𝒦 K = snd K

3.3.2 Consensus Choice

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 Consensus ⇒ 'a Electoral-Module ⇒ bool* **where**
well-formed c m ≡
∀ A p p'. profile A p ∧ profile A p' ∧ c (A, p) ∧ c (A, p') ⟶ m A p = m A p'

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

3.3.3 Auxiliary Lemmas

lemma *unanimity'-consensus-imp-elect-fst-mod-well-formed*:
fixes *a :: 'a*
shows
well-formed (λ c. nonempty-set$_{\mathcal{C}}$ c ∧ nonempty-profile$_{\mathcal{C}}$ c ∧ equal-top$_{\mathcal{C}}$' a c)
elect-first-module
proof (*unfold well-formed-def*, *safe*)
fix
a :: 'a **and**
A :: 'a set **and**
p :: 'a Profile **and**

```
    p' :: 'a Profile
  let ?cond =
    λ c. nonempty-set_C c ∧ nonempty-profile_C c ∧ equal-top_C' a c
  assume
    prof-p: profile A p and
    prof-p': profile A p' and
    eq-top-p: equal-top_C' a (A, p) and
    eq-top-p': equal-top_C' a (A, p') and
    not-empty-A: nonempty-set_C (A, p) and
    not-empty-A': nonempty-set_C (A, p') and
    not-empty-p: nonempty-profile_C (A, p) and
    not-empty-p': nonempty-profile_C (A, p')
  hence
    cond-Ap: ?cond (A, p) and
    cond-Ap': ?cond (A, p')
    by simp-all
  have ∀ a' ∈ A. (above (p!0) a' = {a'}) = (above (p'!0) a' = {a'})
  proof
    fix a' :: 'a
    assume a-in-A: a' ∈ A
    show (above (p!0) a' = {a'}) = (above (p'!0) a' = {a'})
    proof (cases)
      assume a' = a
      thus ?thesis
        using cond-Ap cond-Ap'
        by simp
    next
      assume a'-neq-a: a' ≠ a
      have lens-p-and-p'-ok: 0 < length p ∧ 0 < length p'
        using not-empty-p not-empty-p'
        by simp
      hence A ≠ {} ∧ linear-order-on A (p!0) ∧ linear-order-on A (p'!0)
        using not-empty-A not-empty-A' prof-p prof-p'
        unfolding profile-def
        by simp
      hence (above (p!0) a = {a} ∧ above (p!0) a' = {a'} ⟶ a = a') ∧
            (above (p'!0) a = {a} ∧ above (p'!0) a' = {a'} ⟶ a = a')
        using a-in-A above-trans insert-iff singletonD subset-singletonD
              cond-Ap' insert-not-empty
        unfolding order-on-defs total-on-def equal-top_C'.simps
        by metis
      thus ?thesis
        using a'-neq-a eq-top-p' eq-top-p lens-p-and-p'-ok
        by simp
    qed
  qed
  thus elect-first-module A p = elect-first-module A p'
    by auto
qed
```

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

3.3.4 Consensus Rules

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

Unanimity condition.

definition *unanimity* :: $'a$ *Consensus-Class* **where**
unanimity = *consensus-choice unanimity*$_{\mathcal{C}}$ *elect-first-module*

Strong unanimity condition.

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

3.3.5 Properties

definition *consensus-rule-anonymity* :: $'a$ *Consensus-Class* $\Rightarrow$ *bool* **where**
consensus-rule-anonymity c $\equiv$
$\forall$ *A p q. profile A p* $\wedge$ *profile A q* $\wedge$ *p* $<^{\sim}{}^{\sim}>$ *q* $\wedge$ *consensus-*$\mathcal{K}$ *c* (*A*, *p*)
$\longrightarrow$ *consensus-*$\mathcal{K}$ *c* (*A*, *q*) $\wedge$ (*rule-*$\mathcal{K}$ *c A p* = *rule-*$\mathcal{K}$ *c A q*)

3.3.6 Inference Rules

lemma *consensus-choice-anonymous*:
fixes
$\alpha :: 'a$ *Consensus* **and**
$\beta :: 'a$ *Consensus* **and**
$m :: 'a$ *Electoral-Module* **and**
$\beta' :: 'b \Rightarrow 'a$ *Consensus*
assumes
beta-sat: $\beta = (\lambda\ E.\ \exists\ a.\ \beta'\ a\ E)$ **and**
beta'-anon: $\forall\ x.$ *consensus-anonymity* ($\beta'\ x$) **and**
anon-cons-cond: *consensus-anonymity* α **and**
conditions-univ: $\forall\ x.$ *well-formed* ($\lambda\ E.\ \alpha\ E \wedge \beta'\ x\ E$) *m*
shows *consensus-rule-anonymity* (*consensus-choice* ($\lambda\ E.\ \alpha\ E \wedge \beta\ E$) *m*)
proof (*unfold consensus-rule-anonymity-def*, *safe*)
fix
$A :: 'a$ *set* **and**
$p :: 'a$ *Profile* **and**
$q :: 'a$ *Profile*
assume

prof-p: *profile A p* **and**
prof-q: *profile A q* **and**
perm: $p <\sim\sim> q$ **and**
consensus-cond: *consensus-$\mathcal{K}$* (*consensus-choice* (λ *E*. α *E* $\wedge$ β *E*) *m*) (*A*, *p*)
hence (λ *E*. α *E* $\wedge$ β *E*) (*A*, *p*)
by *simp*
hence
alpha-Ap: α (*A*, *p*) **and**
beta-Ap: β (*A*, *p*)
by *simp-all*
have *alpha-A-perm-p*: α (*A*, *q*)
using *anon-cons-cond alpha-Ap perm prof-p prof-q*
unfolding *consensus-anonymity-def*
by *metis*
moreover have β (*A*, *q*)
using *beta′-anon*
unfolding *consensus-anonymity-def*
using *beta-Ap beta-sat ex-anon-cons-imp-cons-anonymous perm prof-p prof-q*
by *blast*
ultimately show *em-cond-perm*:
consensus-$\mathcal{K}$ (*consensus-choice* (λ *E*. α *E* $\wedge$ β *E*) *m*) (*A*, *q*)
by *simp*
have $\exists$ *x*. β' *x* (*A*, *p*)
using *beta-Ap beta-sat*
by *simp*
then obtain *x* **where**
beta′-x-Ap: β' *x* (*A*, *p*)
by *metis*
hence *beta′-x-A-perm-p*: β' *x* (*A*, *q*)
using *beta′-anon perm prof-p prof-q*
unfolding *consensus-anonymity-def*
by *metis*
have *m A p* = *m A q*
using *alpha-Ap alpha-A-perm-p beta′-x-Ap beta′-x-A-perm-p conditions-univ prof-p prof-q*
unfolding *well-formed-def*
by *metis*
thus *rule-$\mathcal{K}$* (*consensus-choice* (λ *E*. α *E* $\wedge$ β *E*) *m*) *A p* =
rule-$\mathcal{K}$ (*consensus-choice* (λ *E*. α *E* $\wedge$ β *E*) *m*) *A q*
using *consensus-cond em-cond-perm*
by *simp*
qed

3.3.7 Theorems

lemma *unanimity-anonymous*: *consensus-rule-anonymity unanimity*
proof (*unfold unanimity-def*)
let *?ne-cond* = (λ *c*. *nonempty-set*$_{\mathcal{C}}$ *c* $\wedge$ *nonempty-profile*$_{\mathcal{C}}$ *c*)

have *consensus-anonymity ?ne-cond*
using *nonempty-set-cons-anonymous nonempty-profile-cons-anonymous*
unfolding *consensus-anonymity-def*
by *metis*
moreover have $\textit{equal-top}_{\mathcal{C}} = (\lambda\ c.\ \exists\ a.\ \textit{equal-top}_{\mathcal{C}}{}'\ a\ c)$
by *fastforce*
ultimately have *consensus-rule-anonymity*
(*consensus-choice*
(λ *c. nonempty-set*$_{\mathcal{C}}$ *c* $\wedge$ *nonempty-profile*$_{\mathcal{C}}$ *c* $\wedge$ *equal-top*$_{\mathcal{C}}$ *c*) *elect-first-module*)
using *consensus-choice-anonymous*[*of equal-top*$_{\mathcal{C}}$ *equal-top*$_{\mathcal{C}}{}'$ *?ne-cond*]
equal-top-cons'-anonymous unanimity'-consensus-imp-elect-fst-mod-well-formed
by *fastforce*
moreover have
unanimity$_{\mathcal{C}}$ = (λ *c. nonempty-set*$_{\mathcal{C}}$ *c* $\wedge$ *nonempty-profile*$_{\mathcal{C}}$ *c* $\wedge$ *equal-top*$_{\mathcal{C}}$ *c*)
by *force*
hence *consensus-choice*
(λ *c. nonempty-set*$_{\mathcal{C}}$ *c* $\wedge$ *nonempty-profile*$_{\mathcal{C}}$ *c* $\wedge$ *equal-top*$_{\mathcal{C}}$ *c*)
elect-first-module =
consensus-choice unanimity$_{\mathcal{C}}$ *elect-first-module*
by *metis*
ultimately show *consensus-rule-anonymity* (*consensus-choice unanimity*$_{\mathcal{C}}$ *elect-first-module*)
by *metis*
qed

lemma *strong-unanimity-anonymous*: *consensus-rule-anonymity strong-unanimity*
proof (*unfold strong-unanimity-def*)
have *consensus-anonymity* (λ *c. nonempty-set*$_{\mathcal{C}}$ *c* $\wedge$ *nonempty-profile*$_{\mathcal{C}}$ *c*)
using *nonempty-set-cons-anonymous nonempty-profile-cons-anonymous*
unfolding *consensus-anonymity-def*
by *metis*
moreover have $\textit{equal-vote}_{\mathcal{C}} = (\lambda\ c.\ \exists\ v.\ \textit{equal-vote}_{\mathcal{C}}{}'\ v\ c)$
by *fastforce*
ultimately have
consensus-rule-anonymity
(*consensus-choice*
(λ *c. nonempty-set*$_{\mathcal{C}}$ *c* $\wedge$ *nonempty-profile*$_{\mathcal{C}}$ *c* $\wedge$ *equal-vote*$_{\mathcal{C}}$ *c*) *elect-first-module*)
using *consensus-choice-anonymous*[*of equal-vote*$_{\mathcal{C}}$ *equal-vote*$_{\mathcal{C}}{}'$
λ *c. nonempty-set*$_{\mathcal{C}}$ *c* $\wedge$ *nonempty-profile*$_{\mathcal{C}}$ *c*]
nonempty-set-cons-anonymous nonempty-profile-cons-anonymous eq-vote-cons'-anonymous
strong-unanimity'consensus-imp-elect-fst-mod-well-formed
by *fastforce*
moreover have *strong-unanimity*$_{\mathcal{C}}$ =
(λ *c. nonempty-set*$_{\mathcal{C}}$ *c* $\wedge$ *nonempty-profile*$_{\mathcal{C}}$ *c* $\wedge$ *equal-vote*$_{\mathcal{C}}$ *c*)
by *force*
hence
consensus-choice (λ *c. nonempty-set*$_{\mathcal{C}}$ *c* $\wedge$ *nonempty-profile*$_{\mathcal{C}}$ *c* $\wedge$ *equal-vote*$_{\mathcal{C}}$ *c*)
elect-first-module =
consensus-choice strong-unanimity$_{\mathcal{C}}$ *elect-first-module*
by *metis*

ultimately show
consensus-rule-anonymity (*consensus-choice strong-unanimity*$_{\mathcal{C}}$ *elect-first-module*)
by *metis*
qed

end

3.4 Distance Rationalization

theory *Distance-Rationalization*
imports *HOL−Combinatorics.Multiset-Permutations*
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.

3.4.1 Definitions

fun $\mathcal{K}_{\mathcal{E}}$ :: *'a Consensus-Class* ⇒ *'a* ⇒ *'a Election set* **where**
$\mathcal{K}_{\mathcal{E}}$ *K a* =
{(*A*, *p*) | *A p*.
(*consensus-*$\mathcal{K}$ *K*) (*A*, *p*) ∧ *finite-profile A p* ∧ *elect* (*rule-*$\mathcal{K}$ *K*) *A p* = {*a*}}

fun *score* :: *'a Election Distance* ⇒ *'a Consensus-Class* ⇒ *'a Election* ⇒
'a ⇒ *ereal* **where**
score d K E a = *Inf* (*d E* ‘ ($\mathcal{K}_{\mathcal{E}}$ *K a*))

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*))

fun $\mathcal{R}_{\mathcal{W}}$ :: *'a Election Distance* ⇒ *'a Consensus-Class* ⇒ *'a set* ⇒ *'a Profile* ⇒
'a set **where**
$\mathcal{R}_{\mathcal{W}}$ *d K A p* = *arg-min-set* (*score d K* (*A*, *p*)) *A*

fun *distance-*$\mathcal{R}$ :: *'a Election Distance* ⇒ *'a Consensus-Class* ⇒
'a Electoral-Module **where**
distance-$\mathcal{R}$ *d K A p* = ($\mathcal{R}_{\mathcal{W}}$ *d K A p*, *A* − $\mathcal{R}_{\mathcal{W}}$ *d K A p*, {})

3.4.2 Standard Definitions

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

fun *profile-permutations* :: *nat* ⇒ *'a set* ⇒ (*'a Profile*) *set* **where**
profile-permutations n A =
(*if permutations-of-set A* = {}
then {} *else listset* (*replicate n* (*pl-α* ‘ *permutations-of-set A*)))

fun $\mathcal{K}_{\mathcal{E}}$*-std* :: *'a Consensus-Class* ⇒ *'a* ⇒ *'a set* ⇒ *nat* ⇒ *'a Election set* **where**
$\mathcal{K}_{\mathcal{E}}$*-std K a A n* =
(λ *p*. (*A*, *p*)) ‘
(*Set.filter*
(λ *p*. (*consensus-*$\mathcal{K}$ *K*) (*A*, *p*) ∧ *elect* (*rule-*$\mathcal{K}$ *K*) *A p* = {*a*})
(*profile-permutations n A*))

fun *score-std* :: *'a Election Distance* ⇒ *'a Consensus-Class* ⇒ *'a Election* ⇒
'a ⇒ *ereal* **where**
score-std d K E a =
(*if* $\mathcal{K}_{\mathcal{E}}$*-std K a* (*alts-*$\mathcal{E}$ *E*) (*length* (*prof-*$\mathcal{E}$ *E*)) = {}
then ∞ *else Min* (*d E* ‘ ($\mathcal{K}_{\mathcal{E}}$*-std K a* (*alts-*$\mathcal{E}$ *E*) (*length* (*prof-*$\mathcal{E}$ *E*)))))

fun $\mathcal{R}_{\mathcal{W}}$*-std* :: *'a Election Distance* ⇒ *'a Consensus-Class* ⇒ *'a set* ⇒
'a Profile ⇒ *'a set* **where**
$\mathcal{R}_{\mathcal{W}}$*-std d K A p* = *arg-min-set* (*score-std d K* (*A*, *p*)) *A*

fun *distance-*$\mathcal{R}$*-std* :: *'a Election Distance* ⇒ *'a Consensus-Class* ⇒
'a Electoral-Module **where**
distance-$\mathcal{R}$*-std d K A p* = ($\mathcal{R}_{\mathcal{W}}$*-std d K A p*, *A* − $\mathcal{R}_{\mathcal{W}}$*-std d K A p*, {})

3.4.3 Auxiliary Lemmas

lemma *lin-ord-pl-α*:
fixes
r :: *'a rel* **and**
A :: *'a set*
assumes
lin-ord-r: *linear-order-on A r* **and**
fin-A: *finite A*
shows *r* ∈ *pl-α* ‘ *permutations-of-set A*
proof −
let *?φ* = λ *a*. *card* ((*underS r a*) ∩ *A*)
let *?inv* = *the-inv-into A ?φ*
let *?l* = *map* (λ *x*. *?inv x*) (*rev* [*0* ..< *card A*])
have *antisym*: ∀ *a b*. *a* ∉ (*underS r b*) ∩ *A* ∨ *b* ∉ (*underS r a*) ∩ *A*
using *lin-ord-r*
unfolding *underS-def linear-order-on-def partial-order-on-def antisym-def*

by *simp*
hence $\forall\ a\ b\ c.$
$a \in (underS\ r\ b) \cap A \longrightarrow b \in (underS\ r\ c) \cap A \longrightarrow a \in (underS\ r\ c) \cap A$
using *lin-ord-r CollectD CollectI transD IntE IntI*
unfolding *underS-def linear-order-on-def partial-order-on-def preorder-on-def trans-def*
by (*metis* (*mono-tags*, *lifting*))
hence $\forall\ a\ b.\ a \in (underS\ r\ b) \cap A \longrightarrow (underS\ r\ a) \cap A \subset (underS\ r\ b) \cap A$
using *antisym*
by *blast*
hence *mono*: $\forall\ a\ b.\ a \in (underS\ r\ b) \cap A \longrightarrow ?\varphi\ a < ?\varphi\ b$
using *fin-A*
by (*simp add*: *psubset-card-mono*)
moreover have *total-underS*:
$\forall\ a\ b.\ a \in A \wedge b \in A \wedge a \neq b \longrightarrow$
$a \in (underS\ r\ b) \cap A \vee b \in (underS\ r\ a) \cap A$
using *lin-ord-r totalp-onD totalp-on-total-on-eq*
unfolding *underS-def linear-order-on-def partial-order-on-def antisym-def*
by *fastforce*
ultimately have $\forall\ a\ b.\ a \in A \wedge b \in A \wedge a \neq b \longrightarrow ?\varphi\ a \neq ?\varphi\ b$
using *order-less-imp-not-eq2*
by *metis*
hence *inj*: *inj-on* $?\varphi\ A$
unfolding *inj-on-def*
by *metis*
have *in-bounds*: $\forall\ a \in A.\ ?\varphi\ a < card\ A$
using *CollectD IntD1 card-seteq inf-le2 linorder-le-less-linear fin-A*
unfolding *underS-def*
by (*metis* (*mono-tags*, *lifting*))
hence $?\varphi\ `\ A \subseteq \{0\ ..<\ card\ A\}$
using *atLeast0LessThan*
by *blast*
moreover have $card\ (?\varphi\ `\ A) = card\ A$
using *inj card-image*
by *metis*
ultimately have $?\varphi\ `\ A = \{0\ ..<\ card\ A\}$
by (*simp add*: *card-subset-eq*)
hence *bij*: *bij-betw* $?\varphi\ A\ \{0\ ..<\ card\ A\}$
using *inj*
unfolding *bij-betw-def*
by *presburger*
hence *bij-inv*: *bij-betw* $?inv\ \{0\ ..<\ card\ A\}\ A$
using *bij-betw-the-inv-into*
by *metis*
hence $?inv\ `\ \{0\ ..<\ card\ A\} = A$
using *bij-inv*
unfolding *bij-betw-def*
by *presburger*
hence $set\ ?l = A$

by *simp*
moreover have *dist-inv-of-rev*: *distinct ?l*
using *bij-inv bij-betw-imp-inj-on*
by (*simp add*: *distinct-map*)
ultimately have *?l* ∈ *permutations-of-set A*
by *blast*
moreover have *index-eq*: ∀ *a* ∈ *A*. *index ?l a* = *card A* − *1* − *?φ a*
proof (*safe*)
fix *a* :: *'a*
assume *a-in-A*: *a* ∈ *A*
have ∀ *l*. ∀ *i* < *length l*. (*rev l*)!*i* = *l*!(*length l* − *1* − *i*)
using *rev-nth*
by *auto*
hence ∀ *i* < *length* [*0* ..< *card A*].
(*rev* [*0* ..< *card A*])!*i* = [*0* ..< *card A*]!(*length* [*0* ..< *card A*] − *1* − *i*)
by *blast*
moreover have ∀ *i* < *card A*. [*0* ..< *card A*]!*i* = *i*
by *simp*
moreover have *len-card-A*: *length* [*0* ..< *card A*] = *card A*
by *simp*
ultimately have ∀ *i* < *card A*. (*rev* [*0* ..< *card A*])!*i* = *card A* − *1* − *i*
using *diff-Suc-eq-diff-pred diff-less diff-self-eq-0 less-imp-diff-less zero-less-Suc*
by *metis*
moreover have ∀ *i* < *card A*. *?l*!*i* = *?inv* ((*rev* [*0* ..< *card A*])!*i*)
by *simp*
ultimately have ∀ *i* < *card A*. *?l*!*i* = *?inv* (*card A* − *1* − *i*)
by *presburger*
moreover have
card A − *1* − (*card A* − *1* − *card* (*underS r a* ∩ *A*))
= *card* (*underS r a* ∩ *A*)
using *in-bounds a-in-A*
by *auto*
moreover have *?inv* (*card* (*underS r a* ∩ *A*)) = *a*
using *a-in-A inj the-inv-into-f-f*
by *fastforce*
ultimately have *?l*!(*card A* − *1* − *card* (*underS r a* ∩ *A*)) = *a*
using *in-bounds a-in-A diff-less-Suc Suc-diff-Suc*
diff-Suc-eq-diff-pred not-less-eq
by *metis*
thus *index ?l a* = *card A* − *1* − *card* (*underS r a* ∩ *A*)
using *bij-inv dist-inv-of-rev a-in-A len-card-A card-Diff-singleton*
card-Suc-Diff1 diff-less-Suc index-nth-id length-map length-rev
card.infinite in-bounds not-less-zero
by *metis*
qed
moreover have *pl-α ?l* = *r*
proof
show *r* ⊆ *pl-α ?l*
proof (*unfold pl-α-def*, *auto*)

```
    fix
      a :: 'a and
      b :: 'a
    assume (a, b) ∈ r
    hence a ∈ A
      using lin-ord-r
     unfolding linear-order-on-def partial-order-on-def preorder-on-def refl-on-def
      by blast
    thus a ∈ ?inv ‘ {0 ..< card A}
      using bij-inv bij-betw-def
      by metis
  next
    fix
      a :: 'a and
      b :: 'a
    assume (a, b) ∈ r
    hence b ∈ A
      using lin-ord-r
     unfolding linear-order-on-def partial-order-on-def preorder-on-def refl-on-def
      by blast
    thus b ∈ ?inv ‘ {0 ..< card A}
      using bij-inv bij-betw-def
      by metis
  next
    fix
      a :: 'a and
      b :: 'a
    assume rel: (a, b) ∈ r
    hence a-b-in-A: a ∈ A ∧ b ∈ A
      using lin-ord-r
     unfolding linear-order-on-def partial-order-on-def preorder-on-def refl-on-def
      by blast
    moreover have a ∉ underS r b ⟶ a = b
      using lin-ord-r rel
      unfolding underS-def
      by simp
    ultimately have ?φ a ≤ ?φ b
      using mono le-eq-less-or-eq
      by blast
    thus index ?l b ≤ index ?l a
      using index-eq a-b-in-A diff-le-mono2
      by metis
  qed
next
  show pl-α ?l ⊆ r
  proof (unfold pl-α-def, clarsimp)
    fix
      a :: nat and
      b :: nat
```

assume
a-lt-card-A: $a < card\ A$ **and**
b-lt-card-A: $b < card\ A$ **and**
index-b-lte-a: $index\ ?l\ (?inv\ b) \leq index\ ?l\ (?inv\ a)$
have *inv-a-in-A*: $(?inv\ a) \in A$
using *bij-inv a-lt-card-A atLeast0LessThan*
unfolding *bij-betw-def*
by *blast*
moreover have *inv-b-in-A*: $(?inv\ b) \in A$
using *bij-inv b-lt-card-A atLeast0LessThan*
unfolding *bij-betw-def*
by *blast*
ultimately have $card\ A - 1 - ?\varphi\ (?inv\ b) \leq card\ A - 1 - ?\varphi\ (?inv\ a)$
using *index-b-lte-a index-eq*
by *metis*
moreover have $\forall\ a < card\ A.\ ?\varphi\ (?inv\ a) < card\ A$
using *bij-inv bij*
unfolding *bij-betw-def*
by *fastforce*
hence $?\varphi\ (?inv\ b) \leq card\ A - 1 \wedge ?\varphi\ (?inv\ a) \leq card\ A - 1$
using *a-lt-card-A b-lt-card-A*
by *fastforce*
ultimately have $?\varphi\ (?inv\ b) \geq ?\varphi\ (?inv\ a)$
using *le-diff-iff'*
by *blast*
hence $?\varphi\ (?inv\ a) < ?\varphi\ (?inv\ b) \vee ?\varphi\ (?inv\ a) = ?\varphi\ (?inv\ b)$
by *auto*
moreover have $\forall\ a\ b.\ a \in A \wedge b \in A \wedge ?\varphi\ a < ?\varphi\ b \longrightarrow a \in underS\ r\ b$
using *mono total-underS antisym IntD1 order-less-not-sym*
by *metis*
hence $?\varphi\ (?inv\ a) < ?\varphi\ (?inv\ b) \longrightarrow (?inv\ a,\ ?inv\ b) \in r$
using *inv-a-in-A inv-b-in-A*
unfolding *underS-def*
by *blast*
moreover have $\forall\ a\ b.\ a \in A \wedge b \in A \wedge ?\varphi\ a = ?\varphi\ b \longrightarrow a = b$
using *mono total-underS antisym order-less-not-sym*
by *metis*
hence $?\varphi\ (?inv\ a) = ?\varphi\ (?inv\ b) \longrightarrow (?inv\ a,\ ?inv\ b) \in r$
using *lin-ord-r inv-a-in-A inv-b-in-A*
unfolding *linear-order-on-def partial-order-on-def preorder-on-def refl-on-def*
by *metis*
ultimately show $(?inv\ a,\ ?inv\ b) \in r$
by *metis*
qed
qed
ultimately show $r \in pl\text{-}\alpha\ `\ permutations\text{-}of\text{-}set\ A$
by *blast*
qed

lemma *profile-permutation-set*:
fixes
A :: *'a set* **and**
p :: *'a Profile*
shows *profile-permutations* (*length p*) *A* =
{*p'* :: *'a Profile*. *finite-profile A p'* ∧ *length p'* = *length p*}
proof (*cases* ¬ *finite A*, *clarsimp*)
case *fin-A*: *False*
show *?thesis*
proof (*induction p*, *safe*)
case *not-zero-lt-len-p'*: *Nil*
show *finite A*
using *fin-A*
by *simp*
fix *p'* :: *'a Profile*
assume *p'-in-prof*: *p'* ∈ *profile-permutations* (*length* []) *A*
hence *profile-permutations* (*length* []) *A* ≠ {}
by *force*
hence *perms-nonempty*: *pl-α* ' *permutations-of-set A* ≠ {}
by *auto*
thus *len-eq*: *length p'* = *length* []
using *p'-in-prof*
by *simp*
thus *profile A p'*
unfolding *profile-def*
by *force*
next
case *not-zero-lt-len-p'*: *Nil*
fix *p'* :: *'a Profile*
assume *length p'* = *length* []
hence [] = *p'*
by *simp*
moreover have {*q*. *finite-profile A q* ∧ *length q* = *length* []} ⊆ {[]}
using *not-zero-lt-len-p'*
by *auto*
moreover have *profile-permutations* (*length* []) *A* = {[]}
using *fin-A not-zero-lt-len-p'*
by *simp*
ultimately show *p'*∈ *profile-permutations* (*length* []) *A*
by *simp*
next
case *zero-lt-len-p*: (*Cons r p'*)
fix *p'* :: *'a Profile*
from *fin-A*
show *finite A*
by *simp*
fix
r :: *'a Preference-Relation* **and**
q :: *'a Profile*

assume
prof-perms-eq-set-induct:
profile-permutations (*length q*) *A* =
{*q'*. *finite-profile A q'* ∧ *length q'* = *length q*} **and**
p'-in-prof: *p'* ∈ *profile-permutations* (*length* (*r*#*q*)) *A*
show *len-eq*: *length p'* = *length* (*r*#*q*)
using *all-ls-elems-same-len fin-A length-replicate p'-in-prof*
permutations-of-set-empty-iff profile-permutations.simps
by (*metis* (*no-types*))
have *perms-nonempty*: *pl-α* ‘ *permutations-of-set A* ≠ {}
using *p'-in-prof prof-perms-eq-set-induct*
by *auto*
have *length* (*replicate* (*length q*) (*pl-α* ‘ *permutations-of-set A*)) = *length q*
by *simp*
hence ∀ *q'* ∈ *listset* (*replicate* (*length q*) (*pl-α* ‘ *permutations-of-set A*)).
length q' = *length q*
using *all-ls-elems-same-len*
by *metis*
show *profile A p'*
proof (*unfold profile-def*, *safe*)
fix *i* :: *nat*
assume *i-lt-len-p'*: *i* < *length p'*
hence *p'*!*i* ∈ *replicate* (*length p'*) (*pl-α* ‘ *permutations-of-set A*)!*i*
using *p'-in-prof perms-nonempty all-ls-elems-in-ls-set image-is-empty length-replicate*
all-ls-elems-same-len
unfolding *profile-permutations.simps*
by *metis*
hence *p'*!*i* ∈ *pl-α* ‘ *permutations-of-set A*
using *i-lt-len-p'*
by *simp*
hence *relation-of*:
p'!*i* ∈ {*relation-of* (λ *a a'*. *rank-l l a'* ≤ *rank-l l a*) (*set l*) |
l. *l* ∈ *permutations-of-set A*}
proof (*safe*)
fix *l* :: *'a Preference-List*
assume
i-th-rel: *p'*!*i* = *pl-α l* **and**
perm-l: *l* ∈ *permutations-of-set A*
have *rel-of-set-l-eq-l-list*: *relation-of* (λ *a a'*. *a* ≲*l a'*) (*set l*) = *pl-α l*
using *rel-of-pref-pred-for-set-eq-list-to-rel*
by *blast*
have *relation-of* (λ *a a'*. *rank-l l a'* ≤ *rank-l l a*) (*set l*) = *pl-α l*
proof (*unfold relation-of-def rank-l.simps*, *safe*)
fix
a :: *'a* **and**
b :: *'a*
assume
idx-b-lte-idx-a: (*if b* ∈ *set l then index l b* + *1 else 0*) ≤
(*if a* ∈ *set l then index l a* + *1 else 0*) **and**

a-in-l: $a \in set\ l$ **and**
b-in-l : $b \in set\ l$
moreover have $\{(a', b').\ (a', b') \in set\ l \times set\ l \wedge a' \lesssim_l b'\} = pl\text{-}\alpha\ l$
using *rel-of-set-l-eq-l-list*
unfolding *relation-of-def*
by *simp*
moreover have $a \in set\ l$
using *a-in-l*
by *simp*
ultimately show $(a, b) \in pl\text{-}\alpha\ l$
by *fastforce*
next
fix
$a :: 'a$ **and**
$b :: 'a$
assume $(a, b) \in pl\text{-}\alpha\ l$
thus $a \in set\ l$
using *Collect-mem-eq case-prod-eta in-rel-Collect-case-prod-eq*
is-less-preferred-than-l.simps
unfolding *pl-α-def*
by (*metis* (*no-types*))
next
fix
$a :: 'a$ **and**
$b :: 'a$
assume $(a, b) \in pl\text{-}\alpha\ l$
moreover have $\{(a', b').\ (a', b') \in set\ l \times set\ l \wedge a' \lesssim_l b'\} = pl\text{-}\alpha\ l$
using *rel-of-set-l-eq-l-list*
unfolding *relation-of-def*
by *simp*
ultimately show $b \in set\ l$
unfolding *is-less-preferred-than-l.simps*
by *blast*
next
fix
$a :: 'a$ **and**
$b :: 'a$
assume $(a, b) \in pl\text{-}\alpha\ l$
moreover have $\{(a', b').\ (a', b') \in set\ l \times set\ l \wedge a' \lesssim_l b'\} = pl\text{-}\alpha\ l$
using *rel-of-set-l-eq-l-list*
unfolding *relation-of-def*
by *simp*
ultimately have $a \lesssim_l b$
using *case-prodE mem-Collect-eq prod.inject*
by *blast*
thus (*if* $b \in set\ l$ *then index* $l\ b + 1$ *else* 0) $\leq$
(*if* $a \in set\ l$ *then index* $l\ a + 1$ *else* 0)
by *simp*
qed

```
show ∃ l'.
  p'!i = relation-of (λ a b. rank-l l' b ≤ rank-l l' a) (set l') ∧
  l' ∈ permutations-of-set A
proof
  have relation-of (λ a b. rank-l l b ≤ rank-l l a) (set l) = pl-α l
  proof (unfold relation-of-def rank-l.simps, safe)
    fix
      a :: 'a and
      b :: 'a
    assume
      idx-b-lte-idx-a: (if b ∈ set l then index l b + 1 else 0) ≤
                       (if a ∈ set l then index l a + 1 else 0) and
      a-in-l: a ∈ set l and
      b-in-l : b ∈ set l
    moreover have {(a', b'). (a', b') ∈ set l × set l ∧ a' ≲l b'} = pl-α l
      using rel-of-set-l-eq-l-list
      unfolding relation-of-def
      by simp
    moreover have a ∈ set l
      using a-in-l
      by simp
    ultimately show (a, b) ∈ pl-α l
      by fastforce
  next
    fix
      a :: 'a and
      b :: 'a
    assume (a, b) ∈ pl-α l
    thus a ∈ set l
      using Collect-mem-eq case-prod-eta in-rel-Collect-case-prod-eq
            is-less-preferred-than-l.simps
      unfolding pl-α-def
      by (metis (no-types))
  next
    fix
      a :: 'a and
      b :: 'a
    assume (a, b) ∈ pl-α l
    moreover have {(a', b'). (a', b') ∈ set l × set l ∧ a' ≲l b'} = pl-α l
      using rel-of-set-l-eq-l-list
      unfolding relation-of-def
      by simp
    ultimately show b ∈ set l
      using is-less-preferred-than-l.simps
      by blast
  next
    fix
      a :: 'a and
      b :: 'a
```

assume $(a, b) \in pl\text{-}\alpha\ l$
moreover have $\{(a', b').\ (a', b') \in set\ l \times set\ l \wedge a' \lesssim_l b'\} = pl\text{-}\alpha\ l$
using *rel-of-set-l-eq-l-list*
unfolding *relation-of-def*
by *simp*
ultimately have $a \lesssim_l b$
using *case-prodE mem-Collect-eq prod.inject*
by *blast*
thus $(if\ b \in set\ l\ then\ index\ l\ b + 1\ else\ 0) \leq$
$(if\ a \in set\ l\ then\ index\ l\ a + 1\ else\ 0)$
by *force*
qed
thus $p'!i = relation\text{-}of\ (\lambda\ a\ b.\ rank\text{-}l\ l\ b \leq rank\text{-}l\ l\ a)\ (set\ l) \wedge$
$l \in permutations\text{-}of\text{-}set\ A$
using *perm-l i-th-rel*
by *presburger*
qed
qed
let $?P = \lambda\ l\ a\ b.\ rank\text{-}l\ l\ b \leq rank\text{-}l\ l\ a$
have $\forall\ l.\ preorder\text{-}on\ (set\ l)\ (relation\text{-}of\ (?P\ l)\ (set\ l))$
unfolding *preorder-on-def refl-on-def Relation.trans-def relation-of-def*
by (*safe*, *linarith*)
moreover have $\forall\ l.\ antisym\ (relation\text{-}of\ (?P\ l)\ (set\ l))$
unfolding *antisym-def relation-of-def*
by *simp*
ultimately have $\forall\ l.\ partial\text{-}order\text{-}on\ (set\ l)\ (relation\text{-}of\ (?P\ l)\ (set\ l))$
unfolding *partial-order-on-def*
by *metis*
moreover have *set*: $\forall\ l.\ l \in permutations\text{-}of\text{-}set\ A \longrightarrow set\ l = A$
by (*simp add*: *permutations-of-setD*)
ultimately have $partial\text{-}order\text{-}on\ A\ (p'!i)$
using *relation-of*
by *fastforce*
moreover have $\forall\ l.\ total\text{-}on\ (set\ l)\ (relation\text{-}of\ (?P\ l)\ (set\ l))$
using *relation-of*
unfolding *total-on-def relation-of-def*
by *auto*
hence $total\text{-}on\ A\ (p'!i)$
using *relation-of set*
by *fastforce*
ultimately show $linear\text{-}order\text{-}on\ A\ (p'!i)$
unfolding *linear-order-on-def*
by *simp*
qed
next
fix
$r :: 'a\ Preference\text{-}Relation$ **and**
$q :: 'a\ Profile$ **and**
$p' :: 'a\ Profile$

assume
len-eq: *length* $p' = length\ (r\#q)$ **and**
fin-A: *finite* A **and**
prof-p': *profile* $A\ p'$
hence $\forall\ i < length\ (r\#q).\ linear\text{-}order\text{-}on\ A\ (p'!i)$
unfolding *profile-def*
by *simp*
hence $\forall\ i < length\ (r\#q).\ p'!i \in (pl\text{-}\alpha\ `\ permutations\text{-}of\text{-}set\ A)$
using *fin-A lin-ord-pl-α*
by *blast*
hence $p' \in listset\ (replicate\ (length\ (r\#q))\ (pl\text{-}\alpha\ `\ permutations\text{-}of\text{-}set\ A))$
using *all-ls-in-ls-set len-eq length-replicate nth-replicate fin-A*
by (*metis* (*no-types*, *lifting*))
thus $p' \in profile\text{-}permutations\ (length\ (r\#q))\ A$
using *fin-A*
unfolding *len-eq*
by *simp*
qed
qed

3.4.4 Soundness

lemma $\mathcal{R}$*-sound*:
fixes
$K :: 'a\ Consensus\text{-}Class$ **and**
$d :: 'a\ Election\ Distance$
shows *electoral-module* (*distance-*$\mathcal{R}$ $d\ K$)
unfolding *electoral-module-def*
by (*auto simp add*: *is-arg-min-def*)

3.4.5 Inference Rules

lemma *standard-distance-imp-equal-score*:
fixes
$d :: 'a\ Election\ Distance$ **and**
$K :: 'a\ Consensus\text{-}Class$ **and**
$A :: 'a\ set$ **and**
$p :: 'a\ Profile$ **and**
$a :: 'a$
assumes *std*: *standard d*
shows *score d K* $(A,\ p)\ a$ = *score-std d K* $(A,\ p)\ a$
proof −
have $\mathcal{K}_{\mathcal{E}}\ K\ a \cap$
$Pair\ A\ `\ \{p' :: 'a\ Profile.\ finite\text{-}profile\ A\ p' \wedge length\ p' = length\ p\} \subseteq$
$\mathcal{K}_{\mathcal{E}}\ K\ a$
by *simp*
hence *inf-lte-inf-int-pair*:
$Inf\ (d\ (A,\ p)\ `\ (\mathcal{K}_{\mathcal{E}}\ K\ a)) \leq$
$Inf\ (d\ (A,\ p)\ `\ (\mathcal{K}_{\mathcal{E}}\ K\ a \cap$
$Pair\ A\ `\ \{p' :: 'a\ Profile.\ finite\text{-}profile\ A\ p' \wedge length\ p' = length\ p\}))$

using *INF-superset-mono dual-order.refl*
by *blast*
moreover have *inf-gte-inf-int-pair*:
Inf (*d* (*A*, *p*) ‘ (*K*$_{\mathcal{E}}$ *K a*)) ≥
Inf (*d* (*A*, *p*) ‘ ((*K*$_{\mathcal{E}}$ *K a*) ∩
Pair A ‘ {*p*′ :: ′*a Profile*. *finite-profile A p*′ ∧ *length p*′ = *length p*}))
proof (*rule INF-greatest*)
let *?inf* =
Inf (*d* (*A*, *p*) ‘
(*K*$_{\mathcal{E}}$ *K a* ∩ *Pair A* ‘ {*p*′. *finite-profile A p*′ ∧ *length p*′ = *length p*}))
let *?compl* =
(*K*$_{\mathcal{E}}$ *K a*) −
(*K*$_{\mathcal{E}}$ *K a* ∩ *Pair A* ‘ {*p*′. *finite-profile A p*′ ∧ *length p*′ = *length p*})
fix *i* :: ′*a Election*
assume *i-in-K*$_{\mathcal{E}}$: *i* ∈ *K*$_{\mathcal{E}}$ *K a*
have *in-intersect*:
i ∈ (*K*$_{\mathcal{E}}$ *K a* ∩ *Pair A* ‘ {*p*′. *finite-profile A p*′ ∧ *length p*′ = *length p*}) ⟶
?inf ≤ *d* (*A*, *p*) *i*
using *INF-lower*
by (*metis* (*no-types*, *lifting*))
have *i* ∈ *?compl* ⟶
¬ (*A* = *fst i* ∧ *finite-profile A* (*snd i*) ∧ *length* (*snd i*) = *length p*)
by *fastforce*
moreover have *A* ≠ *fst i* ⟶ *d* (*A*, *p*) *i* = ∞
using *std*
unfolding *standard-def*
using *prod.collapse*
by *metis*
moreover have *length* (*snd i*) ≠ *length p* ⟶ *d* (*A*, *p*) *i* = ∞
using *std*
unfolding *standard-def*
using *prod.exhaust-sel*
by *metis*
moreover have
A = *fst i* ∧ *length* (*snd i*) = *length p* ⟶ *finite-profile A* (*snd i*)
using *i-in-K*$_{\mathcal{E}}$
unfolding *K*$_{\mathcal{E}}$*.simps*
by *auto*
ultimately have
i ∈ *?compl* ⟶
Inf (*d* (*A*, *p*) ‘
(*K*$_{\mathcal{E}}$ *K a* ∩ *Pair A* ‘
{*p*′. *finite-profile A p*′ ∧ *length p*′ = *length p*})) ≤
d (*A*, *p*) *i*
using *ereal-less-eq i-in-K*$_{\mathcal{E}}$
by (*metis* (*mono-tags*, *lifting*))
thus
Inf (*d* (*A*, *p*) ‘
(*K*$_{\mathcal{E}}$ *K a* ∩ *Pair A* ‘

```
        {p'. finite-profile A p' ∧ length p' = length p})) ≤
      d (A, p) i
    using in-intersect i-in-K_E
    by force
qed
have profile-perm-set:
  profile-permutations (length p) A =
    {p' :: 'a Profile. finite-profile A p' ∧ length p' = length p}
  using profile-permutation-set
  by blast
hence eq-intersect: K_E-std K a A (length p) =
        K_E K a ∩ Pair A ‘
         {p' :: 'a Profile. finite-profile A p' ∧ length p' = length p}
  by force
moreover have
  Inf (d (A, p) ‘ (K_E K a)) =
    Inf (d (A, p) ‘
      (K_E K a ∩
        Pair A ‘
          {p' :: 'a Profile. finite-profile A p' ∧ length p' = length p}))
  using inf-gte-inf-int-pair order-antisym inf-lte-inf-int-pair
  by (simp add: INF-superset-mono Orderings.order-eq-iff)
ultimately have inf-eq-inf-for-std-cons:
  Inf (d (A, p) ‘ (K_E K a)) =
    Inf (d (A, p) ‘ (K_E-std K a A (length p)))
  by simp
also have inf-eq-min-for-std-cons: ... = score-std d K (A, p) a
proof (cases K_E-std K a A (length p) = {})
  case True
  hence (d (A, p) ‘ (K_E-std K a A (length p))) = {}
    by simp
  hence Inf (d (A, p) ‘ (K_E-std K a A (length p))) = ∞
    using top-ereal-def
    by simp
  also have score-std d K (A, p) a = ∞
    using True score-std.simps
    unfolding Let-def
    by simp
  finally show ?thesis
    by simp
next
  case False
  hence d (A, p) ‘ (K_E-std K a A (length p)) ≠ {}
    by simp
  moreover have finite (K_E-std K a A (length p))
  proof −
    have finite (pl-α ‘ permutations-of-set A)
      by simp
    moreover have fin-A-imp-fin-all:
```

```
      ∀ n A. finite A ⟶ finite (profile-permutations n A)
      using listset-finiteness
      by force
    hence finite (profile-permutations (length p) A)
    proof (cases finite A)
      case True
      thus ?thesis
        using fin-A-imp-fin-all
        by metis
    next
      case False
      hence permutations-of-set A = {}
        using permutations-of-set-infinite
        by simp
      hence list-perm-A-empty: pl-α ‘ permutations-of-set A = {}
        by simp
      let ?p = replicate (length p) (pl-α ‘ permutations-of-set A)
      from list-perm-A-empty
      have ∀ i < length ?p. ?p!i = {}
        by simp
      hence finite (listset (replicate (length p) (pl-α ‘ permutations-of-set A)))
        using listset-finiteness finite.emptyI length-replicate nth-replicate
        by metis
      thus ?thesis
        by simp
    qed
    thus ?thesis
      by simp
  qed
  ultimately show ?thesis
    by (simp add: Lattices-Big.complete-linorder-class.Min-Inf)
 qed
 finally show score d K (A, p) a = score-std d K (A, p) a
  using inf-eq-inf-for-std-cons inf-eq-min-for-std-cons top-ereal-def
  by simp
qed

lemma anonymous-distance-and-consensus-imp-rule-anonymity:
 fixes
  d :: 'a Election Distance and
  K :: 'a Consensus-Class
 assumes
  d-anon: distance-anonymity d and
  K-anon: consensus-rule-anonymity K
 shows anonymity (distance-ℛ d K)
proof (unfold anonymity-def, safe)
 show electoral-module (distance-ℛ d K)
  by (simp add: ℛ-sound)
next
```

fix
  $A :: 'a\ set$ **and**
  $p :: 'a\ Profile$ **and**
  $q :: 'a\ Profile$
assume
  $profile\ A\ p$ **and**
  $profile\ A\ q$ **and**
  $p <^{\sim\sim}> q$
then obtain π **where**
  $perm_\pi$: $\pi\ permutes\ \{..< length\ p\}$ **and**
  pq: $permute\text{-}list\ \pi\ p = q$
  using *mset-eq-permutation*
  by *metis*
let $?\pi_l = permute\text{-}list\ \pi$
let $?\pi = \lambda\ n.\ (if\ n = length\ p\ then\ \pi\ else\ id)$
have *perm*: $\forall\ n.\ (?\pi\ n)\ permutes\ \{..< n\}$
  using $perm_\pi$
  by *simp*
let $?\pi_l' = \lambda\ l.\ permute\text{-}list\ (?\pi\ (length\ l))\ l$
let $?m = distance\text{-}\mathcal{R}\ d\ K$
let $?P = \lambda\ a\ A'\ p'.\ (A', p') \in \mathcal{K}_\mathcal{E}\ K\ a$
have $\forall\ a.\ \{(A', p') \mid A'\ p'.\ ?P\ a\ A'\ p'\} = \{(A', ?\pi_l'\ p') \mid A'\ p'.\ ?P\ a\ A'\ p'\}$
proof (*clarify*)
  fix $a :: 'a$
  have *apply-perm*: $\forall\ S\ x\ y.\ x <^{\sim\sim}> y \longrightarrow ?P\ a\ S\ x \longrightarrow ?P\ a\ S\ y$
  proof (*safe*)
    fix
      $S :: 'a\ set$ **and**
      $x :: 'a\ Profile$ **and**
      $y :: 'a\ Profile$
    assume
      perm: $x <^{\sim\sim}> y$ **and**
      fav-cons: $(S, x) \in \mathcal{K}_\mathcal{E}\ K\ a$
    hence *fin-S-x*: *finite-profile* $S\ x$
      by *simp*
    from *perm*
    have *fin-S-y*: *finite-profile* $S\ y$
      unfolding *profile-def*
      using *fin-S-x nth-mem perm-set-eq profile-set*
      by *metis*
    moreover have $(consensus\text{-}\mathcal{K}\ K)\ (S, x) \wedge elect\ (rule\text{-}\mathcal{K}\ K)\ S\ x = \{a\}$
      using *perm fav-cons*
      by *simp*
    hence $(consensus\text{-}\mathcal{K}\ K)\ (S, y) \wedge elect\ (rule\text{-}\mathcal{K}\ K)\ S\ y = \{a\}$
      using *K-anon*
      unfolding *consensus-rule-anonymity-def anonymity-def*
      using *perm fin-S-x fin-S-y calculation*
      by (*metis* (*no-types*))
    ultimately show $(S, y) \in \mathcal{K}_\mathcal{E}\ K\ a$

```
    by simp
qed
show {(A', p') | A' p'. ?P a A' p'} =
       {(A', ?π_l' p') | A' p'. ?P a A' p'} (is ?X = ?Y)
proof
  show ?X ⊆ ?Y
  proof
    fix E :: 'a Election
    let
      ?A = alts-ℰ E and
      ?p = prof-ℰ E
    assume consensus-election-E: E ∈ {(A', p') | A' p'. ?P a A' p'}
    hence consens-elect-E-inst: ?P a ?A ?p
      by simp
    show E ∈ {(A', ?π_l' p') | A' p'. ?P a A' p'}
    proof (cases length ?p = length p)
      case True
      hence permute-list (inv π) ?p <~~> ?p
        using perm_π
        by (simp add: permutes-inv)
      hence ?P a ?A (permute-list (inv π) ?p)
        using consens-elect-E-inst apply-perm
        by presburger
      moreover have length (permute-list (inv π) ?p) = length p
        using True
        by simp
      ultimately have
        (?A, ?π_l (permute-list (inv π) ?p)) ∈
           {(A', ?π_l p') | A' p'. length p' = length p ∧ ?P a A' p'}
        by auto
      also have permute-list π (permute-list (inv π) ?p) = ?p
        using True permute-list-compose permute-list-id perm_π surj-iff
              permutes-inv permutes-inv-inv permutes-surj
        by metis
      finally show ?thesis
        by auto
    next
      case False
      thus ?thesis
        using consensus-election-E
        by fastforce
    qed
  qed
next
  show ?Y ⊆ ?X
  proof
    fix E :: 'a Election
    let
      ?A = alts-ℰ E and
```

```
      ?r = prof-𝓔 E
    assume consensus-elect-permut-E: E ∈ {(A', ?π_l' p') | A' p'. ?P a A' p'}
    hence ∃ p'. ?r = ?π_l' p' ∧ ?P a ?A p'
      by auto
    then obtain p' where
      rp': ?r = ?π_l' p' and
      consens-elect-inst: ?P a ?A p'
      by metis
    show E ∈ {(A', p') | A' p'. ?P a A' p'}
    proof (cases length p' = length p)
      case True
      hence length ?r = length p
        using rp'
        by simp
      moreover have ?r <~~> p'
        using perm_π rp'
        by simp
      hence ?P a ?A ?r
        unfolding rp'
        using consens-elect-inst apply-perm
        by presburger
      ultimately show E ∈ {(A', p') | A' p'. ?P a A' p'}
        by simp
    next
      case False
      thus ?thesis
        using consensus-elect-permut-E rp'
        by fastforce
    qed
  qed
qed
hence ∀ a ∈ A. d (A, q) ' {(A', p') | A' p'. ?P a A' p'}
        = d (A, q) ' {(A', ?π_l' p') | A' p'. ?P a A' p'}
  by (metis (no-types, lifting))
hence ∀ a ∈ A. {d (A, q) (A', p') | A' p'. ?P a A' p'}
        = {d (A, q) (A', ?π_l' p') | A' p'. ?P a A' p'}
  by blast
moreover from d-anon
have ∀ a ∈ A. {d (A, p) (A', p') | A' p'. ?P a A' p'}
        = {d (A, ?π_l' p) (A', ?π_l' p') | A' p'. ?P a A' p'}
proof (clarify)
  fix a :: 'a
  have ?π_l' = (λ p. permute-list (?π (length p)) p)
    by simp
  from d-anon
  have ∀ A' p' π. (∀ n. (π n) permutes {..< n}) ⟶
        d (A, p) (A', p') =
          d (A, permute-list (π (length p)) p)
```

```
        (A', permute-list (π (length p')) p')
    unfolding distance-anonymity-def
    by blast
  thus {d (A, p) (A', p') | A' p'. ?P a A' p'} =
       {d (A, ?π_l' p) (A', ?π_l' p') | A' p'. ?P a A' p'}
    using perm
    unfolding distance-anonymity-def
    by simp
 qed
 hence ∀ a ∈ A. {d (A, p) (A', p') | A' p'. ?P a A' p'} =
       {d (A, q) (A', ?π_l' p') | A' p'. ?P a A' p'}
  using pq
  by simp
 ultimately have
  ∀ a ∈ A. {d (A, q) (A', p') | A' p'. (A', p') ∈ K_ℰ K a} =
           {d (A, p) (A', p') | A' p'. (A', p') ∈ K_ℰ K a}
  by simp
 hence ∀ a ∈ A. d (A, q) `K_ℰ K a = d (A, p) `K_ℰ K a
  by fast
 hence ∀ a ∈ A. score d K (A, p) a = score d K (A, q) a
  by simp
 thus distance-R d K A p = distance-R d K A q
  using is-arg-min-equal[of A score d K (A, p) score d K (A, q)]
  by auto
qed

end
```

3.5 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.

3.5.1 Common Rationalizations

```
fun swap-R :: ('a Election ⇒ bool) × 'a Electoral-Module ⇒
              'a Electoral-Module where
 swap-R A p = distance-R (votewise-distance swap l-one) A p
```

3.5.2 Theorems

lemma *swap-standard*: *standard* (*votewise-distance swap l-one*)
proof (*unfold standard-def*, *clarify*)
fix
C :: *'a set* **and**
B :: *'a set* **and**
p :: *'a Profile* **and**
q :: *'a Profile*
assume *len-p-neq-len-q-or-C-neq-B*: *length p* ≠ *length q* ∨ *C* ≠ *B*
thus *votewise-distance swap l-one* (*C*, *p*) (*B*, *q*) = ∞
proof (*cases length p* ≠ *length q* ∨ *length p* = *0*, *simp*)
case *False*
hence *C-neq-B*: *C* ≠ *B*
using *len-p-neq-len-q-or-C-neq-B*
by *simp*
from *False*
have (*map2* (λ *x y*. *swap* (*C*, *x*) (*B*, *y*)) *p q*)!*0* = *swap* (*C*, (*p*!*0*)) (*B*, (*q*!*0*))
using *case-prod-conv length-zip min.idem nth-map nth-zip zero-less-iff-neq-zero*
by (*metis* (*no-types*, *lifting*))
also have ... = ∞
using *C-neq-B*
by *simp*
finally have (*map2* (λ *x y*. *swap* (*C*, *x*) (*B*, *y*)) *p q*)!*0* = ∞
by *simp*
have *len-gt-zero*: *0* < *length* (*map2* (λ *x y*. *swap* (*C*, *x*) (*B*, *y*)) *p q*)
using *False*
by *force*
moreover have
(∑ *i*::*nat* < *min* (*length p*) (*length q*). *ereal-of-enat* (∞)) = ∞
using *finite-lessThan sum-Pinfty lessThan-iff min.idem*
False not-gr-zero of-nat-eq-enat ereal-of-enat-simps
by *metis*
ultimately have *l-one* (*map2* (λ *x y*. *swap* (*C*, *x*) (*B*, *y*)) *p q*) = ∞
using *C-neq-B*
by *simp*
thus *?thesis*
using *False*
by *simp*
qed
qed

3.5.3 Equivalence Lemmas

lemma *equal-score-swap*:
score (*votewise-distance swap l-one*) =
score-std (*votewise-distance swap l-one*)
using *standard-distance-imp-equal-score swap-standard*
by *fast*

lemma *swap-$\mathcal{R}$-code*[*code*]: *swap-$\mathcal{R}$* = *distance-$\mathcal{R}$-std* (*votewise-distance swap l-one*)
proof −
from *equal-score-swap*
have ∀ *K E a*. *score* (*votewise-distance swap l-one*) *K E a* =
score-std (*votewise-distance swap l-one*) *K E a*
by *metis*
hence ∀ *K A p*. $\mathcal{R}_{\mathcal{W}}$ (*votewise-distance swap l-one*) *K A p* =
$\mathcal{R}_{\mathcal{W}}$*-std* (*votewise-distance swap l-one*) *K A p*
by (*simp add*: *equal-score-swap*)
hence ∀ *K A p*. *distance-$\mathcal{R}$* (*votewise-distance swap l-one*) *K A p* =
distance-$\mathcal{R}$-std (*votewise-distance swap l-one*) *K A p*
by *fastforce*
thus *?thesis*
unfolding *swap-$\mathcal{R}$.simps*
by *blast*
qed

end

3.6 Drop Module

theory *Drop-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 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.

3.6.1 Definition

fun *drop-module* :: *nat* ⇒ *'a Preference-Relation* ⇒ *'a Electoral-Module* **where**
drop-module n r A p =
({},
{*a* ∈ *A*. *rank* (*limit A r*) *a* ≤ *n*},
{*a* ∈ *A*. *rank* (*limit A r*) *a* > *n*})

3.6.2 Soundness

theorem *drop-mod-sound*[*simp*]:
fixes
r :: *'a Preference-Relation* **and**

```
    n :: nat
  shows electoral-module (drop-module n r)
proof (unfold electoral-module-def, safe)
  fix
    A :: 'a set and
    p :: 'a Profile
  let ?mod = drop-module n r
  have ∀ a ∈ A. a ∈ {x ∈ A. rank (limit A r) x ≤ n} ∨
                  a ∈ {x ∈ A. rank (limit A r) x > n}
    by auto
  hence {a ∈ A. rank (limit A r) a ≤ n} ∪ {a ∈ A. rank (limit A r) a > n} = A
    by blast
  hence set-partition: set-equals-partition A (drop-module n r A p)
    by simp
  have ∀ a ∈ A.
         ¬ (a ∈ {x ∈ A. rank (limit A r) x ≤ n} ∧
            a ∈ {x ∈ A. rank (limit A r) x > n})
    by simp
  hence {a ∈ A. rank (limit A r) a ≤ n} ∩ {a ∈ A. rank (limit A r) a > n} = {}
    by blast
  thus well-formed A (?mod A p)
    using set-partition
    by simp
qed
```

3.6.3 Non-Electing

The drop module is non-electing.

```
theorem drop-mod-non-electing[simp]:
  fixes
    r :: 'a Preference-Relation and
    n :: nat
  shows non-electing (drop-module n r)
  unfolding non-electing-def
  by simp
```

3.6.4 Properties

The drop module is strictly defer-monotone.

```
theorem drop-mod-def-lift-inv[simp]:
  fixes
    r :: 'a Preference-Relation and
    n :: nat
  shows defer-lift-invariance (drop-module n r)
  unfolding defer-lift-invariance-def
  by simp

end
```

3.7 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.

3.7.1 Definition

fun *pass-module* :: *nat* ⇒ *'a Preference-Relation* ⇒ *'a Electoral-Module* **where**
pass-module n r A p =
({},
{*a* ∈ *A*. *rank* (*limit A r*) *a* > *n*},
{*a* ∈ *A*. *rank* (*limit A r*) *a* ≤ *n*})

3.7.2 Soundness

theorem *pass-mod-sound*[*simp*]:
fixes
r :: *'a Preference-Relation* **and**
n :: *nat*
shows *electoral-module* (*pass-module n r*)
proof (*unfold electoral-module-def*, *safe*)
fix
A :: *'a set* **and**
p :: *'a Profile*
let *?mod* = *pass-module n r*
have ∀ *a* ∈ *A*. *a* ∈ {*x* ∈ *A*. *rank* (*limit A r*) *x* > *n*} ∨
a ∈ {*x* ∈ *A*. *rank* (*limit A r*) *x* ≤ *n*}
using *CollectI not-less*
by *metis*
hence {*a* ∈ *A*. *rank* (*limit A r*) *a* > *n*} ∪ {*a* ∈ *A*. *rank* (*limit A r*) *a* ≤ *n*} = *A*
by *blast*
hence *set-equals-partition A* (*pass-module n r A p*)
by *simp*
moreover have
∀ *a* ∈ *A*.
¬ (*a* ∈ {*x* ∈ *A*. *rank* (*limit A r*) *x* > *n*} ∧

$a \in \{x \in A.\ rank\ (limit\ A\ r)\ x \leq n\}$)
by *simp*
hence $\{a \in A.\ rank\ (limit\ A\ r)\ a > n\} \cap \{a \in A.\ rank\ (limit\ A\ r)\ a \leq n\} = \{\}$
by *blast*
ultimately show *well-formed A* (*?mod A p*)
by *simp*
qed

3.7.3 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**
g0-n: $n > 0$
shows *non-blocking* (*pass-module n r*)
proof (*unfold non-blocking-def*, *safe*)
show *electoral-module* (*pass-module n r*)
by *simp*
next
fix
$A :: 'a$ *set* **and**
$p :: 'a$ *Profile* **and**
$a :: 'a$
assume
fin-A: *finite A* **and**
rej-pass-A: *reject* (*pass-module n r*) *A p* $= A$ **and**
a-in-A: $a \in A$
moreover have *linear-order-on A* (*limit A r*)
using *limit-presv-lin-ord order top-greatest*
by *metis*
moreover have
$\exists\ b \in A.\ above\ (limit\ A\ r)\ b = \{b\}$
$\wedge\ (\forall\ c \in A.\ above\ (limit\ A\ r)\ c = \{c\} \longrightarrow c = b)$
using *calculation above-one fin-A*
by *blast*
moreover have $\{b \in A.\ rank\ (limit\ A\ r)\ b > n\} \neq A$
using *Suc-leI g0-n leD mem-Collect-eq above-rank calculation*
unfolding *One-nat-def*
by (*metis* (*no-types*, *lifting*))
ultimately have *reject* (*pass-module n r*) *A p* $\neq A$
by *simp*
thus $a \in \{\}$
using *rej-pass-A*
by *simp*
qed

3.7.4 Non-Electing

The pass module is non-electing.

theorem *pass-mod-non-electing*[*simp*]:
fixes
r :: *'a Preference-Relation* **and**
n :: *nat*
assumes *linear-order r*
shows *non-electing* (*pass-module n r*)
unfolding *non-electing-def*
using *assms*
by *simp*

3.7.5 Properties

The pass module is strictly defer-monotone.

theorem *pass-mod-dl-inv*[*simp*]:
fixes
r :: *'a Preference-Relation* **and**
n :: *nat*
assumes *linear-order r*
shows *defer-lift-invariance* (*pass-module n r*)
unfolding *defer-lift-invariance-def*
using *assms*
by *simp*

theorem *pass-zero-mod-def-zero*[*simp*]:
fixes *r* :: *'a Preference-Relation*
assumes *linear-order r*
shows *defers 0* (*pass-module 0 r*)
proof (*unfold defers-def*, *safe*)
show *electoral-module* (*pass-module 0 r*)
using *pass-mod-sound assms*
by *simp*
next
fix
A :: *'a set* **and**
p :: *'a Profile*
assume
card-pos: $0 \leq$ *card A* **and**
finite-A: *finite A* **and**
prof-A: *profile A p*
have *linear-order-on A* (*limit A r*)
using *assms limit-presv-lin-ord*
by *blast*
hence *limit-is-connex*: *connex A* (*limit A r*)
using *lin-ord-imp-connex*
by *simp*

```
  have ∀ n. (n::nat) ≤ 0 ⟶ n = 0
    by blast
  hence ∀ a A'. a ∈ A' ∧ a ∈ A ⟶ connex A' (limit A r) ⟶
        ¬ rank (limit A r) a ≤ 0
    using above-connex above-presv-limit card-eq-0-iff equals0D finite-A
        assms rev-finite-subset
    unfolding rank.simps
    by (metis (no-types))
  hence {a ∈ A. rank (limit A r) a ≤ 0} = {}
    using limit-is-connex
    by simp
  hence card {a ∈ A. rank (limit A r) a ≤ 0} = 0
    using card.empty
    by metis
  thus card (defer (pass-module 0 r) A p) = 0
    by simp
qed
```

For any natural number n and any linear order, the according pass module defers n alternatives (if there are n alternatives). NOTE: The induction proof is still missing. The following are the proofs for n=1 and n=2.

```
theorem pass-one-mod-def-one[simp]:
  fixes r :: 'a Preference-Relation
  assumes linear-order r
  shows defers 1 (pass-module 1 r)
proof (unfold defers-def, safe)
  show electoral-module (pass-module 1 r)
    using pass-mod-sound assms
    by simp
next
  fix
    A :: 'a set and
    p :: 'a Profile
  assume
    card-pos: 1 ≤ card A and
    finite-A: finite A and
    prof-A: profile A p
  show card (defer (pass-module 1 r) A p) = 1
  proof −
    have A ≠ {}
      using card-pos
      by auto
    moreover have lin-ord-on-A: linear-order-on A (limit A r)
      using assms limit-presv-lin-ord
      by blast
    ultimately have winner-exists:
      ∃ a ∈ A. above (limit A r) a = {a} ∧
          (∀ b ∈ A. above (limit A r) b = {b} ⟶ b = a)
      using finite-A
```

by (*simp add*: *above-one*)
then obtain *w* **where** *w-unique-top*:
above (*limit A r*) *w* = {*w*} ∧
(∀ *a* ∈ *A*. *above* (*limit A r*) *a* = {*a*} ⟶ *a* = *w*)
using *above-one*
by *auto*
hence {*a* ∈ *A*. *rank* (*limit A r*) *a* ≤ *1*} = {*w*}
proof
assume
w-top: *above* (*limit A r*) *w* = {*w*} **and**
w-unique: ∀ *a* ∈ *A*. *above* (*limit A r*) *a* = {*a*} ⟶ *a* = *w*
have *rank* (*limit A r*) *w* ≤ *1*
using *w-top*
by *auto*
hence {*w*} ⊆ {*a* ∈ *A*. *rank* (*limit A r*) *a* ≤ *1*}
using *winner-exists w-unique-top*
by *blast*
moreover have {*a* ∈ *A*. *rank* (*limit A r*) *a* ≤ *1*} ⊆ {*w*}
proof
fix *a* :: *'a*
assume *a-in-winner-set*: *a* ∈ {*b* ∈ *A*. *rank* (*limit A r*) *b* ≤ *1*}
hence *a-in-A*: *a* ∈ *A*
by *auto*
hence *connex-limit*: *connex A* (*limit A r*)
using *lin-ord-imp-connex lin-ord-on-A*
by *simp*
hence *let q* = *limit A r in a* ≼$_q$ *a*
using *connex-limit above-connex pref-imp-in-above a-in-A*
by *metis*
hence (*a*, *a*) ∈ *limit A r*
by *simp*
hence *a-above-a*: *a* ∈ *above* (*limit A r*) *a*
unfolding *above-def*
by *simp*
have *above* (*limit A r*) *a* ⊆ *A*
using *above-presv-limit assms*
by *fastforce*
hence *above-finite*: *finite* (*above* (*limit A r*) *a*)
using *finite-A finite-subset*
by *simp*
have *rank* (*limit A r*) *a* ≤ *1*
using *a-in-winner-set*
by *simp*
moreover have *rank* (*limit A r*) *a* ≥ *1*
using *Suc-leI above-finite card-eq-0-iff equals0D neq0-conv a-above-a*
unfolding *rank.simps One-nat-def*
by *metis*
ultimately have *rank* (*limit A r*) *a* = *1*
by *simp*

```
      hence {a} = above (limit A r) a
        using a-above-a lin-ord-on-A rank-one-imp-above-one
        by metis
      hence a = w
        using w-unique
        by (simp add: a-in-A)
      thus a ∈ {w}
        by simp
    qed
    ultimately have {w} = {a ∈ A. rank (limit A r) a ≤ 1}
      by auto
    thus ?thesis
      by simp
  qed
  thus card (defer (pass-module 1 r) A p) = 1
    by simp
 qed
qed

theorem pass-two-mod-def-two:
 fixes r :: 'a Preference-Relation
 assumes linear-order r
 shows defers 2 (pass-module 2 r)
proof (unfold defers-def, safe)
 show electoral-module (pass-module 2 r)
   using assms
   by simp
next
 fix
   A :: 'a set and
   p :: 'a Profile
 assume
   min-card-two: 2 ≤ card A and
   fin-A: finite A and
   prof-A: profile A p
 from min-card-two
 have not-empty-A: A ≠ {}
   by auto
 moreover have limit-A-order: linear-order-on A (limit A r)
   using limit-presv-lin-ord assms
   by auto
 ultimately obtain a where
   above (limit A r) a = {a}
   using above-one min-card-two fin-A prof-A
   by blast
 hence ∀ b ∈ A. let q = limit A r in (b ≼_q a)
   using limit-A-order pref-imp-in-above empty-iff lin-ord-imp-connex
       insert-iff insert-subset above-presv-limit assms
   unfolding connex-def
```

by *metis*
hence *a-best*: $\forall\ b \in A.\ (b,\ a) \in limit\ A\ r$
by *simp*
hence *a-above*: $\forall\ b \in A.\ a \in above\ (limit\ A\ r)\ b$
unfolding *above-def*
by *simp*
hence $a \in \{a \in A.\ rank\ (limit\ A\ r)\ a \leq 2\}$
using *CollectI not-empty-A empty-iff fin-A insert-iff limit-A-order above-one above-rank one-le-numeral*
by (*metis* (*no-types*, *lifting*))
hence *a-in-defer*: $a \in defer\ (pass\text{-}module\ 2\ r)\ A\ p$
by *simp*
have *finite* $(A - \{a\})$
using *fin-A*
by *simp*
moreover have *A-not-only-a*: $A - \{a\} \neq \{\}$
using *Diff-empty Diff-idemp Diff-insert0 not-empty-A insert-Diff finite.emptyI card.insert-remove card.empty min-card-two Suc-n-not-le-n numeral-2-eq-2*
by *metis*
moreover have *limit-A-without-a-order*:
$linear\text{-}order\text{-}on\ (A - \{a\})\ (limit\ (A - \{a\})\ r)$
using *limit-presv-lin-ord assms top-greatest*
by *blast*
ultimately obtain b **where**
b: $above\ (limit\ (A - \{a\})\ r)\ b = \{b\}$
using *above-one*
by *metis*
hence $\forall\ c \in A - \{a\}.\ let\ q = limit\ (A - \{a\})\ r\ in\ (c \preceq_q b)$
using *limit-A-without-a-order pref-imp-in-above empty-iff lin-ord-imp-connex insert-iff insert-subset above-presv-limit assms*
unfolding *connex-def*
by *metis*
hence *b-in-limit*: $\forall\ c \in A - \{a\}.\ (c,\ b) \in limit\ (A - \{a\})\ r$
by *simp*
hence *b-best*: $\forall\ c \in A - \{a\}.\ (c,\ b) \in limit\ A\ r$
by *auto*
hence $\forall\ c \in A - \{a,\ b\}.\ c \notin above\ (limit\ A\ r)\ b$
using *b Diff-iff Diff-insert2 above-presv-limit insert-subset assms limit-presv-above limit-rel-presv-above*
by *metis*
moreover have *above-subset*: $above\ (limit\ A\ r)\ b \subseteq A$
using *above-presv-limit assms*
by *metis*
moreover have *b-above-b*: $b \in above\ (limit\ A\ r)\ b$
using *b b-best above-presv-limit mem-Collect-eq assms insert-subset*
unfolding *above-def*
by *metis*
ultimately have *above-b-eq-ab*: $above\ (limit\ A\ r)\ b = \{a,\ b\}$
using *a-above*

by *auto*
hence *card-above-b-eq-two*: *rank* (*limit A r*) *b* = *2*
using *A-not-only-a b-in-limit*
by *auto*
hence *b-in-defer*: *b* ∈ *defer* (*pass-module 2 r*) *A p*
using *b-above-b above-subset*
by *auto*
have *b-above*: ∀ *c* ∈ *A* − {*a*}. *b* ∈ *above* (*limit A r*) *c*
using *b-best mem-Collect-eq*
unfolding *above-def*
by *metis*
have *connex A* (*limit A r*)
using *limit-A-order lin-ord-imp-connex*
by *auto*
hence ∀ *c* ∈ *A*. *c* ∈ *above* (*limit A r*) *c*
by (*simp add*: *above-connex*)
hence ∀ *c* ∈ *A* − {*a*, *b*}. {*a*, *b*, *c*} ⊆ *above* (*limit A r*) *c*
using *a-above b-above*
by *auto*
moreover have ∀ *c* ∈ *A* − {*a*, *b*}. *card* {*a*, *b*, *c*} = *3*
using *DiffE Suc-1 above-b-eq-ab card-above-b-eq-two above-subset fin-A*
card-insert-disjoint finite-subset insert-commute numeral-3-eq-3
unfolding *One-nat-def rank.simps*
by *metis*
ultimately have ∀ *c* ∈ *A* − {*a*, *b*}. *rank* (*limit A r*) *c* ≥ *3*
using *card-mono fin-A finite-subset above-presv-limit assms*
unfolding *rank.simps*
by *metis*
hence ∀ *c* ∈ *A* − {*a*, *b*}. *rank* (*limit A r*) *c* > *2*
using *Suc-le-eq Suc-1 numeral-3-eq-3*
unfolding *One-nat-def*
by *metis*
hence ∀ *c* ∈ *A* − {*a*, *b*}. *c* ∉ *defer* (*pass-module 2 r*) *A p*
by (*simp add*: *not-le*)
moreover have *defer* (*pass-module 2 r*) *A p* ⊆ *A*
by *auto*
ultimately have *defer* (*pass-module 2 r*) *A p* ⊆ {*a*, *b*}
by *blast*
hence *defer* (*pass-module 2 r*) *A p* = {*a*, *b*}
using *a-in-defer b-in-defer*
by *fastforce*
thus *card* (*defer* (*pass-module 2 r*) *A p*) = *2*
using *above-b-eq-ab card-above-b-eq-two*
unfolding *rank.simps*
by *presburger*
qed

end

3.8 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.

3.8.1 Definition

fun *elect-module* :: *'a Electoral-Module* **where**
  elect-module A p = (*A*, {}, {})

3.8.2 Soundness

theorem *elect-mod-sound*[*simp*]: *electoral-module elect-module*
  unfolding *electoral-module-def*
  by *simp*

3.8.3 Electing

theorem *elect-mod-electing*[*simp*]: *electing elect-module*
  unfolding *electing-def*
  by *simp*

end

3.9 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.

3.9.1 Definition

fun *plurality-score* :: *'a Evaluation-Function* **where**
plurality-score x A p = *win-count p x*

fun *plurality* :: *'a Electoral-Module* **where**
plurality A p = *max-eliminator plurality-score A p*

fun *plurality'* :: *'a Electoral-Module* **where**
plurality' A p =
({},
{*a* ∈ *A*. ∃ *x* ∈ *A*. *win-count p x* > *win-count p a*},
{*a* ∈ *A*. ∀ *x* ∈ *A*. *win-count p x* ≤ *win-count p a*})

lemma *plurality-mod-elim-equiv*:
fixes
A :: *'a set* **and**
p :: *'a Profile*
assumes
non-empty-A: *A* ≠ {} **and**
fin-prof-A: *finite-profile A p*
shows *plurality A p* = *plurality' A p*
proof (*unfold plurality.simps plurality'.simps plurality-score.simps*, *standard*)
show *elect* (*max-eliminator* (λ *x A p*. *win-count p x*)) *A p* =
elect-r ({},
{*a* ∈ *A*. ∃ *b* ∈ *A*. *win-count p a* < *win-count p b*},
{*a* ∈ *A*. ∀ *b* ∈ *A*. *win-count p b* ≤ *win-count p a*})
using *max-elim-non-electing fin-prof-A*
by *simp*
next
have *rej-eq*:
reject (*max-eliminator* (λ *b A p*. *win-count p b*)) *A p* =
{*a* ∈ *A*. ∃ *b* ∈ *A*. *win-count p a* < *win-count p b*}
proof (*simp del*: *win-count.simps*, *safe*)
fix
a :: *'a* **and**
b :: *'a*
assume
b ∈ *A* **and**
win-count p a < *Max* {*win-count p a'* | *a'*. *a'* ∈ *A*} **and**
¬ *win-count p b* < *Max* {*win-count p a'* | *a'*. *a'* ∈ *A*}
thus ∃ *b* ∈ *A*. *win-count p a* < *win-count p b*
using *dual-order.strict-trans1 not-le-imp-less*
by *blast*
next
fix
a :: *'a* **and**
b :: *'a*
assume
b-in-A: *b* ∈ *A* **and**

wc-a-lt-wc-b: *win-count p a* < *win-count p b*
moreover have ∀ *t*. *t b* ≤ *Max* {*n*. ∃ *a'*. (*n*::*nat*) = *t a'* ∧ *a'* ∈ *A*}
using *fin-prof-A b-in-A*
by (*simp add*: *score-bounded*)
ultimately show *win-count p a* < *Max* {*win-count p a'* | *a'*. *a'* ∈ *A*}
using *dual-order.strict-trans1*
by *blast*
next
assume {*a* ∈ *A*. *win-count p a* < *Max* {*win-count p b* | *b*. *b* ∈ *A*}} = *A*
hence *A* = {}
using *max-score-contained*[**where** *A*=*A* **and** *e*=(λ *a*. *win-count p a*)]
fin-prof-A nat-less-le
by *blast*
thus *False*
using *non-empty-A*
by *simp*
qed
have *defer* (*max-eliminator* (λ *x A p*. *win-count p x*)) *A p* =
{*a* ∈ *A*. ∀ *a'* ∈ *A*. *win-count p a'* ≤ *win-count p a*}
proof (*auto simp del*: *win-count.simps*)
fix
a :: *'a* **and**
b :: *'a*
assume
a ∈ *A* **and**
b ∈ *A* **and**
¬ *win-count p a* < *Max* {*win-count p a'* | *a'*. *a'* ∈ *A*}
moreover from *this*
have *win-count p a* = *Max* {*win-count p a'* | *a'*. *a'* ∈ *A*}
using *score-bounded*[**where** *A*=*A* **and** *e* =(λ *a'*. *win-count p a'*)] *fin-prof-A*
order-le-imp-less-or-eq
by *blast*
ultimately show *win-count p b* ≤ *win-count p a*
using *score-bounded*[**where** *A*= *A* **and** *e* = (λ *x*. *win-count p x*)] *fin-prof-A*
by *presburger*
next
fix
a :: *'a* **and**
b :: *'a*
assume {*a'* ∈ *A*. *win-count p a'* < *Max* {*win-count p b'* | *b'*. *b'* ∈ *A*}} = *A*
hence *A* = {}
using *max-score-contained*[**where** *A*= *A* **and** *e* = (λ *x*. *win-count p x*)]
fin-prof-A nat-less-le
by *auto*
thus *win-count p a* ≤ *win-count p b*
using *non-empty-A*
by *simp*
qed
thus *snd* (*max-eliminator* (λ *b A p*. *win-count p b*) *A p*) =

snd ({},
{$a \in A.\ \exists\ b \in A.$ *win-count p a* $<$ *win-count p b*},
{$a \in A.\ \forall\ b \in A.$ *win-count p b* $\leq$ *win-count p a*})
using *rej-eq prod.collapse snd-conv*
by *metis*
qed

3.9.2 Soundness

theorem *plurality-sound*[*simp*]: *electoral-module plurality*
unfolding *plurality.simps*
using *max-elim-sound*
by *metis*

theorem *plurality′-sound*[*simp*]: *electoral-module plurality′*
proof (*unfold electoral-module-def*, *safe*)
fix
A :: *'a set* **and**
p :: *'a Profile*
have *disjoint3* (
{},
{$a \in A.\ \exists\ a' \in A.$ *win-count p a* $<$ *win-count p a′*},
{$a \in A.\ \forall\ a' \in A.$ *win-count p a′* $\leq$ *win-count p a*})
by *auto*
moreover have
{$a \in A.\ \exists\ x \in A.$ *win-count p a* $<$ *win-count p x*} $\cup$
{$a \in A.\ \forall\ x \in A.$ *win-count p x* $\leq$ *win-count p a*} $= A$
using *not-le-imp-less*
by *auto*
ultimately show *well-formed A* (*plurality′ A p*)
by *simp*
qed

3.9.3 Non-Blocking

The plurality module is non-blocking.

theorem *plurality-mod-non-blocking*[*simp*]: *non-blocking plurality*
unfolding *plurality.simps*
using *max-elim-non-blocking*
by *metis*

3.9.4 Non-Electing

The plurality module is non-electing.

theorem *plurality-non-electing*[*simp*]: *non-electing plurality*
using *max-elim-non-electing*
unfolding *plurality.simps non-electing-def*
by *metis*

theorem *plurality'-non-electing*[*simp*]: *non-electing plurality'*
by (*simp add*: *non-electing-def*)

3.9.5 Property

lemma *plurality-def-inv-mono-alts*:
fixes
A :: *'a set* **and**
p :: *'a Profile* **and**
q :: *'a Profile* **and**
a :: *'a*
assumes
defer-a: *a* ∈ *defer plurality A p* **and**
lift-a: *lifted A p q a*
shows *defer plurality A q* = *defer plurality A p* ∨ *defer plurality A q* = {*a*}
proof −
have *set-disj*: ∀ *b c*. (*b*::*'a*) ∉ {*c*} ∨ *b* = *c*
by *force*
have *lifted-winner*:
∀ *b* ∈ *A*.
∀ *i*::*nat*. *i* < *length p* ⟶
above (*p*!*i*) *b* = {*b*} ⟶ *above* (*q*!*i*) *b* = {*b*} ∨ *above* (*q*!*i*) *a* = {*a*}
using *lift-a lifted-above-winner-alts*
unfolding *Profile.lifted-def*
by (*metis* (*no-types*, *lifting*))
hence ∀ *i*::*nat*. *i* < *length p* ⟶ *above* (*p*!*i*) *a* = {*a*} ⟶ *above* (*q*!*i*) *a* = {*a*}
using *defer-a lift-a*
unfolding *Profile.lifted-def*
by *metis*
hence *a-win-subset*:
{*i*::*nat*. *i* < *length p* ∧ *above* (*p*!*i*) *a* = {*a*}} ⊆
{*i*::*nat*. *i* < *length p* ∧ *above* (*q*!*i*) *a* = {*a*}}
by *blast*
moreover have *sizes*: *length p* = *length q*
using *lift-a*
unfolding *Profile.lifted-def*
by *metis*
ultimately have *win-count-a*: *win-count p a* ≤ *win-count q a*
by (*simp add*: *card-mono*)
have *fin-A*: *finite A*
using *lift-a*
unfolding *Profile.lifted-def*
by *metis*
hence
∀ *b* ∈ *A* − {*a*}.
∀ *i*::*nat*. *i* < *length p* ⟶ *above* (*q*!*i*) *a* = {*a*} ⟶ *above* (*q*!*i*) *b* ≠ {*b*}
using *DiffE above-one-eq lift-a insertCI insert-absorb insert-not-empty sizes*
unfolding *Profile.lifted-def profile-def*

by *metis*
with *lifted-winner*
have *above-QtoP*:
$\forall\ b \in A - \{a\}.$
$\forall\ i{::}nat.\ i < length\ p \longrightarrow above\ (q!i)\ b = \{b\} \longrightarrow above\ (p!i)\ b = \{b\}$
using *lifted-above-winner-other lift-a*
unfolding *Profile.lifted-def*
by *metis*
hence $\forall\ b \in A - \{a\}.$
$\{i{::}nat.\ i < length\ p \wedge above\ (q!i)\ b = \{b\}\} \subseteq$
$\{i{::}nat.\ i < length\ p \wedge above\ (p!i)\ b = \{b\}\}$
by (*simp add*: *Collect-mono*)
hence *win-count-other*: $\forall\ b \in A - \{a\}.\ win\text{-}count\ p\ b \geq win\text{-}count\ q\ b$
by (*simp add*: *card-mono sizes*)
show *defer plurality A q* = *defer plurality A p* $\vee$ *defer plurality A q* = $\{a\}$
proof (*cases*)
assume *win-count p a* = *win-count q a*
hence *card* $\{i{::}nat.\ i < length\ p \wedge above\ (p!i)\ a = \{a\}\}$ =
card $\{i{::}nat.\ i < length\ p \wedge above\ (q!i)\ a = \{a\}\}$
using *sizes*
by *simp*
moreover have *finite* $\{i{::}nat.\ i < length\ p \wedge above\ (q!i)\ a = \{a\}\}$
by *simp*
ultimately have
$\{i{::}nat.\ i < length\ p \wedge above\ (p!i)\ a = \{a\}\} =$
$\{i{::}nat.\ i < length\ p \wedge above\ (q!i)\ a = \{a\}\}$
using *a-win-subset*
by (*simp add*: *card-subset-eq*)
hence *above-pq*:
$\forall\ i{::}nat.\ i < length\ p \longrightarrow (above\ (p!i)\ a = \{a\}) = (above\ (q!i)\ a = \{a\})$
by *blast*
moreover have
$\forall\ b \in A - \{a\}.$
$\forall\ i{::}nat.\ i < length\ p \longrightarrow$
$above\ (p!i)\ b = \{b\} \longrightarrow above\ (q!i)\ b = \{b\} \vee above\ (q!i)\ a = \{a\}$
using *lifted-winner*
by *auto*
moreover have
$\forall\ b \in A - \{a\}.$
$\forall\ i{::}nat.\ i < length\ p \longrightarrow above\ (p!i)\ b = \{b\} \longrightarrow above\ (p!i)\ a \neq \{a\}$
proof (*rule ccontr*, *simp*, *safe*, *simp*)
fix
$b :: 'a$ **and**
$i :: nat$
assume
b-in-A: $b \in A$ **and**
i-in-range: $i < length\ p$ **and**
abv-b: *above* $(p!i)\ b = \{b\}$ **and**
abv-a: *above* $(p!i)\ a = \{a\}$

```
    moreover from b-in-A
    have A ≠ {}
      by auto
    moreover from i-in-range
    have linear-order-on A (p!i)
      using lift-a
      unfolding Profile.lifted-def profile-def
      by simp
    ultimately show b = a
      using fin-A above-one-eq
      by metis
  qed
  ultimately have above-PtoQ:
    ∀ b ∈ A − {a}. ∀ i::nat.
      i < length p ⟶ above (p!i) b = {b} ⟶ above (q!i) b = {b}
    by simp
  hence ∀ b ∈ A.
        card {i::nat. i < length p ∧ above (p!i) b = {b}} =
          card {i::nat. i < length q ∧ above (q!i) b = {b}}
  proof (safe)
    fix b :: 'a
    assume
      above-c:
        ∀ c ∈ A − {a}. ∀ i < length p.
          above (p!i) c = {c} ⟶ above (q!i) c = {c} and
      b-in-A: b ∈ A
    show card {i. i < length p ∧ above (p!i) b = {b}} =
           card {i. i < length q ∧ above (q!i) b = {b}}
      using DiffI b-in-A set-disj above-PtoQ above-QtoP above-pq sizes
      by (metis (no-types, lifting))
  qed
  hence {b ∈ A. ∀ c ∈ A. win-count p c ≤ win-count p b} =
          {b ∈ A. ∀ c ∈ A. win-count q c ≤ win-count q b}
    by auto
  hence defer plurality' A q = defer plurality' A p ∨ defer plurality' A q = {a}
    by simp
  hence defer plurality A q = defer plurality A p ∨ defer plurality A q = {a}
    using plurality-mod-elim-equiv empty-not-insert insert-absorb lift-a
    unfolding Profile.lifted-def
    by (metis (no-types, opaque-lifting))
  thus ?thesis
    by simp
next
  assume win-count p a ≠ win-count q a
  hence strict-less: win-count p a < win-count q a
    using win-count-a
    by simp
  have a ∈ defer plurality A p
    using defer-a plurality.elims
```

by (*metis* (*no-types*))
moreover have *non-empty-A*: $A \neq \{\}$
using *lift-a equals0D equiv-prof-except-a-def lifted-imp-equiv-prof-except-a*
by *metis*
moreover have *fin-A*: *finite-profile A p*
using *lift-a*
unfolding *Profile.lifted-def*
by *simp*
ultimately have $a \in$ *defer plurality′ A p*
using *plurality-mod-elim-equiv*
by *metis*
hence *a-in-win-p*: $a \in \{b \in A. \forall\ c \in A.$ *win-count p c* $\leq$ *win-count p b*$\}$
by *simp*
hence $\forall\ b \in A.$ *win-count p b* $\leq$ *win-count p a*
by *simp*
hence *less*: $\forall\ b \in A - \{a\}.$ *win-count q b* $<$ *win-count q a*
using *DiffD1 antisym dual-order.trans not-le-imp-less win-count-a strict-less win-count-other*
by *metis*
hence $\forall\ b \in A - \{a\}. \exists\ c \in A. \neg$ *win-count q c* $\leq$ *win-count q b*
using *lift-a not-le*
unfolding *Profile.lifted-def*
by *metis*
hence $\forall\ b \in A - \{a\}.\ b \notin \{c \in A. \forall\ b \in A.$ *win-count q b* $\leq$ *win-count q c*$\}$
by *blast*
hence $\forall\ b \in A - \{a\}.\ b \notin$ *defer plurality′ A q*
by *simp*
hence $\forall\ b \in A - \{a\}.\ b \notin$ *defer plurality A q*
using *lift-a non-empty-A plurality-mod-elim-equiv*
unfolding *Profile.lifted-def*
by (*metis* (*no-types*, *lifting*))
hence $\forall\ b \in A - \{a\}.\ b \notin$ *defer plurality A q*
by *simp*
moreover have $a \in$ *defer plurality A q*
proof −
have $\forall\ b \in A - \{a\}.$ *win-count q b* $\leq$ *win-count q a*
using *less less-imp-le*
by *metis*
moreover have *win-count q a* $\leq$ *win-count q a*
by *simp*
ultimately have $\forall\ b \in A.$ *win-count q b* $\leq$ *win-count q a*
by *auto*
moreover have $a \in A$
using *a-in-win-p*
by *simp*
ultimately have $a \in \{b \in A. \forall\ c \in A.$ *win-count q c* $\leq$ *win-count q b*$\}$
by *simp*
hence $a \in$ *defer plurality′ A q*
by *simp*

```
    hence a ∈ defer plurality A q
      using plurality-mod-elim-equiv non-empty-A fin-A lift-a non-empty-A
      unfolding Profile.lifted-def
      by (metis (no-types))
    thus ?thesis
      by simp
  qed
  moreover have defer plurality A q ⊆ A
    by simp
  ultimately show ?thesis
    by blast
 qed
qed
```

The plurality rule is invariant-monotone.

```
theorem plurality-mod-def-inv-mono[simp]: defer-invariant-monotonicity plurality
proof (unfold defer-invariant-monotonicity-def, intro conjI impI allI)
 show electoral-module plurality
   by simp
next
 show non-electing plurality
   by simp
next
 fix
   A :: 'a set and
   p :: 'a Profile and
   q :: 'a Profile and
   a :: 'a
 assume a ∈ defer plurality A p ∧ Profile.lifted A p q a
 thus defer plurality A q = defer plurality A p ∨ defer plurality A q = {a}
   using plurality-def-inv-mono-alts
   by metis
qed

end
```

3.10 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.

3.10.1 Definition

fun *borda-score* :: *'a Evaluation-Function* **where**
  borda-score x A p = (∑ *y* ∈ *A*. (*prefer-count p x y*))

fun *borda* :: *'a Electoral-Module* **where**
  borda A p = *max-eliminator borda-score A p*

3.10.2 Soundness

theorem *borda-sound*: *electoral-module borda*
  unfolding *borda.simps*
  using *max-elim-sound*
  by *metis*

3.10.3 Non-Blocking

The Borda module is non-blocking.

theorem *borda-mod-non-blocking*[*simp*]: *non-blocking borda*
  unfolding *borda.simps*
  using *max-elim-non-blocking*
  by *metis*

3.10.4 Non-Electing

The Borda module is non-electing.

theorem *borda-mod-non-electing*[*simp*]: *non-electing borda*
  using *max-elim-non-electing*
  unfolding *borda.simps non-electing-def*
  by *metis*

end

3.11 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.

3.11.1 Definition

fun *condorcet-score* :: *'a Evaluation-Function* **where**
  condorcet-score x A p =
    (*if* (*condorcet-winner A p x*) *then 1 else 0*)

fun *condorcet* :: *'a Electoral-Module* **where**
  condorcet A p = (*max-eliminator condorcet-score*) *A p*

3.11.2 Soundness

theorem *condorcet-sound*: *electoral-module condorcet*
  unfolding *condorcet.simps*
  using *max-elim-sound*
  by *metis*

3.11.3 Property

theorem *condorcet-score-is-condorcet-rating*: *condorcet-rating condorcet-score*
proof (*unfold condorcet-rating-def*, *safe*)
  fix
    A :: *'a set* **and**
    p :: *'a Profile* **and**
    w :: *'a* **and**
    l :: *'a*
  assume
    c-win: *condorcet-winner A p w* **and**
    l-neq-w: *l* ≠ *w*
  hence ¬ *condorcet-winner A p l*
    using *cond-winner-unique-eq*
    by (*metis* (*no-types*))
  thus *condorcet-score l A p* < *condorcet-score w A p*
    using *c-win*
    by *simp*
qed

theorem *condorcet-is-dcc*: *defer-condorcet-consistency condorcet*
proof (*unfold defer-condorcet-consistency-def electoral-module-def*, *safe*)
  fix
    A :: *'a set* **and**
    p :: *'a Profile*
  assume *profile A p*
  hence *well-formed A* (*max-eliminator condorcet-score A p*)
    using *max-elim-sound*

```
    unfolding electoral-module-def
    by metis
  thus well-formed A (condorcet A p)
    by simp
next
  fix
    A :: 'a set and
    p :: 'a Profile and
    a :: 'a
  assume c-win-w: condorcet-winner A p a
  have defer-condorcet-consistency (max-eliminator condorcet-score)
    using cr-eval-imp-dcc-max-elim
    by (simp add: condorcet-score-is-condorcet-rating)
  hence max-eliminator condorcet-score A p =
        ({},
        A − defer (max-eliminator condorcet-score) A p,
        {b ∈ A. condorcet-winner A p b})
    using c-win-w
    unfolding defer-condorcet-consistency-def
    by (metis (no-types))
  thus condorcet A p =
        ({},
        A − defer condorcet A p,
        {d ∈ A. condorcet-winner A p d})
    by simp
qed

end
```

3.12 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.

3.12.1 Definition

```
fun copeland-score :: 'a Evaluation-Function where
```

```
  copeland-score x A p =
    card {y ∈ A . wins x p y} − card {y ∈ A . wins y p x}

fun copeland :: 'a Electoral-Module where
  copeland A p = max-eliminator copeland-score A p
```

3.12.2 Soundness

```
theorem copeland-sound: electoral-module copeland
  unfolding copeland.simps
  using max-elim-sound
  by metis
```

3.12.3 Lemmas

For a Condorcet winner w, we have: "{*card y ∈ A . wins x p y*} = |*A*| − *1*".

```
lemma cond-winner-imp-win-count:
  fixes
    A :: 'a set and
    p :: 'a Profile and
    w :: 'a
  assumes condorcet-winner A p w
  shows card {a ∈ A. wins w p a} = card A − 1
proof −
  have ∀ a ∈ A − {w}. wins w p a
    using assms
    by simp
  hence {a ∈ A − {w}. wins w p a} = A − {w}
    by blast
  hence winner-wins-against-all-others:
    card {a ∈ A − {w}. wins w p a} = card (A − {w})
    by simp
  have w ∈ A
    using assms
    by simp
  hence card (A − {w}) = card A − 1
    using card-Diff-singleton assms
    by metis
  hence winner-amount-one: card {a ∈ A − {w}. wins w p a} = card (A) − 1
    using winner-wins-against-all-others
    by linarith
  have win-for-winner-not-reflexive: ∀ a ∈ {w}. ¬ wins a p a
    by (simp add: wins-irreflex)
  hence {a ∈ {w}. wins w p a} = {}
    by blast
  hence winner-amount-zero: card {a ∈ {w}. wins w p a} = 0
    by simp
  have union:
    {a ∈ A − {w}. wins w p a} ∪ {x ∈ {w}. wins w p x} = {a ∈ A. wins w p a}
```

using *win-for-winner-not-reflexive*
by *blast*
have *finite-defeated*: *finite* $\{a \in A - \{w\}.\ wins\ w\ p\ a\}$
using *assms*
by *simp*
have *finite* $\{a \in \{w\}.\ wins\ w\ p\ a\}$
by *simp*
hence *card* $(\{a \in A - \{w\}.\ wins\ w\ p\ a\} \cup \{a \in \{w\}.\ wins\ w\ p\ a\}) =$
card $\{a \in A - \{w\}.\ wins\ w\ p\ a\} + card\ \{a \in \{w\}.\ wins\ w\ p\ a\}$
using *finite-defeated card-Un-disjoint*
by *blast*
hence *card* $\{a \in A.\ wins\ w\ p\ a\} =$
card $\{a \in A - \{w\}.\ wins\ w\ p\ a\} + card\ \{a \in \{w\}.\ wins\ w\ p\ a\}$
using *union*
by *simp*
thus *?thesis*
using *winner-amount-one winner-amount-zero*
by *linarith*
qed

For a Condorcet winner w, we have: "*card* $\{y \in A\ .\ wins\ y\ p\ x = 0$".

lemma *cond-winner-imp-loss-count*:
fixes
$A :: 'a\ set$ **and**
$p :: 'a\ Profile$ **and**
$w :: 'a$
assumes *condorcet-winner A p w*
shows *card* $\{a \in A.\ wins\ a\ p\ w\} = 0$
using *Collect-empty-eq card-eq-0-iff insert-Diff insert-iff wins-antisym assms*
unfolding *condorcet-winner.simps*
by (*metis* (*no-types*, *lifting*))

Copeland score of a Condorcet winner.

lemma *cond-winner-imp-copeland-score*:
fixes
$A :: 'a\ set$ **and**
$p :: 'a\ Profile$ **and**
$w :: 'a$
assumes *condorcet-winner A p w*
shows *copeland-score w A p = card A − 1*
proof (*unfold copeland-score.simps*)
have *card* $\{a \in A.\ wins\ w\ p\ a\} = card\ A - 1$
using *cond-winner-imp-win-count assms*
by *simp*
moreover have *card* $\{a \in A.\ wins\ a\ p\ w\} = 0$
using *cond-winner-imp-loss-count assms*
by (*metis* (*no-types*))
ultimately show
card $\{a \in A.\ wins\ w\ p\ a\} - card\ \{a \in A.\ wins\ a\ p\ w\} = card\ A - 1$

by *simp*
qed

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

lemma *non-cond-winner-imp-win-count*:
fixes
A :: *'a set* **and**
p :: *'a Profile* **and**
w :: *'a* **and**
l :: *'a*
assumes
winner: *condorcet-winner A p w* **and**
loser: $l \neq w$ **and**
l-in-A: $l \in A$
shows *card* $\{a \in A$. *wins l p* $a\} \leq$ *card A* $- 2$
proof −
have *wins w p l*
using *assms*
by *simp*
hence ¬ *wins l p w*
using *wins-antisym*
by *simp*
moreover have ¬ *wins l p l*
using *wins-irreflex*
by *simp*
ultimately have *wins-of-loser-eq-without-winner*:
$\{y \in A$. *wins l p* $y\} = \{y \in A - \{l, w\}$. *wins l p* $y\}$
by *blast*
have $\forall$ *M f*. *finite M* $\longrightarrow$ *card* $\{x \in M$. *f* $x\} \leq$ *card M*
by (*simp add*: *card-mono*)
moreover have *finite* $(A - \{l, w\})$
using *finite-Diff winner*
by *simp*
ultimately have *card* $\{y \in A - \{l, w\}$. *wins l p* $y\} \leq$ *card* $(A - \{l, w\})$
using *winner*
by (*metis* (*full-types*))
thus *?thesis*
using *assms wins-of-loser-eq-without-winner*
by (*simp add*: *card-Diff-subset*)
qed

3.12.4 Property

The Copeland score is Condorcet rating.

theorem *copeland-score-is-cr*: *condorcet-rating copeland-score*
proof (*unfold condorcet-rating-def*, *unfold copeland-score.simps*, *safe*)
fix

```
    A :: 'a set and
    p :: 'a Profile and
    w :: 'a and
    l :: 'a
  assume
    winner: condorcet-winner A p w and
    l-in-A: l ∈ A and
    l-neq-w: l ≠ w
  hence card {y ∈ A. wins l p y} ≤ card A − 2
    using non-cond-winner-imp-win-count
    by (metis (mono-tags, lifting))
  hence card {y ∈ A. wins l p y} − card {y ∈ A. wins y p l} ≤ card A − 2
    using diff-le-self order.trans
    by blast
  moreover have finite A
    using winner
    by simp
  moreover have card A − 2 < card A − 1
    using card-0-eq diff-less-mono2 empty-iff l-in-A l-neq-w neq0-conv less-one
        Suc-1 zero-less-diff add-diff-cancel-left' diff-is-0-eq Suc-eq-plus1
        card-1-singleton-iff order-less-le singletonD le-zero-eq winner
    unfolding condorcet-winner.simps
    by metis
  ultimately have
    card {y ∈ A. wins l p y} − card {y ∈ A. wins y p l} < card A − 1
    using order-le-less-trans
    by blast
  moreover have card {a ∈ A. wins a p w} = 0
    using cond-winner-imp-loss-count winner
    by (metis (no-types))
  moreover have card A − 1 = card {a ∈ A. wins w p a}
    using cond-winner-imp-win-count winner
    by (metis (full-types))
  ultimately show
    card {y ∈ A. wins l p y} − card {y ∈ A. wins y p l} <
      card {y ∈ A. wins w p y} − card {y ∈ A. wins y p w}
    by linarith
qed

theorem copeland-is-dcc: defer-condorcet-consistency copeland
proof (unfold defer-condorcet-consistency-def electoral-module-def, safe)
  fix
    A :: 'a set and
    p :: 'a Profile
  assume profile A p
  hence well-formed A (max-eliminator copeland-score A p)
    using max-elim-sound
    unfolding electoral-module-def
    by metis
```

```
  thus well-formed A (copeland A p)
    by simp
next
  fix
    A :: 'a set and
    p :: 'a Profile and
    w :: 'a
  assume condorcet-winner A p w
  moreover have defer-condorcet-consistency (max-eliminator copeland-score)
    by (simp add: copeland-score-is-cr)
  moreover have ∀ A p. copeland A p = max-eliminator copeland-score A p
    by simp
  ultimately show
    copeland A p = ({}, A − defer copeland A p, {d ∈ A. condorcet-winner A p d})
    using Collect-cong
    unfolding defer-condorcet-consistency-def
    by (metis (no-types, lifting))
qed

end
```

3.13 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.

3.13.1 Definition

```
fun minimax-score :: 'a Evaluation-Function where
  minimax-score x A p =
    Min {prefer-count p x y | y . y ∈ A − {x}}

fun minimax :: 'a Electoral-Module where
  minimax A p = max-eliminator minimax-score A p
```

3.13.2 Soundness

```
theorem minimax-sound: electoral-module minimax
```

unfolding *minimax.simps*
using *max-elim-sound*
by *metis*

3.13.3 Lemma

lemma *non-cond-winner-minimax-score*:
fixes
A :: *'a set* **and**
p :: *'a Profile* **and**
w :: *'a* **and**
l :: *'a*
assumes
prof: *profile A p* **and**
winner: *condorcet-winner A p w* **and**
l-in-A: $l \in A$ **and**
l-neq-w: $l \neq w$
shows *minimax-score l A p* $\leq$ *prefer-count p l w*
proof (*simp*)
let
?set = {*prefer-count p l y* | *y* . $y \in A - \{l\}$} **and**
?lscore = *minimax-score l A p*
have *finite*: *finite ?set*
using *prof winner finite-Diff*
by *simp*
have *w-not-l*: $w \in A - \{l\}$
using *winner l-neq-w*
by *simp*
hence *not-empty*: *?set* $\neq$ {}
by *blast*
have *?lscore* = *Min ?set*
by *simp*
hence *?lscore* $\in$ *?set* $\wedge$ ($\forall$ *p* $\in$ *?set*. *?lscore* $\leq$ *p*)
using *finite not-empty Min-le Min-eq-iff*
by (*metis* (*no-types*, *lifting*))
thus *Min* {*card* {*i*. *i* < *length p* $\wedge$ (*y*, *l*) $\in$ *p*!*i*} | *y*. *y* $\in$ *A* $\wedge$ *y* $\neq$ *l*} $\leq$
card {*i*. *i* < *length p* $\wedge$ (*w*, *l*) $\in$ *p*!*i*}
using *w-not-l*
by *auto*
qed

3.13.4 Property

theorem *minimax-score-cond-rating*: *condorcet-rating minimax-score*
proof (*unfold condorcet-rating-def minimax-score.simps prefer-count.simps*,
safe, *rule ccontr*)
fix
A :: *'a set* **and**
p :: *'a Profile* **and**
w :: *'a* **and**

$l :: 'a$
assume
winner: *condorcet-winner A p w* **and**
l-in-A: $l \in A$ **and**
l-neq-w:$l \neq w$ **and**
min-leq:
$\neg$ *Min* {*card* {*i*. *i* < *length p* $\wedge$ (*let r* = (*p*!*i*) *in* (*y* $\preceq_r$ *l*))} |
y. *y* $\in$ *A* − {*l*}} <
Min {*card* {*i*. *i* < *length p* $\wedge$ (*let r* = (*p*!*i*) *in* (*y* $\preceq_r$ *w*))} |
y. *y* $\in$ *A* − {*w*}}
hence *min-count-ineq*:
Min {*prefer-count p l y* | *y*. *y* $\in$ *A* − {*l*}} ≥
Min {*prefer-count p w y* | *y*. *y* $\in$ *A* − {*w*}}
by *simp*
have *pref-count-gte-min*:
prefer-count p l w ≥ *Min* {*prefer-count p l y* | *y* . *y* $\in$ *A* − {*l*}}
using *l-in-A l-neq-w condorcet-winner.simps winner non-cond-winner-minimax-score*
minimax-score.simps
by *metis*
have *l-in-A-without-w*: $l \in A - \{w\}$
using *l-in-A*
by (*simp add*: *l-neq-w*)
hence *pref-counts-non-empty*: {*prefer-count p w y* | *y* . *y* $\in$ *A* − {*w*}} ≠ {}
by *blast*
have *finite* (*A* − {*w*})
using *condorcet-winner.simps winner finite-Diff*
by *metis*
hence *finite* {*prefer-count p w y* | *y* . *y* $\in$ *A* − {*w*}}
by *simp*
hence $\exists$ *n* $\in$ *A* − {*w*}. *prefer-count p w n* =
Min {*prefer-count p w y* | *y* . *y* $\in$ *A* − {*w*}}
using *pref-counts-non-empty Min-in*
by *fastforce*
then obtain *n* **where** *pref-count-eq-min*:
prefer-count p w n =
Min {*prefer-count p w y* | *y* . *y* $\in$ *A* − {*w*}} **and**
n-not-w: *n* $\in$ *A* − {*w*}
by *metis*
hence *n-in-A*: *n* $\in$ *A*
using *DiffE*
by *metis*
have *n-neq-w*: *n* ≠ *w*
using *n-not-w*
by *simp*
have *w-in-A*: *w* $\in$ *A*
using *winner*
by *simp*
have *pref-count-n-w-ineq*: *prefer-count p w n* > *prefer-count p n w*
using *n-not-w winner*

```
    by simp
  have pref-count-l-w-n-ineq: prefer-count p l w ≥ prefer-count p w n
    using pref-count-gte-min min-count-ineq pref-count-eq-min
    by linarith
  hence prefer-count p n w ≥ prefer-count p w l
    using n-in-A w-in-A l-in-A n-neq-w l-neq-w pref-count-sym winner
    unfolding condorcet-winner.simps
    by metis
  hence prefer-count p l w > prefer-count p w l
    using n-in-A w-in-A l-in-A n-neq-w l-neq-w pref-count-sym winner
        pref-count-n-w-ineq pref-count-l-w-n-ineq
    unfolding condorcet-winner.simps
    by linarith
  hence wins l p w
    by simp
  thus False
    using l-in-A-without-w wins-antisym winner
    unfolding condorcet-winner.simps
    by metis
qed

theorem minimax-is-dcc: defer-condorcet-consistency minimax
proof (unfold defer-condorcet-consistency-def electoral-module-def, safe)
  fix
    A :: 'a set and
    p :: 'a Profile
  assume profile A p
  hence well-formed A (max-eliminator minimax-score A p)
    using max-elim-sound par-comp-result-sound
    by metis
  thus well-formed A (minimax A p)
    by simp
next
  fix
    A :: 'a set and
    p :: 'a Profile and
    w :: 'a
  assume cwin-w: condorcet-winner A p w
  have max-mmaxscore-dcc:
    defer-condorcet-consistency (max-eliminator minimax-score)
    using cr-eval-imp-dcc-max-elim
    by (simp add: minimax-score-cond-rating)
  hence
    max-eliminator minimax-score A p =
      ({},
       A − defer (max-eliminator minimax-score) A p,
       {a ∈ A. condorcet-winner A p a})
    using cwin-w
    unfolding defer-condorcet-consistency-def
```

by (*metis* (*no-types*))
thus
minimax A p =
({},
A − *defer minimax A p*,
{*d* ∈ *A*. *condorcet-winner A p d*})
by *simp*
qed

end

Chapter 4

Compositional Structures

4.1 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.

4.1.1 Properties

theorem *drop-zero-mod-rej-zero*[*simp*]:
  fixes $r :: 'a$ *Preference-Relation*
  assumes *linear-order r*
  shows *rejects 0* (*drop-module 0 r*)
proof (*unfold rejects-def*, *safe*)
  show *electoral-module* (*drop-module 0 r*)
    using *assms*
    by *simp*
next
  fix
    $A :: 'a$ *set* **and**
    $p :: 'a$ *Profile*
  assume
    finite-A: *finite A* **and**
    prof-A: *profile A p*
  have *connex UNIV r*
    using *assms lin-ord-imp-connex*
    by *auto*
  hence *connex*: *connex A* (*limit A r*)
    using *limit-presv-connex subset-UNIV*
    by *metis*
  have $\forall\ B\ a.\ B \neq \{\} \vee (a::'a) \notin B$

```
    by simp
  hence ∀ a B. a ∈ A ∧ a ∈ B ⟶ connex B (limit A r) ⟶
          ¬ card (above (limit A r) a) ≤ 0
    using above-connex above-presv-limit card-eq-0-iff
          finite-A finite-subset le-0-eq assms
    by (metis (no-types))
  hence {a ∈ A. card (above (limit A r) a) ≤ 0} = {}
    using connex
    by auto
  hence card {a ∈ A. card (above (limit A r) a) ≤ 0} = 0
    using card.empty
    by (metis (full-types))
  thus card (reject (drop-module 0 r) A p) = 0
    by simp
qed
```

The drop module rejects n alternatives (if there are n alternatives). NOTE: The induction proof is still missing. Following is the proof for n=2.

```
theorem drop-two-mod-rej-two[simp]:
  fixes r :: 'a Preference-Relation
  assumes linear-order r
  shows rejects 2 (drop-module 2 r)
proof −
  have rej-drop-eq-def-pass: reject (drop-module 2 r) = defer (pass-module 2 r)
    by simp
  obtain
    m :: 'a Electoral-Module ⇒ nat ⇒ 'a set and
    m' :: 'a Electoral-Module ⇒ nat ⇒ 'a Profile where
      ∀ f n. (∃ A p. n ≤ card A ∧ finite-profile A p ∧ card (reject f A p) ≠ n) =
          (n ≤ card (m f n) ∧ finite-profile (m f n) (m' f n) ∧
            card (reject f (m f n) (m' f n)) ≠ n)
    by moura
  hence rejected-card:
    ∀ f n.
      ¬ rejects n f ∧ electoral-module f ⟶
        n ≤ card (m f n) ∧ finite-profile (m f n) (m' f n) ∧
          card (reject f (m f n) (m' f n)) ≠ n
    unfolding rejects-def
    by blast
  have
    2 ≤ card (m (drop-module 2 r) 2) ∧ finite (m (drop-module 2 r) 2) ∧
      profile (m (drop-module 2 r) 2) (m' (drop-module 2 r) 2) ⟶
        card (reject (drop-module 2 r) (m (drop-module 2 r) 2)
          (m' (drop-module 2 r) 2)) = 2
    using rej-drop-eq-def-pass assms pass-two-mod-def-two
    unfolding defers-def
    by (metis (no-types))
  thus ?thesis
    using rejected-card drop-mod-sound assms
```

by *blast*
qed

The pass and drop module are (disjoint-)compatible.

theorem *drop-pass-disj-compat*[*simp*]:
fixes
r :: *'a Preference-Relation* **and**
n :: *nat*
assumes *linear-order r*
shows *disjoint-compatibility* (*drop-module n r*) (*pass-module n r*)
proof (*unfold disjoint-compatibility-def*, *safe*)
show *electoral-module* (*drop-module n r*)
using *assms*
by *simp*
next
show *electoral-module* (*pass-module n r*)
using *assms*
by *simp*
next
fix *A* :: *'a set*
obtain *p* :: *'a Profile* **where**
profile A p
using *empty-iff empty-set profile-set*
by *metis*
show
∃ *B* ⊆ *A*.
(∀ *a* ∈ *B*. *indep-of-alt* (*drop-module n r*) *A a* ∧
(∀ *p*. *profile A p* ⟶ *a* ∈ *reject* (*drop-module n r*) *A p*)) ∧
(∀ *a* ∈ *A* − *B*. *indep-of-alt* (*pass-module n r*) *A a* ∧
(∀ *p*. *profile A p* ⟶ *a* ∈ *reject* (*pass-module n r*) *A p*))
proof
have *same-A*:
∀ *p q*. *profile A p* ∧ *profile A q* ⟶
reject (*drop-module n r*) *A p* = *reject* (*drop-module n r*) *A q*
by *auto*
let *?A* = *reject* (*drop-module n r*) *A p*
have *?A* ⊆ *A*
by *auto*
moreover have ∀ *a* ∈ *?A*. *indep-of-alt* (*drop-module n r*) *A a*
using *assms*
unfolding *indep-of-alt-def*
by *simp*
moreover have
∀ *a* ∈ *?A*. ∀ *p*. *profile A p* ⟶ *a* ∈ *reject* (*drop-module n r*) *A p*
by *auto*
moreover have ∀ *a* ∈ *A* − *?A*. *indep-of-alt* (*pass-module n r*) *A a*
using *assms*
unfolding *indep-of-alt-def*
by *simp*

```
    moreover have
     ∀ a ∈ A − ?A. ∀ p. profile A p ⟶ a ∈ reject (pass-module n r) A p
     by auto
    ultimately show
     ?A ⊆ A ∧
      (∀ a ∈ ?A. indep-of-alt (drop-module n r) A a ∧
        (∀ p. profile A p ⟶ a ∈ reject (drop-module n r) A p)) ∧
      (∀ a ∈ A − ?A. indep-of-alt (pass-module n r) A a ∧
        (∀ p. profile A p ⟶ a ∈ reject (pass-module n r) A p))
     by simp
  qed
qed

end
```

4.2 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.

4.2.1 Definition

```
fun revision-composition :: 'a Electoral-Module ⇒ 'a Electoral-Module where
  revision-composition m A p = ({}, A − elect m A p, elect m A p)

abbreviation rev ::
'a Electoral-Module ⇒ 'a Electoral-Module (-↓ 50) where
  m↓ == revision-composition m
```

4.2.2 Soundness

```
theorem rev-comp-sound[simp]:
  fixes m :: 'a Electoral-Module
  assumes electoral-module m
  shows electoral-module (revision-composition m)
proof −
  from assms
  have ∀ A p. profile A p ⟶ elect m A p ⊆ A
    using elect-in-alts
    by metis
```

hence $\forall\ A\ p.\ profile\ A\ p \longrightarrow (A - elect\ m\ A\ p) \cup elect\ m\ A\ p = A$
by *blast*
hence *unity*:
$\forall\ A\ p.\ profile\ A\ p \longrightarrow$
set-equals-partition A (*revision-composition* $m\ A\ p$)
by *simp*
have $\forall\ A\ p.\ profile\ A\ p \longrightarrow (A - elect\ m\ A\ p) \cap elect\ m\ A\ p = \{\}$
by *blast*
hence *disjoint*:
$\forall\ A\ p.\ profile\ A\ p \longrightarrow$ *disjoint3* (*revision-composition* $m\ A\ p$)
by *simp*
from *unity disjoint*
show *?thesis*
by (*simp add*: *electoral-module-def*)
qed

4.2.3 Composition Rules

An electoral module received by revision is never electing.

theorem *rev-comp-non-electing*[*simp*]:
fixes m :: *'a Electoral-Module*
assumes *electoral-module* m
shows *non-electing* ($m\downarrow$)
using *assms*
unfolding *non-electing-def*
by *simp*

Revising an electing electoral module results in a non-blocking electoral module.

theorem *rev-comp-non-blocking*[*simp*]:
fixes m :: *'a Electoral-Module*
assumes *electing* m
shows *non-blocking* ($m\downarrow$)
proof (*unfold non-blocking-def*, *safe*, *simp-all*)
show *electoral-module* ($m\downarrow$)
using *assms rev-comp-sound*
unfolding *electing-def*
by (*metis* (*no-types*, *lifting*))
next
fix
A :: *'a set* **and**
p :: *'a Profile* **and**
x :: *'a*
assume
fin-A: *finite* A **and**
prof-A: *profile* $A\ p$ **and**
no-elect: $A - elect\ m\ A\ p = A$ **and**
x-in-A: $x \in A$

```
  from no-elect have non-elect:
    non-electing m
    using assms prof-A x-in-A fin-A empty-iff
          Diff-disjoint Int-absorb2 elect-in-alts
    unfolding electing-def
    by (metis (no-types, lifting))
  show False
    using non-elect assms empty-iff fin-A prof-A x-in-A
    unfolding electing-def non-electing-def
    by (metis (no-types, lifting))
qed
```

Revising an invariant monotone electoral module results in a defer-invariant-monotone electoral module.

```
theorem rev-comp-def-inv-mono[simp]:
  fixes m :: 'a Electoral-Module
  assumes invariant-monotonicity m
  shows defer-invariant-monotonicity (m↓)
proof (unfold defer-invariant-monotonicity-def, safe)
  show electoral-module (m↓)
    using assms rev-comp-sound
    unfolding invariant-monotonicity-def
    by simp
next
  show non-electing (m↓)
    using assms rev-comp-non-electing
    unfolding invariant-monotonicity-def
    by simp
next
  fix
    A :: 'a set and
    p :: 'a Profile and
    q :: 'a Profile and
    a :: 'a and
    x :: 'a and
    x' :: 'a
  assume
    rev-p-defer-a: a ∈ defer (m↓) A p and
    a-lifted: lifted A p q a and
    rev-q-defer-x: x ∈ defer (m↓) A q and
    x-non-eq-a: x ≠ a and
    rev-q-defer-x': x' ∈ defer (m↓) A q
  from rev-p-defer-a
  have elect-a-in-p: a ∈ elect m A p
    by simp
  from rev-q-defer-x x-non-eq-a
  have elect-no-unique-a-in-q: elect m A q ≠ {a}
    by force
  from assms
```

```
  have elect m A q = elect m A p
    using a-lifted elect-a-in-p elect-no-unique-a-in-q
    unfolding invariant-monotonicity-def
    by (metis (no-types))
  thus x' ∈ defer (m↓) A p
    using rev-q-defer-x'
    by simp
next
  fix
    A :: 'a set and
    p :: 'a Profile and
    q :: 'a Profile and
    a :: 'a and
    x :: 'a and
    x' :: 'a
  assume
    rev-p-defer-a: a ∈ defer (m↓) A p and
    a-lifted: lifted A p q a and
    rev-q-defer-x: x ∈ defer (m↓) A q and
    x-non-eq-a: x ≠ a and
    rev-p-defer-x': x' ∈ defer (m↓) A p
  have reject-and-defer:
    (A − elect m A q, elect m A q) = snd ((m↓) A q)
    by force
  have elect-p-eq-defer-rev-p: elect m A p = defer (m↓) A p
    by simp
  hence elect-a-in-p: a ∈ elect m A p
    using rev-p-defer-a
    by presburger
  have elect m A q ≠ {a}
    using rev-q-defer-x x-non-eq-a
    by force
  with assms
  show x' ∈ defer (m↓) A q
    using a-lifted rev-p-defer-x' snd-conv elect-a-in-p
          elect-p-eq-defer-rev-p reject-and-defer
    unfolding invariant-monotonicity-def
    by (metis (no-types))
next
  fix
    A :: 'a set and
    p :: 'a Profile and
    q :: 'a Profile and
    a :: 'a and
    x :: 'a and
    x' :: 'a
  assume
    a ∈ defer (m↓) A p and
    lifted A p q a and
```

```
    x' ∈ defer (m↓) A q
  with assms
  show x' ∈ defer (m↓) A p
    using empty-iff insertE snd-conv revision-composition.elims
    unfolding invariant-monotonicity-def
    by metis
next
  fix
    A :: 'a set and
    p :: 'a Profile and
    q :: 'a Profile and
    a :: 'a and
    x :: 'a and
    x' :: 'a
  assume
    rev-p-defer-a: a ∈ defer (m↓) A p and
    a-lifted: lifted A p q a and
    rev-q-not-defer-a: a ∉ defer (m↓) A q
  from assms
  have lifted-inv:
    ∀ A p q a. a ∈ elect m A p ∧ lifted A p q a ⟶
      elect m A q = elect m A p ∨ elect m A q = {a}
    unfolding invariant-monotonicity-def
    by (metis (no-types))
  have p-defer-rev-eq-elect: defer (m↓) A p = elect m A p
    by simp
  have q-defer-rev-eq-elect: defer (m↓) A q = elect m A q
    by simp
  thus x' ∈ defer (m↓) A q
    using p-defer-rev-eq-elect lifted-inv a-lifted rev-p-defer-a rev-q-not-defer-a
    by blast
qed

end
```

4.3 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.

4.3.1 Definition

fun *sequential-composition* :: *'a Electoral-Module* ⇒ *'a Electoral-Module* ⇒
'a Electoral-Module **where**
sequential-composition m n A p =
(*let new-A* = *defer m A p*;
new-p = *limit-profile new-A p in* (
(*elect m A p*) ∪ (*elect n new-A new-p*),
(*reject m A p*) ∪ (*reject n new-A new-p*),
defer n new-A new-p))

abbreviation *sequence* ::
'a Electoral-Module ⇒ *'a Electoral-Module* ⇒ *'a Electoral-Module*
(**infix** ▷ *50*) **where**
m ▷ *n* == *sequential-composition m n*

fun *sequential-composition'* :: *'a Electoral-Module* ⇒ *'a Electoral-Module* ⇒
'a Electoral-Module **where**
sequential-composition' m n A p =
(*let* (*m-e*, *m-r*, *m-d*) = *m A p*; *new-A* = *m-d*;
new-p = *limit-profile new-A p*;
(*n-e*, *n-r*, *n-d*) = *n new-A new-p in*
(*m-e* ∪ *n-e*, *m-r* ∪ *n-r*, *n-d*))

lemma *seq-comp-presv-disj*:
fixes
m :: *'a Electoral-Module* **and**
n :: *'a Electoral-Module* **and**
A :: *'a set* **and**
p :: *'a Profile*
assumes *module-m*: *electoral-module m* **and**
module-n: *electoral-module n* **and**
prof-p: *profile A p*
shows *disjoint3* ((*m* ▷ *n*) *A p*)
proof −
let *?new-A* = *defer m A p*
let *?new-p* = *limit-profile ?new-A p*
have *prof-def-lim*: *profile* (*defer m A p*) (*limit-profile* (*defer m A p*) *p*)
using *def-presv-prof prof-p module-m*
by *metis*
have *defer-in-A*:
∀ *A' p' m' a*.
profile A' p' ∧ *electoral-module m'* ∧ (*a*::*'a*) ∈ *defer m' A' p'* ⟶ *a* ∈ *A'*
using *UnCI result-presv-alts*
by (*metis* (*mono-tags*))
from *module-m prof-p*
have *disjoint-m*: *disjoint3* (*m A p*)
unfolding *electoral-module-def well-formed.simps*
by *blast*
from *module-m module-n def-presv-prof prof-p*

have *disjoint-n*: *disjoint3* (*n ?new-A ?new-p*)
unfolding *electoral-module-def well-formed.simps*
by *metis*
have *disj-n*:
elect m A p ∩ *reject m A p* = {} ∧
elect m A p ∩ *defer m A p* = {} ∧
reject m A p ∩ *defer m A p* = {}
using *prof-p module-m*
by (*simp add*: *result-disj*)
have *reject n* (*defer m A p*) (*limit-profile* (*defer m A p*) *p*) ⊆ *defer m A p*
using *def-presv-prof reject-in-alts prof-p module-m module-n*
by *metis*
with *disjoint-m module-m module-n prof-p*
have *elect-reject-diff*: *elect m A p* ∩ *reject n ?new-A ?new-p* = {}
using *disj-n*
by (*simp add*: *disjoint-iff-not-equal subset-eq*)
from *prof-p module-m module-n*
have *elec-n-in-def-m*:
elect n (*defer m A p*) (*limit-profile* (*defer m A p*) *p*) ⊆ *defer m A p*
using *def-presv-prof elect-in-alts*
by *metis*
have *elect-defer-diff*: *elect m A p* ∩ *defer n ?new-A ?new-p* = {}
proof −
obtain *f* :: *'a set* ⇒ *'a set* ⇒ *'a* **where**
∀ *B B'*.
(∃ *a b*. *a* ∈ *B'* ∧ *b* ∈ *B* ∧ *a* = *b*) =
(*f B B'* ∈ *B'* ∧ (∃ *a*. *a* ∈ *B* ∧ *f B B'* = *a*))
using *disjoint-iff*
by *metis*
then obtain *g* :: *'a set* ⇒ *'a set* ⇒ *'a* **where**
∀ *B B'*.
(*B* ∩ *B'* = {} ⟶ (∀ *a b*. *a* ∈ *B* ∧ *b* ∈ *B'* ⟶ *a* ≠ *b*)) ∧
(*B* ∩ *B'* ≠ {} ⟶ *f B B'* ∈ *B* ∧ *g B B'* ∈ *B'* ∧ *f B B'* = *g B B'*)
by *auto*
thus *?thesis*
using *defer-in-A disj-n module-n prof-def-lim*
by (*metis* (*no-types*))
qed
have *rej-intersect-new-elect-empty*: *reject m A p* ∩ *elect n ?new-A ?new-p* = {}
using *disj-n disjoint-m disjoint-n def-presv-prof prof-p*
module-m module-n elec-n-in-def-m
by *blast*
have (*elect m A p* ∪ *elect n ?new-A ?new-p*) ∩
(*reject m A p* ∪ *reject n ?new-A ?new-p*) = {}
proof (*safe*)
fix *x* :: *'a*
assume
x ∈ *elect m A p* **and**
x ∈ *reject m A p*

hence $x \in elect\ m\ A\ p \cap reject\ m\ A\ p$
by *simp*
thus $x \in \{\}$
using *disj-n*
by *simp*
next
fix $x :: 'a$
assume
$x \in elect\ m\ A\ p$ **and**
$x \in reject\ n\ (defer\ m\ A\ p)$
$(limit\text{-}profile\ (defer\ m\ A\ p)\ p)$
thus $x \in \{\}$
using *elect-reject-diff*
by *blast*
next
fix $x :: 'a$
assume
$x \in elect\ n\ (defer\ m\ A\ p)\ (limit\text{-}profile\ (defer\ m\ A\ p)\ p)$ **and**
$x \in reject\ m\ A\ p$
thus $x \in \{\}$
using *rej-intersect-new-elect-empty*
by *blast*
next
fix $x :: 'a$
assume
$x \in elect\ n\ (defer\ m\ A\ p)\ (limit\text{-}profile\ (defer\ m\ A\ p)\ p)$ **and**
$x \in reject\ n\ (defer\ m\ A\ p)\ (limit\text{-}profile\ (defer\ m\ A\ p)\ p)$
thus $x \in \{\}$
using *disjoint-iff-not-equal module-n prof-def-lim result-disj*
by *metis*
qed
moreover have
$(elect\ m\ A\ p \cup elect\ n\ ?new\text{-}A\ ?new\text{-}p) \cap (defer\ n\ ?new\text{-}A\ ?new\text{-}p) = \{\}$
using *Int-Un-distrib2 Un-empty elect-defer-diff module-n*
prof-def-lim result-disj
by (*metis* (*no-types*))
moreover have
$(reject\ m\ A\ p \cup reject\ n\ ?new\text{-}A\ ?new\text{-}p) \cap (defer\ n\ ?new\text{-}A\ ?new\text{-}p) = \{\}$
proof (*safe*)
fix $x :: 'a$
assume
x-in-def: $x \in defer\ n\ (defer\ m\ A\ p)\ (limit\text{-}profile\ (defer\ m\ A\ p)\ p)$ **and**
x-in-rej: $x \in reject\ m\ A\ p$
from *x-in-def*
have $x \in defer\ m\ A\ p$
using *defer-in-A module-n prof-def-lim*
by *blast*
with *x-in-rej*
have $x \in reject\ m\ A\ p \cap defer\ m\ A\ p$

```
    by fastforce
  thus x ∈ {}
    using disj-n
    by blast
next
  fix x :: 'a
  assume
    x ∈ defer n (defer m A p) (limit-profile (defer m A p) p) and
    x ∈ reject n (defer m A p) (limit-profile (defer m A p) p)
  thus x ∈ {}
    using module-n prof-def-lim reject-not-elec-or-def
    by fastforce
qed
ultimately have
  disjoint3 (elect m A p ∪ elect n ?new-A ?new-p,
             reject m A p ∪ reject n ?new-A ?new-p,
             defer n ?new-A ?new-p)
  by simp
thus ?thesis
  unfolding sequential-composition.simps
  by metis
qed

lemma seq-comp-presv-alts:
  fixes
    m :: 'a Electoral-Module and
    n :: 'a Electoral-Module and
    A :: 'a set and
    p :: 'a Profile
  assumes module-m: electoral-module m and
          module-n: electoral-module n and
          prof-p:   profile A p
  shows set-equals-partition A ((m ▷ n) A p)
proof −
  let ?new-A = defer m A p
  let ?new-p = limit-profile ?new-A p
  have elect-reject-diff: elect m A p ∪ reject m A p ∪ ?new-A = A
    using module-m prof-p
    by (simp add: result-presv-alts)
  have elect n ?new-A ?new-p ∪
         reject n ?new-A ?new-p ∪
           defer n ?new-A ?new-p = ?new-A
    using module-m module-n prof-p def-presv-prof result-presv-alts
    by metis
  hence (elect m A p ∪ elect n ?new-A ?new-p) ∪
         (reject m A p ∪ reject n ?new-A ?new-p) ∪
           defer n ?new-A ?new-p = A
    using elect-reject-diff
    by blast
```

hence *set-equals-partition A*
(*elect m A p* ∪ *elect n ?new-A ?new-p*,
reject m A p ∪ *reject n ?new-A ?new-p*,
defer n ?new-A ?new-p)
by *simp*
thus *?thesis*
unfolding *sequential-composition.simps*
by *metis*
qed

lemma *seq-comp-alt-eq*[*code*]: *sequential-composition* = *sequential-composition′*
proof (*unfold sequential-composition′.simps sequential-composition.simps*)
have ∀ *m n A E*.
(*case m A E of* (*e*, *r*, *d*) ⇒
case n d (*limit-profile d E*) *of* (*e′*, *r′*, *d′*) ⇒
(*e* ∪ *e′*, *r* ∪ *r′*, *d′*)) =
(*elect m A E* ∪ *elect n* (*defer m A E*) (*limit-profile* (*defer m A E*) *E*),
reject m A E ∪ *reject n* (*defer m A E*) (*limit-profile* (*defer m A E*) *E*),
defer n (*defer m A E*) (*limit-profile* (*defer m A E*) *E*))
using *case-prod-beta′*
by (*metis* (*no-types*, *lifting*))
thus
(λ *m n A p*.
let A′ = *defer m A p*; *p′* = *limit-profile A′ p in*
(*elect m A p* ∪ *elect n A′ p′*, *reject m A p* ∪ *reject n A′ p′*, *defer n A′ p′*)) =
(λ *m n A pr*.
let (*e*, *r*, *d*) = *m A pr*; *A′* = *d*; *p′* = *limit-profile A′ pr*;
(*e′*, *r′*, *d′*) = *n A′ p′ in*
(*e* ∪ *e′*, *r* ∪ *r′*, *d′*))
by *metis*
qed

4.3.2 Soundness

theorem *seq-comp-sound*[*simp*]:
fixes
m :: *′a Electoral-Module* **and**
n :: *′a Electoral-Module* **and**
A :: *′a set* **and**
p :: *′a Profile*
assumes
electoral-module m **and**
electoral-module n
shows *electoral-module* (*m* ▷ *n*)
proof (*unfold electoral-module-def*, *safe*)
fix
A :: *′a set* **and**
p :: *′a Profile*
assume *prof-A*: *profile A p*

have $\forall\ r.\ well\text{-}formed\ (A::'a\ set)\ r = (disjoint3\ r \wedge set\text{-}equals\text{-}partition\ A\ r)$
by *simp*
thus *well-formed A* $((m \triangleright n)\ A\ p)$
using *assms seq-comp-presv-disj seq-comp-presv-alts prof-A*
by *metis*
qed

4.3.3 Lemmas

lemma *seq-comp-dec-only-def*:
fixes
$m :: 'a$ *Electoral-Module* **and**
$n :: 'a$ *Electoral-Module* **and**
$A :: 'a$ *set* **and**
$p :: 'a$ *Profile*
assumes
module-m: *electoral-module m* **and**
module-n: *electoral-module n* **and**
f-prof: *profile A p* **and**
empty-defer: *defer m A p* $= \{\}$
shows $(m \triangleright n)\ A\ p = m\ A\ p$
proof
have
$\forall\ m'\ A'\ p'.$
electoral-module $m' \wedge$ *profile* $A'\ p' \longrightarrow$
profile $(defer\ m'\ A'\ p')\ (limit\text{-}profile\ (defer\ m'\ A'\ p')\ p')$
using *def-presv-prof*
by *metis*
hence *profile* $\{\}$ $(limit\text{-}profile\ (defer\ m\ A\ p)\ p)$
using *empty-defer f-prof module-m*
by *metis*
hence
$(elect\ m\ A\ p) \cup (elect\ n\ (defer\ m\ A\ p)\ (limit\text{-}profile\ (defer\ m\ A\ p)\ p)) =$
$elect\ m\ A\ p$
using *elect-in-alts empty-defer module-n*
by *auto*
thus $elect\ (m \triangleright n)\ A\ p = elect\ m\ A\ p$
using *fst-conv*
unfolding *sequential-composition.simps*
by *metis*
next
have *rej-empty*:
$\forall\ m'\ p'.$
electoral-module $m' \wedge$ *profile* $(\{\}::'a\ set)\ p' \longrightarrow reject\ m'\ \{\}\ p' = \{\}$
using *bot.extremum-uniqueI reject-in-alts*
by *metis*
have *prof-no-alt*: *profile* $\{\}$ $(limit\text{-}profile\ (defer\ m\ A\ p)\ p)$
using *empty-defer f-prof module-m limit-profile-sound*
by *auto*

hence (*reject m A p*, *defer n* {} (*limit-profile* {} *p*)) = *snd* (*m A p*)
using *bot.extremum-uniqueI defer-in-alts empty-defer module-n prod.collapse*
by (*metis* (*no-types*))
thus *snd* ((*m* ▷ *n*) *A p*) = *snd* (*m A p*)
using *rej-empty empty-defer module-n prof-no-alt*
by *simp*
qed

lemma *seq-comp-def-then-elect*:
fixes
m :: *'a Electoral-Module* **and**
n :: *'a Electoral-Module* **and**
A :: *'a set* **and**
p :: *'a 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 A p*
shows *elect* (*m* ▷ *n*) *A p* = *defer m A p*
proof (*cases*)
assume *A* = {}
with *electing-n n-electing-m f-prof*
show *?thesis*
using *bot.extremum-uniqueI defer-in-alts elect-in-alts seq-comp-sound*
unfolding *electing-def non-electing-def*
by *metis*
next
assume *non-empty-A*: *A* ≠ {}
from *n-electing-m f-prof*
have *ele*: *elect m A p* = {}
unfolding *non-electing-def*
by *simp*
from *non-empty-A def-one-m f-prof finite*
have *def-card*: *card* (*defer m A p*) = *1*
unfolding *defers-def*
by (*simp add*: *Suc-leI card-gt-0-iff*)
with *n-electing-m f-prof*
have *def*: ∃ *a* ∈ *A*. *defer m A p* = {*a*}
using *card-1-singletonE defer-in-alts singletonI subsetCE*
unfolding *non-electing-def*
by *metis*
from *ele def n-electing-m*
have *rej*: ∃ *a* ∈ *A*. *reject m A p* = *A* − {*a*}
using *Diff-empty def-one-m f-prof reject-not-elec-or-def*
unfolding *defers-def*
by *metis*
from *ele rej def n-electing-m f-prof*
have *res-m*: ∃ *a* ∈ *A*. *m A p* = ({}, *A* − {*a*}, {*a*})

```
    using Diff-empty elect-rej-def-combination reject-not-elec-or-def
    unfolding non-electing-def
    by metis
  hence ∃ a ∈ A. elect (m ▷ n) A p = elect n {a} (limit-profile {a} p)
    using prod.sel sup-bot.left-neutral
    unfolding sequential-composition.simps
    by metis
  with def-card def electing-n n-electing-m f-prof
  have ∃ a ∈ A. elect (m ▷ n) A p = {a}
    using electing-for-only-alt fst-conv def-presv-prof sup-bot.left-neutral
    unfolding non-electing-def sequential-composition.simps
    by metis
  with def def-card electing-n n-electing-m f-prof res-m
  show ?thesis
    using def-presv-prof electing-for-only-alt fst-conv sup-bot.left-neutral
    unfolding non-electing-def sequential-composition.simps
    by metis
qed

lemma seq-comp-def-card-bounded:
  fixes
    m :: 'a Electoral-Module and
    n :: 'a Electoral-Module and
    A :: 'a set and
    p :: 'a Profile
  assumes
    electoral-module m and
    electoral-module n and
    finite-profile A p
  shows card (defer (m ▷ n) A p) ≤ card (defer m A p)
  using card-mono defer-in-alts assms def-presv-prof snd-conv finite-subset
  unfolding sequential-composition.simps
  by metis

lemma seq-comp-def-set-bounded:
  fixes
    m :: 'a Electoral-Module and
    n :: 'a Electoral-Module and
    A :: 'a set and
    p :: 'a Profile
  assumes
    electoral-module m and
    electoral-module n and
    profile A p
  shows defer (m ▷ n) A p ⊆ defer m A p
  using defer-in-alts assms snd-conv def-presv-prof
  unfolding sequential-composition.simps
  by metis
```

lemma *seq-comp-defers-def-set*:
fixes
m :: *'a Electoral-Module* **and**
n :: *'a Electoral-Module* **and**
A :: *'a set* **and**
p :: *'a Profile*
shows *defer* (*m* ▷ *n*) *A p* = *defer n* (*defer m A p*) (*limit-profile* (*defer m A p*) *p*)
using *snd-conv*
unfolding *sequential-composition.simps*
by *metis*

lemma *seq-comp-def-then-elect-elec-set*:
fixes
m :: *'a Electoral-Module* **and**
n :: *'a Electoral-Module* **and**
A :: *'a set* **and**
p :: *'a Profile*
shows *elect* (*m* ▷ *n*) *A p* =
elect n (*defer m A p*) (*limit-profile* (*defer m A p*) *p*) ∪ (*elect m A p*)
using *Un-commute fst-conv*
unfolding *sequential-composition.simps*
by *metis*

lemma *seq-comp-elim-one-red-def-set*:
fixes
m :: *'a Electoral-Module* **and**
n :: *'a Electoral-Module* **and**
A :: *'a set* **and**
p :: *'a Profile*
assumes
electoral-module m **and**
eliminates 1 n **and**
profile A p **and**
card (*defer m A p*) > *1*
shows *defer* (*m* ▷ *n*) *A p* ⊂ *defer m A p*
using *assms snd-conv def-presv-prof single-elim-imp-red-def-set*
unfolding *sequential-composition.simps*
by *metis*

lemma *seq-comp-def-set-sound*:
fixes
m :: *'a Electoral-Module* **and**
n :: *'a Electoral-Module* **and**
A :: *'a set* **and**
p :: *'a Profile*
assumes
electoral-module m **and**
electoral-module n **and**
profile A p

shows *defer* $(m \triangleright n)$ A $p \subseteq$ *defer* m A p
using *assms seq-comp-def-set-bounded*
by *simp*

lemma *seq-comp-def-set-trans*:
fixes
$m :: 'a$ *Electoral-Module* **and**
$n :: 'a$ *Electoral-Module* **and**
$A :: 'a$ *set* **and**
$p :: 'a$ *Profile* **and**
$q :: 'a$ *Profile* **and**
$a :: 'a$
assumes
$a \in$ (*defer* $(m \triangleright n)$ A p) **and**
electoral-module $m \wedge$ *electoral-module* n **and**
profile A p
shows $a \in$ *defer* n (*defer* m A p) (*limit-profile* (*defer* m A p) p) $\wedge$
$a \in$ *defer* m A p
using *seq-comp-def-set-bounded assms in-mono seq-comp-defers-def-set*
by (*metis* (*no-types*, *opaque-lifting*))

4.3.4 Composition Rules

The sequential composition preserves the non-blocking property.

theorem *seq-comp-presv-non-blocking*[*simp*]:
fixes
$m :: 'a$ *Electoral-Module* **and**
$n :: 'a$ *Electoral-Module*
assumes
non-blocking-m: *non-blocking* m **and**
non-blocking-n: *non-blocking* n
shows *non-blocking* $(m \triangleright n)$
proof −
fix
$A :: 'a$ *set* **and**
$p :: 'a$ *Profile*
let *?input-sound* $= A \neq \{\} \wedge$ *finite-profile* A p
from *non-blocking-m*
have *?input-sound* $\longrightarrow$ *reject* m A $p \neq A$
unfolding *non-blocking-def*
by *simp*
with *non-blocking-m*
have *A-reject-diff*: *?input-sound* $\longrightarrow A -$ *reject* m A $p \neq \{\}$
using *Diff-eq-empty-iff reject-in-alts subset-antisym*
unfolding *non-blocking-def*
by *metis*
from *non-blocking-m*
have *?input-sound* $\longrightarrow$ *well-formed* A $(m$ A $p)$
unfolding *electoral-module-def non-blocking-def*

```
  by simp
hence ?input-sound ⟶ elect m A p ∪ defer m A p = A − reject m A p
  using non-blocking-m elec-and-def-not-rej
  unfolding non-blocking-def
  by metis
with A-reject-diff
have ?input-sound ⟶ elect m A p ∪ defer m A p ≠ {}
  by simp
hence ?input-sound ⟶ (elect m A p ≠ {} ∨ defer m A p ≠ {})
  by simp
with non-blocking-m non-blocking-n
show ?thesis
proof (unfold non-blocking-def)
  assume
    emod-reject-m:
    electoral-module m ∧
      (∀ A p. A ≠ {} ∧ finite-profile A p ⟶ reject m A p ≠ A) and
    emod-reject-n:
    electoral-module n ∧
      (∀ A p. A ≠ {} ∧ finite-profile A p ⟶ reject n A p ≠ A)
  show
    electoral-module (m ▷ n) ∧
      (∀ A p. A ≠ {} ∧ finite-profile A p ⟶ reject (m ▷ n) A p ≠ A)
  proof (safe)
    show electoral-module (m ▷ n)
      using emod-reject-m emod-reject-n
      by simp
  next
    fix
      A :: 'a set and
      p :: 'a Profile and
      x :: 'a
    assume
      fin-A: finite A and
      prof-A: profile A p and
      rej-mn: reject (m ▷ n) A p = A and
      x-in-A: x ∈ A
    from emod-reject-m fin-A prof-A
    have fin-defer: finite-profile (defer m A p) (limit-profile (defer m A p) p)
      using def-presv-prof defer-in-alts finite-subset
      by (metis (no-types))
    from emod-reject-m emod-reject-n fin-A prof-A
    have seq-elect:
      elect (m ▷ n) A p =
        elect n (defer m A p) (limit-profile (defer m A p) p) ∪ elect m A p
      using seq-comp-def-then-elect-elec-set
      by metis
    from emod-reject-n emod-reject-m fin-A prof-A
    have def-limit:
```

```
      defer (m ▷ n) A p = defer n (defer m A p) (limit-profile (defer m A p) p)
      using seq-comp-defers-def-set
      by metis
    from emod-reject-n emod-reject-m fin-A prof-A
    have elect (m ▷ n) A p ∪ defer (m ▷ n) A p = A − reject (m ▷ n) A p
      using elec-and-def-not-rej seq-comp-sound
      by metis
    hence elect-def-disj:
      elect n (defer m A p) (limit-profile (defer m A p) p) ∪
        elect m A p ∪
        defer n (defer m A p) (limit-profile (defer m A p) p) = {}
      using def-limit seq-elect Diff-cancel rej-mn
      by auto
    have rej-def-eq-set:
      defer n (defer m A p) (limit-profile (defer m A p) p) −
        defer n (defer m A p) (limit-profile (defer m A p) p) = {} ⟶
          reject n (defer m A p) (limit-profile (defer m A p) p) =
            defer m A p
      using elect-def-disj emod-reject-n fin-defer
      by (simp add: reject-not-elec-or-def)
    have
      defer n (defer m A p) (limit-profile (defer m A p) p) −
        defer n (defer m A p) (limit-profile (defer m A p) p) = {} ⟶
          elect m A p = elect m A p ∩ defer m A p
      using elect-def-disj
      by blast
    thus x ∈ {}
      using rej-def-eq-set result-disj fin-defer Diff-cancel Diff-empty fin-A prof-A
          emod-reject-m emod-reject-n reject-not-elec-or-def x-in-A
      by metis
  qed
 qed
qed
```

Sequential composition preserves the non-electing property.

```
theorem seq-comp-presv-non-electing[simp]:
  fixes
    m :: 'a Electoral-Module and
    n :: 'a Electoral-Module
  assumes
    non-electing m and
    non-electing n
  shows non-electing (m ▷ n)
proof (unfold non-electing-def, safe)
  have electoral-module m ∧ electoral-module n
    using assms
    unfolding non-electing-def
    by blast
  thus electoral-module (m ▷ n)
```

by *simp*
next
fix
$A :: 'a\ set$ **and**
$p :: 'a\ Profile$ **and**
$x :: 'a$
assume
profile $A\ p$ **and**
$x \in elect\ (m \triangleright n)\ A\ p$
thus $x \in \{\}$
using *assms*
unfolding *non-electing-def*
using *seq-comp-def-then-elect-elec-set def-presv-prof Diff-empty Diff-partition empty-subsetI*
by *metis*
qed

Composing an electoral module that defers exactly 1 alternative in sequence after an electoral module that is electing results (still) in an electing electoral module.

theorem *seq-comp-electing*[*simp*]:
fixes
$m :: 'a\ Electoral\text{-}Module$ **and**
$n :: 'a\ Electoral\text{-}Module$
assumes
def-one-m: *defers 1 m* **and**
electing-n: *electing n*
shows *electing* $(m \triangleright n)$
proof −
have *defer-card-eq-one*:
$\forall\ A\ p.\ card\ A \geq 1 \wedge profile\ A\ p \longrightarrow card\ (defer\ m\ A\ p) = 1$
using *def-one-m card.infinite not-one-le-zero*
unfolding *defers-def*
by *metis*
hence *def-m1-not-empty*:
$\forall\ A\ p.\ A \neq \{\} \wedge finite\text{-}profile\ A\ p \longrightarrow defer\ m\ A\ p \neq \{\}$
using *One-nat-def Suc-leI card-eq-0-iff card-gt-0-iff zero-neq-one*
by *metis*
thus *?thesis*
proof −
obtain
$p :: 'a\ Electoral\text{-}Module \Rightarrow 'a\ set$ **and**
$A :: 'a\ Electoral\text{-}Module \Rightarrow 'a\ Profile$ **where**
f-mod:
$\forall\ m'\ A'\ p'.$
$(electing\ m' \longrightarrow electoral\text{-}module\ m' \wedge A' \neq \{\} \wedge finite\text{-}profile\ A'\ p'$
$\longrightarrow elect\ m'\ A'\ p' \neq \{\}) \wedge$
$(\neg\ electing\ m' \longrightarrow electoral\text{-}module\ m' \longrightarrow p\ m' \neq \{\} \wedge$
$profile\ (p\ m')\ (A\ m') \wedge elect\ m'\ (p\ m')\ (A\ m') = \{\})$

```
    unfolding electing-def
    by moura
  hence f-elect: electoral-module n
      ∧ (∀ A p. A ≠ {} ∧ finite-profile A p ⟶ elect n A p ≠ {})
    using electing-n
    unfolding electing-def
    by metis
  have def-card-one:
    electoral-module m ∧
      (∀ A p. 1 ≤ card A ∧ profile A p ⟶ card (defer m A p) = 1)
    using def-one-m defer-card-eq-one
    unfolding defers-def
    by blast
  hence electoral-module (m ▷ n)
    using f-elect seq-comp-sound
    by metis
  with f-mod f-elect def-card-one
  show ?thesis
    using seq-comp-def-then-elect-elec-set def-presv-prof defer-in-alts
        def-m1-not-empty bot-eq-sup-iff finite-subset
    unfolding electing-def
    by metis
 qed
qed

lemma def-lift-inv-seq-comp-help:
 fixes
  m :: 'a Electoral-Module and
  n :: 'a Electoral-Module and
  A :: 'a set and
  p :: 'a Profile and
  q :: 'a Profile and
  a :: 'a
 assumes
  monotone-m: defer-lift-invariance m and
  monotone-n: defer-lift-invariance n and
  def-and-lifted: a ∈ (defer (m ▷ n) A p) ∧ lifted A p q a
 shows (m ▷ n) A p = (m ▷ n) A q
proof −
 let ?new-Ap = defer m A p
 let ?new-Aq = defer m A q
 let ?new-p = limit-profile ?new-Ap p
 let ?new-q = limit-profile ?new-Aq q
 from monotone-m monotone-n
 have modules: electoral-module m ∧ electoral-module n
   unfolding defer-lift-invariance-def
   by simp
 hence profile A p ⟶ defer (m ▷ n) A p ⊆ defer m A p
   using seq-comp-def-set-bounded
```

by *metis*
moreover have *profile-p*: *lifted A p q a* $\longrightarrow$ *finite-profile A p*
unfolding *lifted-def*
by *simp*
ultimately have *defer-subset*: *defer* $(m \triangleright n)$ *A p* $\subseteq$ *defer m A p*
using *def-and-lifted*
by *blast*
hence *mono-m*: *m A p* = *m A q*
using *monotone-m def-and-lifted modules profile-p*
seq-comp-def-set-trans
unfolding *defer-lift-invariance-def*
by *metis*
hence *new-A-eq*: *?new-Ap* = *?new-Aq*
by *presburger*
have *defer-eq*: *defer* $(m \triangleright n)$ *A p* = *defer n ?new-Ap ?new-p*
using *snd-conv*
unfolding *sequential-composition.simps*
by *metis*
have *mono-n*: *n ?new-Ap ?new-p* = *n ?new-Aq ?new-q*
proof (*cases*)
assume *lifted ?new-Ap ?new-p ?new-q a*
thus *?thesis*
using *defer-eq mono-m monotone-n def-and-lifted*
unfolding *defer-lift-invariance-def*
by (*metis* (*no-types*, *lifting*))
next
assume *unlifted-a*: $\neg$*lifted ?new-Ap ?new-p ?new-q a*
from *def-and-lifted*
have *finite-profile A q*
unfolding *lifted-def*
by *simp*
with *modules new-A-eq*
have *prof-p*: *profile ?new-Ap ?new-q*
using *def-presv-prof*
by (*metis* (*no-types*))
moreover from *modules profile-p def-and-lifted*
have *prof-q*: *profile ?new-Ap ?new-p*
using *def-presv-prof*
by (*metis* (*no-types*))
moreover from *defer-subset def-and-lifted*
have *a* $\in$ *?new-Ap*
by *blast*
moreover from *def-and-lifted*
have *eql-lengths*: *length ?new-p* = *length ?new-q*
unfolding *lifted-def*
by *simp*
ultimately have *lifted-stmt*:
$(\exists\ i{::}nat.\ i < length\ ?new\text{-}p \wedge$
Preference-Relation.lifted ?new-Ap (*?new-p*!*i*) (*?new-q*!*i*) *a*) $\longrightarrow$

```
      (∃ i::nat. i < length ?new-p ∧
        ¬ Preference-Relation.lifted ?new-Ap (?new-p!i) (?new-q!i) a ∧
          (?new-p!i) ≠ (?new-q!i))
    using unlifted-a def-and-lifted defer-in-alts finite-subset modules
    unfolding lifted-def
    by (metis (no-types, lifting))
  from def-and-lifted modules
  have ∀ i. 0 ≤ i ∧ i < length ?new-p ⟶
        Preference-Relation.lifted A (p!i) (q!i) a ∨ (p!i) = (q!i)
    using limit-prof-presv-size
    unfolding Profile.lifted-def
    by metis
  with def-and-lifted modules mono-m
  have ∀ i. 0 ≤ i ∧ i < length ?new-p ⟶
        Preference-Relation.lifted ?new-Ap (?new-p!i) (?new-q!i) a ∨
          (?new-p!i) = (?new-q!i)
    using limit-lifted-imp-eq-or-lifted defer-in-alts
        limit-prof-presv-size nth-map
    unfolding Profile.lifted-def limit-profile.simps
    by (metis (no-types, lifting))
  with lifted-stmt eql-lengths mono-m
  show ?thesis
    using leI not-less-zero nth-equalityI
    by metis
qed
from mono-m mono-n
show ?thesis
  unfolding sequential-composition.simps
  by (metis (full-types))
qed
```

Sequential composition preserves the property defer-lift-invariance.

```
theorem seq-comp-presv-def-lift-inv[simp]:
  fixes
    m :: 'a Electoral-Module and
    n :: 'a Electoral-Module
  assumes
    defer-lift-invariance m and
    defer-lift-invariance n
  shows defer-lift-invariance (m ▷ n)
  using assms def-lift-inv-seq-comp-help
      seq-comp-sound defer-lift-invariance-def
  by (metis (full-types))
```

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 :: 'a Electoral-Module and
    n :: 'a Electoral-Module
  assumes
    non-blocking-m: non-blocking m and
    non-electing-m: non-electing m and
    def-one-n: defers 1 n
  shows defers 1 (m ▷ n)
proof (unfold defers-def, safe)
  have electoral-module m
    using non-electing-m
    unfolding non-electing-def
    by simp
  moreover have electoral-module n
    using def-one-n
    unfolding defers-def
    by simp
  ultimately show electoral-module (m ▷ n)
    by simp
next
  fix
    A :: 'a set and
    p :: 'a Profile
  assume
    pos-card: 1 ≤ card A and
    fin-A: finite A and
    prof-A: profile A p
  from pos-card
  have A ≠ {}
    by auto
  with fin-A prof-A
  have reject m A p ≠ A
    using non-blocking-m
    unfolding non-blocking-def
    by simp
  hence ∃ a. a ∈ A ∧ a ∉ reject m A p
    using non-electing-m reject-in-alts fin-A prof-A
    unfolding non-electing-def
    by auto
  hence defer m A p ≠ {}
    using electoral-mod-defer-elem empty-iff non-electing-m fin-A prof-A
    unfolding non-electing-def
    by (metis (no-types))
  hence card (defer m A p) ≥ 1
    using Suc-leI card-gt-0-iff fin-A prof-A non-blocking-m
          defer-in-alts infinite-super
    unfolding One-nat-def non-blocking-def
    by metis
  moreover have
    ∀ i m'. defers i m' =
```

```
    (electoral-module m' ∧
      (∀ A' p'. i ≤ card A' ∧ finite A' ∧ profile A' p' ⟶
          card (defer m' A' p') = i))
    unfolding defers-def
    by simp
  ultimately have
    card (defer n (defer m A p) (limit-profile (defer m A p) p)) = 1
    using def-one-n fin-A prof-A non-blocking-m def-presv-prof
        defer-in-alts infinite-super
    unfolding non-blocking-def
    by metis
  moreover have
    defer (m ▷ n) A p = defer n (defer m A p) (limit-profile (defer m A p) p)
    using seq-comp-defers-def-set
    by (metis (no-types, opaque-lifting))
  ultimately show card (defer (m ▷ n) A p) = 1
    by simp
qed
```

Composing a defer-lift invariant and a non-electing electoral module that defers exactly one alternative in sequence with an electing electoral module results in a monotone electoral module.

```
theorem disj-compat-seq[simp]:
  fixes
    m :: 'a Electoral-Module and
    m' :: 'a Electoral-Module and
    n :: 'a Electoral-Module
  assumes
    compatible: disjoint-compatibility m n and
    module-m': electoral-module m'
  shows disjoint-compatibility (m ▷ m') n
proof (unfold disjoint-compatibility-def, safe)
  show electoral-module (m ▷ m')
    using compatible module-m' seq-comp-sound
    unfolding disjoint-compatibility-def
    by metis
next
  show electoral-module n
    using compatible
    unfolding disjoint-compatibility-def
    by metis
next
  fix S :: 'a set
  have modules:
    electoral-module (m ▷ m') ∧ electoral-module n
    using compatible module-m' seq-comp-sound
    unfolding disjoint-compatibility-def
    by metis
  obtain A where rej-A:
```

```
  A ⊆ S ∧
    (∀ a ∈ A.
      indep-of-alt m S a ∧ (∀ p. profile S p ⟶ a ∈ reject m S p)) ∧
    (∀ a ∈ S − A.
      indep-of-alt n S a ∧ (∀ p. profile S p ⟶ a ∈ reject n S p))
  using compatible
  unfolding disjoint-compatibility-def
  by (metis (no-types, lifting))
show
  ∃ A ⊆ S.
    (∀ a ∈ A. indep-of-alt (m ▷ m′) S a ∧
      (∀ p. profile S p ⟶ a ∈ reject (m ▷ m′) S p)) ∧
    (∀ a ∈ S − A.
      indep-of-alt n S a ∧ (∀ p. profile S p ⟶ a ∈ reject n S p))
proof
  have ∀ a p q. a ∈ A ∧ equiv-prof-except-a S p q a ⟶
        (m ▷ m′) S p = (m ▷ m′) S q
  proof (safe)
    fix
      a :: 'a and
      p :: 'a Profile and
      q :: 'a Profile
    assume
      a-in-A: a ∈ A and
      lifting-equiv-p-q: equiv-prof-except-a S p q a
    hence eq-def: defer m S p = defer m S q
      using rej-A
      unfolding indep-of-alt-def
      by metis
    from lifting-equiv-p-q
    have profiles: profile S p ∧ profile S q
      unfolding equiv-prof-except-a-def
      by simp
    hence (defer m S p) ⊆ S
      using compatible defer-in-alts
      unfolding disjoint-compatibility-def
      by metis
    hence limit-profile (defer m S p) p = limit-profile (defer m S q) q
      using rej-A DiffD2 a-in-A lifting-equiv-p-q compatible defer-not-elec-or-rej
          profiles negl-diff-imp-eq-limit-prof
      unfolding disjoint-compatibility-def eq-def
      by (metis (no-types, lifting))
    with eq-def
    have m′ (defer m S p) (limit-profile (defer m S p) p) =
          m′ (defer m S q) (limit-profile (defer m S q) q)
      by simp
    moreover have m S p = m S q
      using rej-A a-in-A lifting-equiv-p-q
      unfolding indep-of-alt-def
```

```
      by metis
    ultimately show (m ▷ m') S p = (m ▷ m') S q
      unfolding sequential-composition.simps
      by (metis (full-types))
  qed
  moreover have
    ∀ a' ∈ A. ∀ p'. profile S p' ⟶ a' ∈ reject (m ▷ m') S p'
    using rej-A UnI1 prod.sel
    unfolding sequential-composition.simps
    by metis
  ultimately show
    A ⊆ S ∧
      (∀ a' ∈ A. indep-of-alt (m ▷ m') S a' ∧
        (∀ p'. profile S p' ⟶ a' ∈ reject (m ▷ m') S p')) ∧
      (∀ a' ∈ S − A. indep-of-alt n S a' ∧
        (∀ p'. profile S p' ⟶ a' ∈ reject n S p'))
    using rej-A indep-of-alt-def modules
    by (metis (mono-tags, lifting))
 qed
qed

theorem seq-comp-cond-compat[simp]:
 fixes
   m :: 'a Electoral-Module and
   n :: 'a Electoral-Module
 assumes
   dcc-m: defer-condorcet-consistency m and
   nb-n: non-blocking n and
   ne-n: non-electing n
 shows condorcet-compatibility (m ▷ n)
proof (unfold condorcet-compatibility-def, safe)
 have electoral-module m
   using dcc-m
   unfolding defer-condorcet-consistency-def
   by presburger
 moreover have electoral-module n
   using nb-n
   unfolding non-blocking-def
   by presburger
 ultimately have electoral-module (m ▷ n)
   by simp
 thus electoral-module (m ▷ n)
   by presburger
next
 fix
   A :: 'a set and
   p :: 'a Profile and
   a :: 'a
 assume
```

cw-a: *condorcet-winner A p a* **and**
a-in-rej-seq-m-n: $a \in$ *reject* $(m \triangleright n)$ *A p*
hence $\exists\ a'$. *defer-condorcet-consistency m* $\wedge$ *condorcet-winner A p a'*
using *dcc-m*
by *blast*
hence *m A p* $= (\{\}, A - ($*defer m A p*$), \{a\})$
using *defer-condorcet-consistency-def cw-a cond-winner-unique*
by (*metis* (*no-types*, *lifting*))
have *sound-m*: *electoral-module m*
using *dcc-m*
unfolding *defer-condorcet-consistency-def*
by *presburger*
moreover have *electoral-module n*
using *nb-n*
unfolding *non-blocking-def*
by *presburger*
ultimately have *sound-seq-m-n*: *electoral-module* $(m \triangleright n)$
by *simp*
have *def-m*: *defer m A p* $= \{a\}$
using *cw-a cond-winner-unique dcc-m snd-conv*
unfolding *defer-condorcet-consistency-def*
by (*metis* (*mono-tags*, *lifting*))
have *rej-m*: *reject m A p* $= A - \{a\}$
using *cw-a cond-winner-unique dcc-m prod.sel*
unfolding *defer-condorcet-consistency-def*
by (*metis* (*mono-tags*, *lifting*))
have *elect m A p* $= \{\}$
using *cw-a dcc-m defer-condorcet-consistency-def fst-conv*
by (*metis* (*mono-tags*, *lifting*))
hence *diff-elect-m*: $A -$ *elect m A p* $= A$
using *Diff-empty*
by (*metis* (*full-types*))
have *cond-win*:
finite A $\wedge$ *profile A p* $\wedge$ $a \in A \wedge (\forall\ a'.\ a' \in A - \{a'\} \longrightarrow$ *wins a p a'*$)$
using *cw-a condorcet-winner.simps DiffD2 singletonI*
by (*metis* (*no-types*))
have $\forall\ a'\ A'.\ (a'::'a) \in A' \longrightarrow$ *insert a'* $(A' - \{a'\}) = A'$
by *blast*
have *nb-n-full*:
electoral-module n $\wedge$
$(\forall\ A'\ p'.\ A' \neq \{\} \wedge$ *finite A'* $\wedge$ *profile A' p'* $\longrightarrow$ *reject n A' p'* $\neq A')$
using *nb-n non-blocking-def*
by *metis*
have *def-seq-diff*:
defer $(m \triangleright n)$ *A p* $=$ *A* $-$ *elect* $(m \triangleright n)$ *A p* $-$ *reject* $(m \triangleright n)$ *A p*
using *defer-not-elec-or-rej cond-win sound-seq-m-n*
by *metis*
have *set-ins*: $\forall\ a'\ A'.\ (a'::'a) \in A' \longrightarrow$ *insert a'* $(A' - \{a'\}) = A'$
by *fastforce*

have $\forall\ p'\ A'\ p''.\ p' = (A'::'a\ set,\ p''::'a\ set \times\ 'a\ set) \longrightarrow snd\ p' = p''$
by *simp*
hence *snd* (*elect m A p* ∪ *elect n* (*defer m A p*) (*limit-profile* (*defer m A p*) *p*),
reject m A p ∪ *reject n* (*defer m A p*) (*limit-profile* (*defer m A p*) *p*),
defer n (*defer m A p*) (*limit-profile* (*defer m A p*) *p*)) =
(*reject m A p* ∪ *reject n* (*defer m A p*) (*limit-profile* (*defer m A p*) *p*),
defer n (*defer m A p*) (*limit-profile* (*defer m A p*) *p*))
by *blast*
hence *seq-snd-simplified*:
snd ((*m* ▷ *n*) *A p*) =
(*reject m A p* ∪ *reject n* (*defer m A p*) (*limit-profile* (*defer m A p*) *p*),
defer n (*defer m A p*) (*limit-profile* (*defer m A p*) *p*))
using *sequential-composition.simps*
by *metis*
hence *seq-rej-union-eq-rej*:
reject m A p ∪ *reject n* (*defer m A p*) (*limit-profile* (*defer m A p*) *p*) =
reject (*m* ▷ *n*) *A p*
by *simp*
hence *seq-rej-union-subset-A*:
reject m A p ∪ *reject n* (*defer m A p*) (*limit-profile* (*defer m A p*) *p*) ⊆ *A*
using *sound-seq-m-n cond-win reject-in-alts*
by (*metis* (*no-types*))
hence *A* − {*a*} = *reject* (*m* ▷ *n*) *A p* − {*a*}
using *seq-rej-union-eq-rej defer-not-elec-or-rej cond-win def-m diff-elect-m*
double-diff rej-m sound-m sup-ge1
by (*metis* (*no-types*))
hence *reject* (*m* ▷ *n*) *A p* ⊆ *A* − {*a*}
using *seq-rej-union-subset-A seq-snd-simplified set-ins def-seq-diff nb-n-full*
cond-win fst-conv Diff-empty Diff-eq-empty-iff a-in-rej-seq-m-n def-m
def-presv-prof sound-m ne-n diff-elect-m insert-not-empty defer-in-alts
reject-not-elec-or-def seq-comp-def-then-elect-elec-set finite-subset
seq-comp-defers-def-set sup-bot.left-neutral
unfolding *non-electing-def*
by (*metis* (*no-types*, *lifting*))
thus *False*
using *a-in-rej-seq-m-n*
by *blast*
next
fix
A :: *'a set* **and**
p :: *'a Profile* **and**
a :: *'a* **and**
a' :: *'a*
assume
cw-a: *condorcet-winner A p a* **and**
not-cw-a': ¬ *condorcet-winner A p a'* **and**
a'-in-elect-seq-m-n: *a'* ∈ *elect* (*m* ▷ *n*) *A p*
hence ∃ *a''*. *defer-condorcet-consistency m* ∧ *condorcet-winner A p a''*
using *dcc-m*

by *blast*
hence *result-m*: $m\ A\ p = (\{},\ A - (defer\ m\ A\ p),\ \{a\})$
using *defer-condorcet-consistency-def cw-a cond-winner-unique*
by (*metis* (*no-types*, *lifting*))
have *sound-m*: *electoral-module m*
using *dcc-m*
unfolding *defer-condorcet-consistency-def*
by *presburger*
moreover have *electoral-module n*
using *nb-n*
unfolding *non-blocking-def*
by *presburger*
ultimately have *sound-seq-m-n*: *electoral-module* $(m \triangleright n)$
by *simp*
have $reject\ m\ A\ p = A - \{a\}$
using *cw-a dcc-m prod.sel result-m*
unfolding *defer-condorcet-consistency-def*
by (*metis* (*mono-tags*, *lifting*))
hence a'*-in-rej*: $a' \in reject\ m\ A\ p$
using *Diff-iff cw-a not-cw-a' a'-in-elect-seq-m-n condorcet-winner.simps elect-in-alts singleton-iff sound-seq-m-n subset-iff*
by (*metis* (*no-types*))
have $\forall\ p'\ A'\ p''.\ p' = (A'::'a\ set,\ p''::'a\ set \times\ 'a\ set) \longrightarrow snd\ p' = p''$
by *simp*
hence *m-seq-n*:
$snd\ (elect\ m\ A\ p \cup elect\ n\ (defer\ m\ A\ p)\ (limit\text{-}profile\ (defer\ m\ A\ p)\ p),$
$reject\ m\ A\ p \cup reject\ n\ (defer\ m\ A\ p)\ (limit\text{-}profile\ (defer\ m\ A\ p)\ p),$
$defer\ n\ (defer\ m\ A\ p)\ (limit\text{-}profile\ (defer\ m\ A\ p)\ p)) =$
$(reject\ m\ A\ p \cup reject\ n\ (defer\ m\ A\ p)\ (limit\text{-}profile\ (defer\ m\ A\ p)\ p),$
$defer\ n\ (defer\ m\ A\ p)\ (limit\text{-}profile\ (defer\ m\ A\ p)\ p))$
by *blast*
have $a' \in elect\ m\ A\ p$
using a'*-in-elect-seq-m-n condorcet-winner.simps cw-a def-presv-prof ne-n seq-comp-def-then-elect-elec-set sound-m sup-bot.left-neutral*
unfolding *non-electing-def*
by (*metis* (*no-types*))
hence *a-in-rej-union*:
$a \in reject\ m\ A\ p \cup reject\ n\ (defer\ m\ A\ p)\ (limit\text{-}profile\ (defer\ m\ A\ p)\ p)$
using *Diff-iff* a'*-in-rej condorcet-winner.simps cw-a reject-not-elec-or-def sound-m*
by (*metis* (*no-types*))
have *m-seq-n-full*:
$(m \triangleright n)\ A\ p =$
$(elect\ m\ A\ p \cup elect\ n\ (defer\ m\ A\ p)\ (limit\text{-}profile\ (defer\ m\ A\ p)\ p),$
$reject\ m\ A\ p \cup reject\ n\ (defer\ m\ A\ p)\ (limit\text{-}profile\ (defer\ m\ A\ p)\ p),$
$defer\ n\ (defer\ m\ A\ p)\ (limit\text{-}profile\ (defer\ m\ A\ p)\ p))$
unfolding *sequential-composition.simps*
by *metis*
have $\forall\ A'\ A''.\ (A'::'a\ set) = fst\ (A',\ A''::'a\ set)$

by *simp*
hence $a \in$ *reject* $(m \triangleright n)$ A p
using *a-in-rej-union m-seq-n m-seq-n-full*
by *presburger*
moreover have
$finite\ A \wedge profile\ A\ p \wedge a \in A \wedge (\forall\ a''.\ a'' \in A - \{a\} \longrightarrow wins\ a\ p\ a'')$
using *cw-a m-seq-n-full a′-in-elect-seq-m-n a′-in-rej ne-n sound-m*
unfolding *condorcet-winner.simps*
by *metis*
ultimately show *False*
using *a′-in-elect-seq-m-n IntI empty-iff result-disj sound-seq-m-n a′-in-rej def-presv-prof*
fst-conv m-seq-n-full ne-n non-electing-def sound-m sup-bot.right-neutral
by *metis*
next
fix
$A :: 'a\ set$ **and**
$p :: 'a\ Profile$ **and**
$a :: 'a$ **and**
$a' :: 'a$
assume
cw-a: *condorcet-winner* $A\ p\ a$ **and**
a′-in-A: $a' \in A$ **and**
not-cw-a′: $\neg$ *condorcet-winner* $A\ p\ a'$
have *reject* $m\ A\ p = A - \{a\}$
using *cw-a cond-winner-unique dcc-m prod.sel*
unfolding *defer-condorcet-consistency-def*
by (*metis* (*mono-tags*, *lifting*))
moreover have $a \neq a'$
using *cw-a not-cw-a′*
by *safe*
ultimately have $a' \in$ *reject* $m\ A\ p$
using *DiffI a′-in-A singletonD*
by (*metis* (*no-types*))
hence $a' \in$ *reject* $m\ A\ p \cup$ *reject* n (*defer* $m\ A\ p$) (*limit-profile* (*defer* $m\ A\ p$) p)
by *blast*
moreover have
$(m \triangleright n)\ A\ p =$
(*elect* $m\ A\ p \cup$ *elect* n (*defer* $m\ A\ p$) (*limit-profile* (*defer* $m\ A\ p$) p),
reject $m\ A\ p \cup$ *reject* n (*defer* $m\ A\ p$) (*limit-profile* (*defer* $m\ A\ p$) p),
defer n (*defer* $m\ A\ p$) (*limit-profile* (*defer* $m\ A\ p$) p))
unfolding *sequential-composition.simps*
by *metis*
moreover have
snd (*elect* $m\ A\ p \cup$ *elect* n (*defer* $m\ A\ p$) (*limit-profile* (*defer* $m\ A\ p$) p),
reject $m\ A\ p \cup$ *reject* n (*defer* $m\ A\ p$) (*limit-profile* (*defer* $m\ A\ p$) p),
defer n (*defer* $m\ A\ p$) (*limit-profile* (*defer* $m\ A\ p$) p)) =
(*reject* $m\ A\ p \cup$ *reject* n (*defer* $m\ A\ p$) (*limit-profile* (*defer* $m\ A\ p$) p),
defer n (*defer* $m\ A\ p$) (*limit-profile* (*defer* $m\ A\ p$) p))
using *snd-conv*

by *metis*
ultimately show $a' \in$ *reject* $(m \triangleright n)$ A p
using *fst-eqD*
by (*metis* (*no-types*))
qed

Composing a defer-condorcet-consistent electoral module in sequence with a non-blocking and non-electing electoral module results in a defer-condorcet-consistent module.

theorem *seq-comp-dcc*[*simp*]:
fixes
$m :: 'a$ *Electoral-Module* **and**
$n :: 'a$ *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* $(m \triangleright n)$
proof (*unfold defer-condorcet-consistency-def*, *safe*)
have *electoral-module* m
using *dcc-m*
unfolding *defer-condorcet-consistency-def*
by *metis*
thus *electoral-module* $(m \triangleright n)$
using *ne-n*
by (*simp add*: *non-electing-def*)
next
fix
$A :: 'a$ *set* **and**
$p :: 'a$ *Profile* **and**
$a :: 'a$
assume *cw-a*: *condorcet-winner* A p a
hence $\exists\ a'.$ *defer-condorcet-consistency* $m \wedge$ *condorcet-winner* A p a'
using *dcc-m*
by *blast*
hence *result-m*: m A $p = (\{\}, A - (\mathit{defer}\ m\ A\ p), \{a\})$
using *defer-condorcet-consistency-def cw-a cond-winner-unique*
by (*metis* (*no-types*, *lifting*))
hence *elect-m-empty*: *elect* m A $p = \{\}$
using *eq-fst-iff*
by *metis*
have *sound-m*: *electoral-module* m
using *dcc-m*
unfolding *defer-condorcet-consistency-def*
by *metis*
hence *sound-seq-m-n*: *electoral-module* $(m \triangleright n)$
using *ne-n*
by (*simp add*: *non-electing-def*)
have *defer-eq-a*: *defer* $(m \triangleright n)$ A $p = \{a\}$

proof (*safe*)
  fix a' :: $'a$
  assume *a'-in-def-seq-m-n*: $a' \in$ *defer* $(m \triangleright n)$ A p
  have $\{a\} = \{a \in A.$ *condorcet-winner* A p $a\}$
    using *cond-winner-unique cw-a*
    by *metis*
  moreover have *defer-condorcet-consistency* $m \longrightarrow$
        m A $p = (\{\}, A -$ *defer* m A $p, \{a \in A.$ *condorcet-winner* A p $a\})$
    using *cw-a*
    unfolding *defer-condorcet-consistency-def*
    by (*metis* (*no-types*))
  ultimately have *defer* m A $p = \{a\}$
    using *dcc-m snd-conv*
    by (*metis* (*no-types*, *lifting*))
  hence *defer* $(m \triangleright n)$ A $p = \{a\}$
    using *cw-a a'-in-def-seq-m-n condorcet-winner.simps empty-iff*
        seq-comp-def-set-bounded sound-m subset-singletonD nb-n
    unfolding *non-blocking-def*
    by *metis*
  thus $a' = a$
    using *a'-in-def-seq-m-n*
    by *blast*
next
  have $\exists$ a'. *defer-condorcet-consistency* $m \wedge$ *condorcet-winner* A p a'
    using *cw-a dcc-m*
    by *blast*
  hence m A $p = (\{\}, A -$ (*defer* m A p), $\{a\})$
    using *defer-condorcet-consistency-def cw-a cond-winner-unique*
    by (*metis* (*no-types*, *lifting*))
  hence *elect-m-empty*: *elect* m A $p = \{\}$
    using *eq-fst-iff*
    by *metis*
  have *profile* (*defer* m A p) (*limit-profile* (*defer* m A p) p)
    using *condorcet-winner.simps cw-a def-presv-prof sound-m*
    by (*metis* (*no-types*))
  hence *elect* n (*defer* m A p) (*limit-profile* (*defer* m A p) p) $= \{\}$
    using *ne-n non-electing-def*
    by *metis*
  hence *elect* $(m \triangleright n)$ A $p = \{\}$
    using *elect-m-empty seq-comp-def-then-elect-elec-set sup-bot.right-neutral*
    by (*metis* (*no-types*))
  moreover have *condorcet-compatibility* $(m \triangleright n)$
    using *dcc-m nb-n ne-n*
    by *simp*
  hence $a \notin$ *reject* $(m \triangleright n)$ A p
    unfolding *condorcet-compatibility-def*
    using *cw-a*
    by *metis*
  ultimately show $a \in$ *defer* $(m \triangleright n)$ A p

```
    using cw-a electoral-mod-defer-elem empty-iff
      sound-seq-m-n condorcet-winner.simps
    by metis
  qed
  have profile (defer m A p) (limit-profile (defer m A p) p)
    using condorcet-winner.simps cw-a def-presv-prof sound-m
    by (metis (no-types))
  hence elect n (defer m A p) (limit-profile (defer m A p) p) = {}
    using ne-n non-electing-def
    by metis
  hence elect (m ▷ n) A p = {}
    using elect-m-empty seq-comp-def-then-elect-elec-set sup-bot.right-neutral
    by (metis (no-types))
  moreover have def-seq-m-n-eq-a: defer (m ▷ n) A p = {a}
    using cw-a defer-eq-a
    by (metis (no-types))
  ultimately have (m ▷ n) A p = ({}, A − {a}, {a})
    using Diff-empty cw-a elect-rej-def-combination
      reject-not-elec-or-def sound-seq-m-n condorcet-winner.simps
    by (metis (no-types))
  moreover have {a' ∈ A. condorcet-winner A p a'} = {a}
    using cw-a cond-winner-unique
    by metis
  ultimately show
    (m ▷ n) A p =
     ({}, A − defer (m ▷ n) A p, {a' ∈ A. condorcet-winner A p a'})
    using def-seq-m-n-eq-a
    by metis
qed
```

Composing a defer-lift invariant and a non-electing electoral module that defers exactly one alternative in sequence with an electing electoral module results in a monotone electoral module.

```
theorem seq-comp-mono[simp]:
  fixes
    m :: 'a Electoral-Module and
    n :: 'a Electoral-Module
  assumes
    def-monotone-m: defer-lift-invariance m and
    non-ele-m: non-electing m and
    def-one-m: defers 1 m and
    electing-n: electing n
  shows monotonicity (m ▷ n)
proof (unfold monotonicity-def, safe)
  have electoral-module m
    using non-ele-m
    unfolding non-electing-def
    by simp
  moreover have electoral-module n
```

```
    using electing-n
    unfolding electing-def
    by simp
  ultimately show electoral-module (m ▷ n)
    by simp
next
  fix
    A :: 'a set and
    p :: 'a Profile and
    q :: 'a Profile and
    w :: 'a
  assume
    elect-w-in-p: w ∈ elect (m ▷ n) A p and
    lifted-w: Profile.lifted A p q w
  thus w ∈ elect (m ▷ n) A q
    unfolding lifted-def
    using seq-comp-def-then-elect lifted-w assms
    unfolding defer-lift-invariance-def
    by metis
qed
```

Composing a defer-invariant-monotone electoral module in sequence before a non-electing, defer-monotone electoral module that defers exactly 1 alternative results in a defer-lift-invariant electoral module.

```
theorem def-inv-mono-imp-def-lift-inv[simp]:
  fixes
    m :: 'a Electoral-Module and
    n :: 'a 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
  shows defer-lift-invariance (m ▷ n)
proof (unfold defer-lift-invariance-def, safe)
  have electoral-module m
    using strong-def-mon-m
    unfolding defer-invariant-monotonicity-def
    by metis
  moreover have electoral-module n
    using defers-one
    unfolding defers-def
    by metis
  ultimately show electoral-module (m ▷ n)
    by simp
next
  fix
    A :: 'a set and
    p :: 'a Profile and
```

```
  q :: 'a Profile and
  a :: 'a
assume
  defer-a-p: a ∈ defer (m ▷ n) A p and
  lifted-a: Profile.lifted A p q a
have non-electing-m: non-electing m
  using strong-def-mon-m
  unfolding defer-invariant-monotonicity-def
  by simp
have electoral-mod-m: electoral-module m
  using strong-def-mon-m
  unfolding defer-invariant-monotonicity-def
  by metis
have electoral-mod-n: electoral-module n
  using defers-one
  unfolding defers-def
  by metis
have finite-profile-p: finite-profile A p
  using lifted-a
  unfolding Profile.lifted-def
  by simp
have finite-profile-q: finite-profile A q
  using lifted-a
  unfolding Profile.lifted-def
  by simp
have 1 ≤ card A
 using Profile.lifted-def card-eq-0-iff emptyE less-one lifted-a linorder-le-less-linear
  by metis
hence n-defers-exactly-one-p: card (defer n A p) = 1
  using finite-profile-p defers-one
  unfolding defers-def
  by (metis (no-types))
have fin-prof-def-m-q: profile (defer m A q) (limit-profile (defer m A q) q)
  using def-presv-prof electoral-mod-m finite-profile-q
  by (metis (no-types))
have def-seq-m-n-q:
  defer (m ▷ n) A q = defer n (defer m A q) (limit-profile (defer m A q) q)
  using seq-comp-defers-def-set
  by simp
have prof-def-m: profile (defer m A p) (limit-profile (defer m A p) p)
  using def-presv-prof electoral-mod-m finite-profile-p
  by (metis (no-types))
hence prof-seq-comp-m-n:
  profile (defer n (defer m A p) (limit-profile (defer m A p) p))
        (limit-profile (defer n (defer m A p) (limit-profile (defer m A p) p))
          (limit-profile (defer m A p) p))
  using def-presv-prof electoral-mod-n
  by (metis (no-types))
have a-non-empty: a ∉ {}
```

by *simp*
have *def-seq-m-n*:
defer $(m \triangleright n)$ *A p* = *defer n* (*defer m A p*) (*limit-profile* (*defer m A p*) *p*)
using *seq-comp-defers-def-set*
by *simp*
have $1 \leq$ *card* (*defer n* (*defer m A p*) (*limit-profile* (*defer m A p*) *p*))
using *a-non-empty card-gt-0-iff defer-a-p electoral-mod-n prof-def-m seq-comp-defers-def-set One-nat-def Suc-leI defer-in-alts electoral-mod-m finite-profile-p finite-subset*
by (*metis* (*no-types*))
hence *card* (*defer n* (*defer n* (*defer m A p*) (*limit-profile* (*defer m A p*) *p*)) (*limit-profile* (*defer n* (*defer m A p*) (*limit-profile* (*defer m A p*) *p*)) (*limit-profile* (*defer m A p*) *p*))) = *1*
using *n-defers-exactly-one-p prof-seq-comp-m-n defers-one defer-in-alts electoral-mod-m finite-profile-p finite-subset prof-def-m*
unfolding *defers-def*
by *metis*
hence *def-seq-m-n-eq-one*: *card* (*defer* $(m \triangleright n)$ *A p*) = *1*
using *One-nat-def Suc-leI a-non-empty card-gt-0-iff def-seq-m-n defer-a-p defers-one electoral-mod-m prof-def-m finite-profile-p seq-comp-def-set-trans defer-in-alts rev-finite-subset*
unfolding *defers-def*
by *metis*
hence *def-seq-m-n-eq-a*: *defer* $(m \triangleright n)$ *A p* = $\{a\}$
using *defer-a-p is-singleton-altdef is-singleton-the-elem singletonD*
by (*metis* (*no-types*))
show $(m \triangleright n)$ *A p* = $(m \triangleright n)$ *A q*
proof (*cases*)
assume *defer m A q* $\neq$ *defer m A p*
hence *defer m A q* = $\{a\}$
using *defer-a-p electoral-mod-n finite-profile-p lifted-a seq-comp-def-set-trans strong-def-mon-m*
unfolding *defer-invariant-monotonicity-def*
by (*metis* (*no-types*))
moreover from *this*
have *a* $\in$ *defer m A p* $\longrightarrow$ *card* (*defer* $(m \triangleright n)$ *A q*) = *1*
using *card-eq-0-iff card-insert-disjoint defers-one electoral-mod-m empty-iff order-refl finite.emptyI seq-comp-defers-def-set def-presv-prof finite-profile-q finite.insertI*
unfolding *One-nat-def defers-def*
by *metis*
moreover have *a* $\in$ *defer m A p*
using *electoral-mod-m electoral-mod-n defer-a-p seq-comp-def-set-bounded finite-profile-p finite-profile-q*
by *blast*
ultimately have *defer* $(m \triangleright n)$ *A q* = $\{a\}$
using *Collect-mem-eq card-1-singletonE empty-Collect-eq insertCI subset-singletonD def-seq-m-n-q defer-in-alts electoral-mod-n fin-prof-def-m-q*
by (*metis* (*no-types*, *lifting*))

hence *defer* $(m \triangleright n)$ A p = *defer* $(m \triangleright n)$ A q
using *def-seq-m-n-eq-a*
by *presburger*
moreover have *elect* $(m \triangleright n)$ A p = *elect* $(m \triangleright n)$ A q
using *prof-def-m fin-prof-def-m-q finite-profile-p finite-profile-q non-electing-def non-electing-m non-electing-n seq-comp-def-then-elect-elec-set*
by *metis*
ultimately show *?thesis*
using *electoral-mod-m electoral-mod-n eq-def-and-elect-imp-eq finite-profile-p finite-profile-q seq-comp-sound*
by (*metis* (*no-types*))
next
assume ¬ (*defer m A q* ≠ *defer m A p*)
hence *def-eq*: *defer m A q* = *defer m A p*
by *presburger*
have *elect m A p* = {}
using *finite-profile-p non-electing-m*
unfolding *non-electing-def*
by *simp*
moreover have *elect m A q* = {}
using *finite-profile-q non-electing-m*
unfolding *non-electing-def*
by *simp*
ultimately have *elect-m-equal*: *elect m A p* = *elect m A q*
by *simp*
have (*limit-profile* (*defer m A p*) *p*) = (*limit-profile* (*defer m A p*) *q*) ∨ *lifted* (*defer m A q*) (*limit-profile* (*defer m A p*) *p*) (*limit-profile* (*defer m A p*) *q*) *a*
using *def-eq defer-in-alts electoral-mod-m lifted-a finite-profile-q limit-prof-eq-or-lifted*
by *metis*
hence *defer* $(m \triangleright n)$ A p = *defer* $(m \triangleright n)$ A q
using *a-non-empty card-1-singletonE def-eq def-seq-m-n def-seq-m-n-q defer-a-p defer-monotone-n defer-monotonicity-def defer-seq-m-n-eq-one defers-one electoral-mod-m fin-prof-def-m-q finite-profile-p insertE seq-comp-def-card-bounded lifted-def*
unfolding *defers-def*
by (*metis* (*no-types*, *lifting*))
moreover from *this*
have *reject* $(m \triangleright n)$ A p = *reject* $(m \triangleright n)$ A q
using *electoral-mod-m electoral-mod-n finite-profile-p finite-profile-q non-electing-def non-electing-m non-electing-n eq-def-and-elect-imp-eq seq-comp-presv-non-electing*
by (*metis* (*no-types*))
ultimately have *snd* ((*m* ▷ *n*) *A p*) = *snd* ((*m* ▷ *n*) *A q*)
using *prod-eqI*
by *metis*
moreover have *elect* $(m \triangleright n)$ A p = *elect* $(m \triangleright n)$ A q
using *prof-def-m fin-prof-def-m-q non-electing-n finite-profile-p finite-profile-q non-electing-def def-eq elect-m-equal fst-conv*
unfolding *sequential-composition.simps*

```
    by (metis (no-types))
  ultimately show (m ▷ n) A p = (m ▷ n) A q
    using prod-eqI
    by metis
 qed
qed

end
```

4.4 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.

4.4.1 Definition

```
fun parallel-composition :: 'a Electoral-Module ⇒ 'a Electoral-Module ⇒
      'a Aggregator ⇒ 'a Electoral-Module where
 parallel-composition m n agg A p = agg A (m A p) (n A p)

abbreviation parallel :: 'a Electoral-Module ⇒ 'a Aggregator ⇒
      'a Electoral-Module ⇒ 'a Electoral-Module
    (- ‖- - [50, 1000, 51] 50) where
 m ‖a n == parallel-composition m n a
```

4.4.2 Soundness

```
theorem par-comp-sound[simp]:
 fixes
   m :: 'a Electoral-Module and
   n :: 'a Electoral-Module and
   a :: 'a Aggregator
 assumes
   electoral-module m and
   electoral-module n and
   aggregator a
 shows electoral-module (m ‖a n)
proof (unfold electoral-module-def, safe)
 fix
```

```
    A :: 'a set and
    p :: 'a Profile
  assume profile A p
  moreover have
    ∀ a'. aggregator a' =
      (∀ A' e r d e' r' d'.
        well-formed (A'::'a set) (e, r', d) ∧ well-formed A' (r, d', e') ⟶
          well-formed A' (a' A' (e, r', d) (r, d', e')))
    unfolding aggregator-def
    by blast
  moreover have
    ∀ m' A' p'.
      electoral-module m' ∧ finite (A'::'a set) ∧ profile A' p' ⟶
          well-formed A' (m' A' p')
    using par-comp-result-sound
    by (metis (no-types))
  ultimately have well-formed A (a A (m A p) (n A p))
    using elect-rej-def-combination assms par-comp-result-sound
    by metis
  thus well-formed A ((m ∥a n) A p)
    by simp
qed
```

4.4.3 Composition Rule

Using a conservative aggregator, the parallel composition preserves the property non-electing.

```
theorem conserv-agg-presv-non-electing[simp]:
  fixes
    m :: 'a Electoral-Module and
    n :: 'a Electoral-Module and
    a :: 'a Aggregator
  assumes
    non-electing-m: non-electing m and
    non-electing-n: non-electing n and
    conservative: agg-conservative a
  shows non-electing (m ∥a n)
proof (unfold non-electing-def, safe)
  have electoral-module m
    using non-electing-m
    unfolding non-electing-def
    by simp
  moreover have electoral-module n
    using non-electing-n
    unfolding non-electing-def
    by simp
  moreover have aggregator a
    using conservative
    unfolding agg-conservative-def
```

by *simp*
ultimately show *electoral-module* $(m \parallel_a n)$
using *par-comp-sound*
by *simp*
next
fix
$A :: 'a\ set$ **and**
$p :: 'a\ Profile$ **and**
$w :: 'a$
assume
prof-A: *profile* $A\ p$ **and**
w-wins: $w \in elect\ (m \parallel_a n)\ A\ p$
have *emod-m*: *electoral-module* m
using *non-electing-m*
unfolding *non-electing-def*
by *simp*
have *emod-n*: *electoral-module* n
using *non-electing-n*
unfolding *non-electing-def*
by *simp*
have $\forall\ r\ r'\ d\ d'\ e\ e'\ A'\ f.$
$(well\text{-}formed\ (A'::'a\ set)\ (e',\ r',\ d') \wedge well\text{-}formed\ A'\ (e,\ r,\ d) \longrightarrow$
$elect\text{-}r\ (f\ A'\ (e',\ r',\ d')\ (e,\ r,\ d)) \subseteq e' \cup e \wedge$
$reject\text{-}r\ (f\ A'\ (e',\ r',\ d')\ (e,\ r,\ d)) \subseteq r' \cup r \wedge$
$defer\text{-}r\ (f\ A'\ (e',\ r',\ d')\ (e,\ r,\ d)) \subseteq d' \cup d) =$
$(well\text{-}formed\ A'\ (e',\ r',\ d') \wedge well\text{-}formed\ A'\ (e,\ r,\ d) \longrightarrow$
$elect\text{-}r\ (f\ A'\ (e',\ r',\ d')\ (e,\ r,\ d)) \subseteq e' \cup e \wedge$
$reject\text{-}r\ (f\ A'\ (e',\ r',\ d')\ (e,\ r,\ d)) \subseteq r' \cup r \wedge$
$defer\text{-}r\ (f\ A'\ (e',\ r',\ d')\ (e,\ r,\ d)) \subseteq d' \cup d)$
by *linarith*
hence $\forall\ a'.\ agg\text{-}conservative\ a' =$
$(aggregator\ a' \wedge$
$(\forall\ A'\ e\ e'\ d\ d'\ r\ r'.$
$well\text{-}formed\ (A'::'a\ set)\ (e,\ r,\ d) \wedge well\text{-}formed\ A'\ (e',\ r',\ d') \longrightarrow$
$elect\text{-}r\ (a'\ A'\ (e,\ r,\ d)\ (e',\ r',\ d')) \subseteq e \cup e' \wedge$
$reject\text{-}r\ (a'\ A'\ (e,\ r,\ d)\ (e',\ r',\ d')) \subseteq r \cup r' \wedge$
$defer\text{-}r\ (a'\ A'\ (e,\ r,\ d)\ (e',\ r',\ d')) \subseteq d \cup d'))$
unfolding *agg-conservative-def*
by *simp*
hence $aggregator\ a \wedge$
$(\forall\ A'\ e\ e'\ d\ d'\ r\ r'.$
$well\text{-}formed\ A'\ (e,\ r,\ d) \wedge well\text{-}formed\ A'\ (e',\ r',\ d') \longrightarrow$
$elect\text{-}r\ (a\ A'\ (e,\ r,\ d)\ (e',\ r',\ d')) \subseteq e \cup e' \wedge$
$reject\text{-}r\ (a\ A'\ (e,\ r,\ d)\ (e',\ r',\ d')) \subseteq r \cup r' \wedge$
$defer\text{-}r\ (a\ A'\ (e,\ r,\ d)\ (e',\ r',\ d')) \subseteq d \cup d')$
using *conservative*
by *presburger*
hence $let\ c = (a\ A\ (m\ A\ p)\ (n\ A\ p))\ in$
$elect\text{-}r\ c \subseteq (elect\ m\ A\ p) \cup (elect\ n\ A\ p)$

using *emod-m emod-n par-comp-result-sound*
prod.collapse prof-A
by *metis*
hence $w \in (elect\ m\ A\ p) \cup (elect\ n\ A\ p)$
using *w-wins*
by *auto*
thus $w \in \{\}$
using *sup-bot-right prof-A non-electing-m non-electing-n*
unfolding *non-electing-def*
by (*metis* (*no-types*, *lifting*))
qed

end

4.5 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).

4.5.1 Definition

lemma *loop-termination-helper*:
fixes
$m :: 'a\ Electoral\text{-}Module$ **and**
$t :: 'a\ Termination\text{-}Condition$ **and**
$acc :: 'a\ Electoral\text{-}Module$ **and**
$A :: 'a\ set$ **and**
$p :: 'a\ Profile$
assumes
$\neg\ t\ (acc\ A\ p)$ **and**
$defer\ (acc \triangleright m)\ A\ p \subset defer\ acc\ A\ p$ **and**
$finite\ (defer\ acc\ A\ p)$
shows $((acc \triangleright m, m, t, A, p), (acc, m, t, A, p)) \in$
$measure\ (\lambda\ (acc, m, t, A, p).\ card\ (defer\ acc\ A\ p))$

using *assms psubset-card-mono*
by *simp*

This function handles the accumulator for the following loop composition function.

function *loop-comp-helper* ::
'a Electoral-Module ⇒ *'a Electoral-Module* ⇒
'a Termination-Condition ⇒ *'a Electoral-Module* **where**
finite (*defer acc A p*) ∧ (*defer* (*acc* ▷ *m*) *A p*) ⊂ (*defer acc A p*)
⟶ *t* (*acc A p*) ⟹
loop-comp-helper acc m t A p = *acc A p* |
¬ (*finite* (*defer acc A p*) ∧ (*defer* (*acc* ▷ *m*) *A p*) ⊂ (*defer acc A p*)
⟶ *t* (*acc A p*)) ⟹
loop-comp-helper acc m t A p = *loop-comp-helper* (*acc* ▷ *m*) *m t A p*
proof −
fix
P :: *bool* **and**
accum ::
'a Electoral-Module × *'a Electoral-Module* × *'a Termination-Condition* ×
'a set × *'a Profile*
have *accum-exists*: ∃ *m n t A p*. (*m*, *n*, *t*, *A*, *p*) = *accum*
using *prod-cases5*
by *metis*
assume
⋀ *acc A p m t*.
finite (*defer acc A p*) ∧ *defer* (*acc* ▷ *m*) *A p* ⊂ *defer acc A p*
⟶ *t* (*acc A p*) ⟹ *accum* = (*acc*, *m*, *t*, *A*, *p*) ⟹ *P* **and**
⋀ *acc A p m t*.
¬ (*finite* (*defer acc A p*) ∧ *defer* (*acc* ▷ *m*) *A p* ⊂ *defer acc A p*
⟶ *t* (*acc A p*)) ⟹ *accum* = (*acc*, *m*, *t*, *A*, *p*) ⟹ *P*
thus *P*
using *accum-exists*
by (*metis* (*no-types*))
next
fix
t :: *'b Termination-Condition* **and**
acc :: *'b Electoral-Module* **and**
A :: *'b set* **and**
p :: *'b Profile* **and**
m :: *'b Electoral-Module* **and**
t' :: *'b Termination-Condition* **and**
acc' :: *'b Electoral-Module* **and**
A' :: *'b set* **and**
p' :: *'b Profile* **and**
m' :: *'b Electoral-Module*
assume
finite (*defer acc A p*) ∧ *defer* (*acc* ▷ *m*) *A p* ⊂ *defer acc A p*
⟶ *t* (*acc A p*) **and**
finite (*defer acc' A' p'*) ∧ *defer* (*acc'* ▷ *m'*) *A' p'* ⊂ *defer acc' A' p'*

$\longrightarrow t'\ (acc'\ A'\ p')$ **and**
$(acc,\ m,\ t,\ A,\ p) = (acc',\ m',\ t',\ A',\ p')$
thus $acc\ A\ p = acc'\ A'\ p'$
by *fastforce*
next
fix
$t :: 'a$ *Termination-Condition* **and**
$acc :: 'a$ *Electoral-Module* **and**
$A :: 'a$ *set* **and**
$p :: 'a$ *Profile* **and**
$m :: 'a$ *Electoral-Module* **and**
$t' :: 'a$ *Termination-Condition* **and**
$acc' :: 'a$ *Electoral-Module* **and**
$A' :: 'a$ *set* **and**
$p' :: 'a$ *Profile* **and**
$m' :: 'a$ *Electoral-Module*
assume
$finite\ (defer\ acc\ A\ p) \wedge defer\ (acc \triangleright m)\ A\ p \subset defer\ acc\ A\ p$
$\longrightarrow t\ (acc\ A\ p)$ **and**
$\neg\ (finite\ (defer\ acc'\ A'\ p') \wedge defer\ (acc' \triangleright m')\ A'\ p' \subset defer\ acc'\ A'\ p'$
$\longrightarrow t'\ (acc'\ A'\ p'))$ **and**
$(acc,\ m,\ t,\ A,\ p) = (acc',\ m',\ t',\ A',\ p')$
thus $acc\ A\ p = loop\text{-}comp\text{-}helper\text{-}sumC\ (acc' \triangleright m',\ m',\ t',\ A',\ p')$
by *force*
next
fix
$t :: 'a$ *Termination-Condition* **and**
$acc :: 'a$ *Electoral-Module* **and**
$A :: 'a$ *set* **and**
$p :: 'a$ *Profile* **and**
$m :: 'a$ *Electoral-Module* **and**
$t' :: 'a$ *Termination-Condition* **and**
$acc' :: 'a$ *Electoral-Module* **and**
$A' :: 'a$ *set* **and**
$p' :: 'a$ *Profile* **and**
$m' :: 'a$ *Electoral-Module*
assume
$\neg\ (finite\ (defer\ acc\ A\ p) \wedge defer\ (acc \triangleright m)\ A\ p \subset defer\ acc\ A\ p$
$\longrightarrow t\ (acc\ A\ p))$ **and**
$\neg\ (finite\ (defer\ acc'\ A'\ p') \wedge defer\ (acc' \triangleright m')\ A'\ p' \subset defer\ acc'\ A'\ p'$
$\longrightarrow t'\ (acc'\ A'\ p'))$ **and**
$(acc,\ m,\ t,\ A,\ p) = (acc',\ m',\ t',\ A',\ p')$
thus $loop\text{-}comp\text{-}helper\text{-}sumC\ (acc \triangleright m,\ m,\ t,\ A,\ p) =$
$loop\text{-}comp\text{-}helper\text{-}sumC\ (acc' \triangleright m',\ m',\ t',\ A',\ p')$
by *force*
qed
termination
proof (*safe*)
fix

```
    m :: 'a Electoral-Module and
    n :: 'a Electoral-Module and
    t :: 'a Termination-Condition and
    A :: 'a set and
    p :: 'a Profile
  have term-rel:
    ∃ R. wf R ∧
       (finite (defer m A p) ∧ defer (m ▷ n) A p ⊂ defer m A p ⟶ t (m A p) ∨
         ((m ▷ n, n, t, A, p), (m, n, t, A, p)) ∈ R)
    using loop-termination-helper wf-measure termination
    by (metis (no-types))
  obtain
    R :: ((('a Electoral-Module) × ('a Electoral-Module) ×
           ('a Termination-Condition) × 'a set × 'a Profile) ×
           ('a Electoral-Module) × ('a Electoral-Module) ×
           ('a Termination-Condition) × 'a set × 'a Profile) set where
    wf R ∧
      (finite (defer m A p) ∧ defer (m ▷ n) A p ⊂ defer m A p ⟶ t (m A p) ∨
          ((m ▷ n, n, t, A, p), m, n, t, A, p) ∈ R)
    using term-rel
    by presburger
  have ∀ R'.
    All (loop-comp-helper-dom :: 'a Electoral-Module × 'a Electoral-Module
      × 'a Termination-Condition × - set × (- × -) set list ⇒ bool) ∨
      (∃ t' m' A' p' n'. wf R' ⟶
        ((m' ▷ n', n', t', A'::'a set, p'), m', n', t', A', p') ∉ R' ∧
          finite (defer m' A' p') ∧ defer (m' ▷ n') A' p' ⊂ defer m' A' p' ∧
            ¬ t' (m' A' p'))
    using termination
    by metis
  thus loop-comp-helper-dom (m, n, t, A, p)
    using loop-termination-helper wf-measure
    by (metis (no-types))
qed

lemma loop-comp-code-helper[code]:
  fixes
    m :: 'a Electoral-Module and
    t :: 'a Termination-Condition and
    acc :: 'a Electoral-Module and
    A :: 'a set and
    p :: 'a Profile
  shows
    loop-comp-helper acc m t A p =
      (if finite (defer acc A p) ∧ defer (acc ▷ m) A p ⊂ defer acc A p
          ⟶ t (acc A p)
      then acc A p else loop-comp-helper (acc ▷ m) m t A p)
  by simp
```

function *loop-composition* ::
  'a Electoral-Module ⇒ *'a Termination-Condition* ⇒ *'a Electoral-Module* **where**
 t ({}, {}, *A*) ⟹ *loop-composition m t A p* = *defer-module A p* |
 ¬(*t* ({}, {}, *A*)) ⟹ *loop-composition m t A p* = (*loop-comp-helper m m t*) *A p*
 by (*fastforce*, *simp-all*)
termination
 using *termination wf-empty*
 by *blast*

abbreviation *loop* ::
 'a Electoral-Module ⇒ *'a Termination-Condition* ⇒ *'a Electoral-Module*
  (- ↻- *50*) **where**
 m ↻*t* ≡ *loop-composition m t*

lemma *loop-comp-code*[*code*]:
 fixes
  m :: *'a Electoral-Module* **and**
  t :: *'a Termination-Condition* **and**
  A :: *'a set* **and**
  p :: *'a Profile*
 shows *loop-composition m t A p* =
       (*if* (*t* ({},{},*A*))
         then (*defer-module A p*) *else* (*loop-comp-helper m m t*) *A p*)
 by *simp*

lemma *loop-comp-helper-imp-partit*:
 fixes
  m :: *'a Electoral-Module* **and**
  t :: *'a Termination-Condition* **and**
  acc :: *'a Electoral-Module* **and**
  A :: *'a set* **and**
  p :: *'a Profile* **and**
  n :: *nat*
 assumes
  module-m: *electoral-module m* **and**
  profile: *profile A p* **and**
  module-acc: *electoral-module acc* **and**
  defer-card-n: *n* = *card* (*defer acc A p*)
 shows *well-formed A* (*loop-comp-helper acc m t A p*)
 using *assms*
proof (*induct arbitrary*: *acc rule*: *less-induct*)
 case (*less*)
 have ∀ *m' n'*.
  electoral-module m' ∧ *electoral-module n'* ⟶ *electoral-module* (*m'* ▷ *n'*)
  by *auto*
 hence *electoral-module* (*acc* ▷ *m*)
  using *less.prems module-m*
  by *metis*
 hence ¬ *t* (*acc A p*) ∧ *defer* (*acc* ▷ *m*) *A p* ⊂ *defer acc A p* ⟶

well-formed A (*loop-comp-helper acc m t A p*)
using *less loop-comp-helper.simps psubset-card-mono par-comp-result-sound*
by (*metis* (*no-types*, *lifting*))
moreover have *well-formed A* (*acc A p*)
using *less.prems profile*
unfolding *electoral-module-def*
by *blast*
ultimately show *?case*
using *loop-comp-code-helper*
by (*metis* (*no-types*))
qed

4.5.2 Soundness

theorem *loop-comp-sound*:
fixes
m :: *'a Electoral-Module* **and**
t :: *'a Termination-Condition*
assumes *electoral-module m*
shows *electoral-module* (*m* $\circlearrowleft_t$)
using *def-mod-sound loop-comp-helper-imp-partit assms*
loop-composition.simps electoral-module-def
by *metis*

lemma *loop-comp-helper-imp-no-def-incr*:
fixes
m :: *'a Electoral-Module* **and**
t :: *'a Termination-Condition* **and**
acc :: *'a Electoral-Module* **and**
A :: *'a set* **and**
p :: *'a Profile* **and**
n :: *nat*
assumes
module-m: *electoral-module m* **and**
profile: *profile A p* **and**
mod-acc: *electoral-module acc* **and**
card-n-defer-acc: *n* = *card* (*defer acc A p*)
shows *defer* (*loop-comp-helper acc m t*) *A p* ⊆ *defer acc A p*
using *assms*
proof (*induct arbitrary*: *acc rule*: *less-induct*)
case (*less*)
have *emod-acc-m*: *electoral-module* (*acc* ▷ *m*)
using *less.prems module-m*
by *simp*
have ∀ *A A'*. *finite A* ∧ *A'* ⊂ *A* ⟶ *card A'* < *card A*
using *psubset-card-mono*
by *metis*
hence ¬ *t* (*acc A p*) ∧ *defer* (*acc* ▷ *m*) *A p* ⊂ *defer acc A p* ∧
finite (*defer acc A p*) ⟶

defer (*loop-comp-helper* (*acc* ▷ *m*) *m t*) *A p* ⊆ *defer acc A p*
using *emod-acc-m less.hyps less.prems*
by *blast*
hence ¬ *t* (*acc A p*) ∧ *defer* (*acc* ▷ *m*) *A p* ⊂ *defer acc A p* ∧
finite (*defer acc A p*) ⟶
defer (*loop-comp-helper acc m t*) *A p* ⊆ *defer acc A p*
by *simp*
thus *?case*
using *eq-iff loop-comp-code-helper*
by (*metis* (*no-types*))
qed

4.5.3 Lemmas

lemma *loop-comp-helper-def-lift-inv-helper*:
fixes
m :: *'a Electoral-Module* **and**
t :: *'a Termination-Condition* **and**
acc :: *'a Electoral-Module* **and**
A :: *'a set* **and**
p :: *'a Profile* **and**
n :: *nat*
assumes
monotone-m: *defer-lift-invariance m* **and**
prof: *profile A p* **and**
dli-acc: *defer-lift-invariance acc* **and**
card-n-defer: *n* = *card* (*defer acc A p*) **and**
defer-finite: *finite* (*defer acc A p*)
shows
∀ *q a*. *a* ∈ (*defer* (*loop-comp-helper acc m t*) *A p*) ∧ *lifted A p q a* ⟶
(*loop-comp-helper acc m t*) *A p* = (*loop-comp-helper acc m t*) *A q*
using *assms*
proof (*induct n arbitrary*: *acc rule*: *less-induct*)
case (*less n*)
have *defer-card-comp*:
defer-lift-invariance acc ⟶
(∀ *q a*. *a* ∈ (*defer* (*acc* ▷ *m*) *A p*) ∧ *lifted A p q a* ⟶
card (*defer* (*acc* ▷ *m*) *A p*) = *card* (*defer* (*acc* ▷ *m*) *A q*))
using *monotone-m def-lift-inv-seq-comp-help*
by *metis*
have *defer-lift-invariance acc* ⟶
(∀ *q a*. *a* ∈ (*defer* (*acc*) *A p*) ∧ *lifted A p q a* ⟶
card (*defer* (*acc*) *A p*) = *card* (*defer* (*acc*) *A q*))
unfolding *defer-lift-invariance-def*
by *simp*
hence *defer-card-acc*:
defer-lift-invariance acc ⟶
(∀ *q a*. (*a* ∈ (*defer* (*acc* ▷ *m*) *A p*) ∧ *lifted A p q a*) ⟶
card (*defer* (*acc*) *A p*) = *card* (*defer* (*acc*) *A q*))

using *assms seq-comp-def-set-trans*
unfolding *defer-lift-invariance-def*
by *metis*
thus *?case*
proof (*cases*)
assume *card-unchanged*: *card* (*defer* (*acc* ▷ *m*) *A p*) = *card* (*defer acc A p*)
have *defer-lift-invariance* (*acc*) ⟶
(∀ *q a*. *a* ∈ (*defer* (*acc*) *A p*) ∧ *lifted A p q a* ⟶
(*loop-comp-helper acc m t*) *A q* = *acc A q*)
proof (*safe*)
fix
q :: *'a Profile* **and**
a :: *'a*
assume
dli-acc: *defer-lift-invariance acc* **and**
a-in-def-acc: *a* ∈ *defer acc A p* **and**
lifted-A: *Profile.lifted A p q a*
moreover have *electoral-module m*
using *monotone-m*
unfolding *defer-lift-invariance-def*
by *simp*
moreover have *emod-acc*: *electoral-module acc*
using *dli-acc*
unfolding *defer-lift-invariance-def*
by *simp*
moreover have *acc-eq-pq*: *acc A q* = *acc A p*
using *a-in-def-acc dli-acc lifted-A*
unfolding *defer-lift-invariance-def*
by (*metis* (*full-types*))
ultimately have *finite* (*defer acc A p*)
⟶ *loop-comp-helper acc m t A q* = *acc A q*
using *card-unchanged defer-card-comp prof loop-comp-code-helper*
psubset-card-mono dual-order.strict-iff-order
seq-comp-def-set-bounded less
by (*metis* (*mono-tags*, *lifting*))
thus *loop-comp-helper acc m t A q* = *acc A q*
using *acc-eq-pq loop-comp-code-helper*
by (*metis* (*full-types*))
qed
moreover from *card-unchanged*
have (*loop-comp-helper acc m t*) *A p* = *acc A p*
using *loop-comp-code-helper order.strict-iff-order psubset-card-mono*
by *metis*
ultimately have
defer-lift-invariance (*acc* ▷ *m*) ∧ *defer-lift-invariance acc* ⟶
(∀ *q a*. *a* ∈ (*defer* (*loop-comp-helper acc m t*) *A p*) ∧ *lifted A p q a* ⟶
(*loop-comp-helper acc m t*) *A p* = (*loop-comp-helper acc m t*) *A q*)
unfolding *defer-lift-invariance-def*
by *metis*

```
  moreover have defer-lift-invariance (acc ▷ m)
    using less monotone-m seq-comp-presv-def-lift-inv
    by simp
  ultimately show ?thesis
    using less monotone-m
    by metis
next
  assume card-changed: ¬ (card (defer (acc ▷ m) A p) = card (defer acc A p))
  with prof seq-comp-def-card-bounded
  have card-smaller-for-p:
    electoral-module (acc) ∧ finite A ⟶
      card (defer (acc ▷ m) A p) < card (defer acc A p)
    using monotone-m order.not-eq-order-implies-strict
    unfolding defer-lift-invariance-def
    by (metis (full-types))
  with defer-card-acc defer-card-comp
  have card-changed-for-q:
    defer-lift-invariance (acc) ⟶
        (∀ q a. a ∈ (defer (acc ▷ m) A p) ∧ lifted A p q a ⟶
            card (defer (acc ▷ m) A q) < card (defer acc A q))
    using lifted-def less
    unfolding defer-lift-invariance-def
    by (metis (no-types, lifting))
  thus ?thesis
  proof (cases)
    assume t-not-satisfied-for-p: ¬ t (acc A p)
    hence t-not-satisfied-for-q:
      defer-lift-invariance (acc) ⟶
          (∀ q a. a ∈ (defer (acc ▷ m) A p) ∧ lifted A p q a ⟶ ¬ t (acc A q))
      using monotone-m prof seq-comp-def-set-trans
      unfolding defer-lift-invariance-def
      by metis
    have dli-card-def:
      defer-lift-invariance (acc ▷ m) ∧ defer-lift-invariance (acc) ⟶
          (∀ q a. a ∈ (defer (acc ▷ m) A p) ∧ Profile.lifted A p q a ⟶
              card (defer (acc ▷ m) A q) ≠ (card (defer acc A q)))
    proof −
      have
        ∀ m'.
          (¬ defer-lift-invariance m' ∧ electoral-module m' ⟶
            (∃ A' p' q' a.
              m' A' p' ≠ m' A' q' ∧ lifted A' p' q' a ∧ a ∈ defer m' A' p')) ∧
          (defer-lift-invariance m' ⟶
            electoral-module m' ∧
              (∀ A' p' q' a.
                m' A' p' ≠ m' A' q' ⟶ lifted A' p' q' a ⟶ a ∉ defer m' A' p'))
        unfolding defer-lift-invariance-def
        by blast
      thus ?thesis
```

```
    using card-changed monotone-m prof seq-comp-def-set-trans
    by (metis (no-types, opaque-lifting))
qed
hence dli-def-subset:
  defer-lift-invariance (acc ▷ m) ∧ defer-lift-invariance (acc) ⟶
      (∀ p' a. a ∈ (defer (acc ▷ m) A p) ∧ lifted A p p' a ⟶
          defer (acc ▷ m) A p' ⊂ defer acc A p')
  using Profile.lifted-def dli-card-def defer-lift-invariance-def
        monotone-m psubsetI seq-comp-def-set-bounded
  by (metis (no-types, opaque-lifting))
with t-not-satisfied-for-p
have rec-step-q:
  defer-lift-invariance (acc ▷ m) ∧ defer-lift-invariance (acc) ⟶
      (∀ q a. a ∈ (defer (acc ▷ m) A p) ∧ lifted A p q a ⟶
          loop-comp-helper acc m t A q = loop-comp-helper (acc ▷ m) m t A q)
proof (safe)
  fix
    q :: 'a Profile and
    a :: 'a
  assume
    a-in-def-impl-def-subset:
    ∀ q' a'. a' ∈ defer (acc ▷ m) A p ∧ lifted A p q' a' ⟶
      defer (acc ▷ m) A q' ⊂ defer acc A q' and
    dli-acc: defer-lift-invariance acc and
    a-in-def-seq-acc-m: a ∈ defer (acc ▷ m) A p and
    lifted-pq-a: lifted A p q a
  hence defer (acc ▷ m) A q ⊂ defer acc A q
    by metis
  moreover have electoral-module acc
    using dli-acc
    unfolding defer-lift-invariance-def
    by simp
  moreover have ¬ t (acc A q)
    using dli-acc a-in-def-seq-acc-m lifted-pq-a t-not-satisfied-for-q
    by metis
  ultimately show loop-comp-helper acc m t A q
                  = loop-comp-helper (acc ▷ m) m t A q
    using loop-comp-code-helper defer-in-alts finite-subset lifted-pq-a
    unfolding lifted-def
    by (metis (mono-tags, lifting))
qed
have rec-step-p:
  electoral-module acc ⟶
      loop-comp-helper acc m t A p = loop-comp-helper (acc ▷ m) m t A p
proof (safe)
  assume emod-acc: electoral-module acc
  have sound-imp-defer-subset:
    electoral-module m ⟶ defer (acc ▷ m) A p ⊆ defer acc A p
    using emod-acc prof seq-comp-def-set-bounded
```

by *blast*
hence *card-ineq*: *card* (*defer* (*acc* ▷ *m*) *A p*) < *card* (*defer acc A p*)
using *card-changed card-mono less order-neq-le-trans*
unfolding *defer-lift-invariance-def*
by *metis*
have *def-limited-acc*:
profile (*defer acc A p*) (*limit-profile* (*defer acc A p*) *p*)
using *def-presv-prof emod-acc prof*
by *metis*
have *defer* (*acc* ▷ *m*) *A p* ⊆ *defer acc A p*
using *sound-imp-defer-subset defer-lift-invariance-def monotone-m*
by *blast*
hence *defer* (*acc* ▷ *m*) *A p* ⊂ *defer acc A p*
using *def-limited-acc card-ineq card-psubset less*
by *metis*
with *def-limited-acc*
show *loop-comp-helper acc m t A p* = *loop-comp-helper* (*acc* ▷ *m*) *m t A p*
using *loop-comp-code-helper t-not-satisfied-for-p less*
by (*metis* (*no-types*))
qed
show *?thesis*
proof (*safe*)
fix
q :: *'a Profile* **and**
a :: *'a*
assume
a-in-defer-lch: *a* ∈ *defer* (*loop-comp-helper acc m t*) *A p* **and**
a-lifted: *Profile.lifted A p q a*
have *mod-acc*: *electoral-module acc*
using *defer-lift-invariance-def less*
by *blast*
hence *loop-comp-equiv*:
loop-comp-helper acc m t A p = *loop-comp-helper* (*acc* ▷ *m*) *m t A p*
using *rec-step-p*
by *blast*
moreover have *defer-lift-invariance* (*acc* ▷ *m*) ∧ *a* ∈ *defer* (*acc* ▷ *m*) *A p*
using *rec-step-p subsetD loop-comp-helper-imp-no-def-incr monotone-m*
a-in-defer-lch defer-lift-invariance-def dli-acc prof
seq-comp-presv-def-lift-inv less
by (*metis* (*no-types*, *lifting*))
ultimately show
loop-comp-helper acc m t A p = *loop-comp-helper acc m t A q*
proof (*safe*)
assume
dli-acc-seq-m: *defer-lift-invariance* (*acc* ▷ *m*) **and**
a-in-def-seq: *a* ∈ *defer* (*acc* ▷ *m*) *A p*
moreover from *this* **have** *electoral-module* (*acc* ▷ *m*)
unfolding *defer-lift-invariance-def*
by *blast*

```
        moreover have a ∈ defer (loop-comp-helper (acc ▷ m) m t) A p
          using loop-comp-equiv a-in-defer-lch
          by presburger
        ultimately have
          loop-comp-helper (acc ▷ m) m t A p
            = loop-comp-helper (acc ▷ m) m t A q
          using monotone-m mod-acc less a-lifted card-smaller-for-p
              defer-in-alts infinite-super less
          unfolding lifted-def
          by (metis (no-types))
        moreover have loop-comp-helper acc m t A q
                    = loop-comp-helper (acc ▷ m) m t A q
          using dli-acc-seq-m a-in-def-seq less a-lifted rec-step-q
          by blast
        ultimately show ?thesis
          using loop-comp-equiv
          by presburger
      qed
    qed
  next
    assume ¬ ¬t (acc A p)
    thus ?thesis
      using loop-comp-code-helper less
      unfolding defer-lift-invariance-def
      by metis
  qed
 qed
qed

lemma loop-comp-helper-def-lift-inv:
 fixes
   m :: 'a Electoral-Module and
   t :: 'a Termination-Condition and
   acc :: 'a Electoral-Module and
   A :: 'a set and
   p :: 'a Profile and
   q :: 'a Profile and
   a :: 'a
 assumes
   defer-lift-invariance m and
   defer-lift-invariance acc and
   profile A p and
   lifted A p q a and
   a ∈ defer (loop-comp-helper acc m t) A p
 shows (loop-comp-helper acc m t) A p = (loop-comp-helper acc m t) A q
 using assms loop-comp-helper-def-lift-inv-helper lifted-def
     defer-in-alts defer-lift-invariance-def finite-subset
 by (metis (no-types, lifting))
```

lemma *lifted-imp-fin-prof*:
fixes
A :: *'a set* **and**
p :: *'a Profile* **and**
q :: *'a Profile* **and**
a :: *'a*
assumes *lifted A p q a*
shows *finite-profile A p*
using *assms*
unfolding *lifted-def*
by *simp*

lemma *loop-comp-helper-presv-def-lift-inv*:
fixes
m :: *'a Electoral-Module* **and**
t :: *'a Termination-Condition* **and**
acc :: *'a Electoral-Module*
assumes
defer-lift-invariance m **and**
defer-lift-invariance acc
shows *defer-lift-invariance* (*loop-comp-helper acc m t*)
proof (*unfold defer-lift-invariance-def*, *safe*)
show *electoral-module* (*loop-comp-helper acc m t*)
using *loop-comp-helper-imp-partit assms*
unfolding *electoral-module-def defer-lift-invariance-def*
by (*metis* (*no-types*))
next
fix
A :: *'a set* **and**
p :: *'a Profile* **and**
q :: *'a Profile* **and**
a :: *'a*
assume
a ∈ *defer* (*loop-comp-helper acc m t*) *A p* **and**
lifted A p q a
thus *loop-comp-helper acc m t A p* = *loop-comp-helper acc m t A q*
using *lifted-imp-fin-prof loop-comp-helper-def-lift-inv assms*
by (*metis* (*full-types*))
qed

lemma *loop-comp-presv-non-electing-helper*:
fixes
m :: *'a Electoral-Module* **and**
t :: *'a Termination-Condition* **and**
acc :: *'a Electoral-Module* **and**
A :: *'a set* **and**
p :: *'a Profile* **and**
n :: *nat*
assumes

```
    non-electing-m: non-electing m and
    non-electing-acc: non-electing acc and
    prof: profile A p and
    acc-defer-card: n = card (defer acc A p)
  shows elect (loop-comp-helper acc m t) A p = {}
  using acc-defer-card non-electing-acc
proof (induct n arbitrary: acc rule: less-induct)
  case (less n)
  thus ?case
  proof (safe)
    fix x :: 'a
    assume
      acc-no-elect:
      (⋀ i acc'. i < card (defer acc A p) ⟹
        i = card (defer acc' A p) ⟹ non-electing acc' ⟹
          elect (loop-comp-helper acc' m t) A p = {}) and
      acc-non-elect: non-electing acc and
      x-in-acc-elect: x ∈ elect (loop-comp-helper acc m t) A p
    have ∀ m' n'. non-electing m' ∧ non-electing n' ⟶ non-electing (m' ▷ n')
      by simp
    hence seq-acc-m-non-elect: non-electing (acc ▷ m)
      using acc-non-elect non-electing-m
      by blast
    have ∀ i m'.
          i < card (defer acc A p) ∧ i = card (defer m' A p) ∧
            non-electing m' ⟶
            elect (loop-comp-helper m' m t) A p = {}
      using acc-no-elect
      by blast
    hence ∀ m'.
          finite (defer acc A p) ∧ defer m' A p ⊂ defer acc A p ∧
            non-electing m' ⟶
            elect (loop-comp-helper m' m t) A p = {}
      using psubset-card-mono
      by metis
    hence ¬ t (acc A p) ∧ defer (acc ▷ m) A p ⊂ defer acc A p ∧
              finite (defer acc A p) ⟶
              elect (loop-comp-helper acc m t) A p = {}
      using loop-comp-code-helper seq-acc-m-non-elect
      by (metis (no-types))
    moreover have elect acc A p = {}
      using acc-non-elect prof non-electing-def
      by auto
    ultimately show x ∈ {}
      using loop-comp-code-helper x-in-acc-elect
      by (metis (no-types))
  qed
qed
```

lemma *loop-comp-helper-iter-elim-def-n-helper*:
fixes
m :: *'a Electoral-Module* **and**
t :: *'a Termination-Condition* **and**
acc :: *'a Electoral-Module* **and**
A :: *'a set* **and**
p :: *'a Profile* **and**
n :: *nat* **and**
x :: *nat*
assumes
non-electing-m: *non-electing m* **and**
single-elimination: *eliminates 1 m* **and**
terminate-if-n-left: $\forall\ r.\ t\ r = (card\ (defer\text{-}r\ r) = x)$ **and**
x-greater-zero: $x > 0$ **and**
prof: *profile A p* **and**
n-acc-defer-card: $n = card\ (defer\ acc\ A\ p)$ **and**
n-ge-x: $n \geq x$ **and**
def-card-gt-one: $card\ (defer\ acc\ A\ p) > 1$ **and**
acc-nonelect: *non-electing acc*
shows $card\ (defer\ (loop\text{-}comp\text{-}helper\ acc\ m\ t)\ A\ p) = x$
using *n-ge-x def-card-gt-one acc-nonelect n-acc-defer-card*
proof (*induct n arbitrary*: *acc rule*: *less-induct*)
case (*less n*)
have *mod-acc*: *electoral-module acc*
using *less*
unfolding *non-electing-def*
by *metis*
hence *step-reduces-defer-set*: $defer\ (acc \triangleright m)\ A\ p \subset defer\ acc\ A\ p$
using *seq-comp-elim-one-red-def-set single-elimination prof less*
by *metis*
thus *?case*
proof (*cases t* (*acc A p*))
case *True*
assume *term-satisfied*: *t* (*acc A p*)
thus $card\ (defer\text{-}r\ (loop\text{-}comp\text{-}helper\ acc\ m\ t\ A\ p)) = x$
using *loop-comp-code-helper term-satisfied terminate-if-n-left*
by *metis*
next
case *False*
hence *card-not-eq-x*: $card\ (defer\ acc\ A\ p) \neq x$
using *terminate-if-n-left*
by *metis*
have *fin-def-acc*: *finite* (*defer acc A p*)
using *prof mod-acc less card.infinite not-one-less-zero*
by *metis*
hence *rec-step*:
$loop\text{-}comp\text{-}helper\ acc\ m\ t\ A\ p = loop\text{-}comp\text{-}helper\ (acc \triangleright m)\ m\ t\ A\ p$
using *False step-reduces-defer-set*

by *simp*
have *card-too-big*: *card* (*defer acc A p*) > *x*
using *card-not-eq-x dual-order.order-iff-strict less*
by *simp*
hence *enough-leftover*: *card* (*defer acc A p*) > *1*
using *x-greater-zero*
by *simp*
obtain *k* **where**
new-card-k: *k* = *card* (*defer* (*acc* ▷ *m*) *A p*)
by *metis*
have *defer acc A p* ⊆ *A*
using *defer-in-alts prof mod-acc*
by *metis*
hence *step-profile*: *profile* (*defer acc A p*) (*limit-profile* (*defer acc A p*) *p*)
using *prof limit-profile-sound*
by *metis*
hence
card (*defer m* (*defer acc A p*) (*limit-profile* (*defer acc A p*) *p*)) =
card (*defer acc A p*) − *1*
using *enough-leftover non-electing-m single-elim-decr-def-card*
single-elimination
by *metis*
hence *k-card*: *k* = *card* (*defer acc A p*) − *1*
using *mod-acc prof new-card-k non-electing-m seq-comp-defers-def-set*
by *metis*
hence *new-card-still-big-enough*: *x* ≤ *k*
using *card-too-big*
by *linarith*
show *?thesis*
proof (*cases x* < *k*)
case *True*
hence *1* < *card* (*defer* (*acc* ▷ *m*) *A p*)
using *new-card-k x-greater-zero*
by *linarith*
moreover have *k* < *n*
using *step-reduces-defer-set step-profile psubset-card-mono*
new-card-k less fin-def-acc
by *metis*
moreover have *electoral-module* (*acc* ▷ *m*)
using *mod-acc eliminates-def seq-comp-sound single-elimination*
by *metis*
moreover have *non-electing* (*acc* ▷ *m*)
using *less non-electing-m*
by *simp*
ultimately have *card* (*defer* (*loop-comp-helper* (*acc* ▷ *m*) *m t*) *A p*) = *x*
using *new-card-k new-card-still-big-enough less*
by *metis*
thus *?thesis*
using *rec-step*

```
      by presburger
    next
      case False
      thus ?thesis
        using dual-order.strict-iff-order new-card-k
              new-card-still-big-enough rec-step
              terminate-if-n-left
        by simp
    qed
  qed
qed

lemma loop-comp-helper-iter-elim-def-n:
  fixes
    m :: 'a Electoral-Module and
    t :: 'a Termination-Condition and
    acc :: 'a Electoral-Module and
    A :: 'a set and
    p :: 'a Profile and
    x :: nat
  assumes
    non-electing m and
    eliminates 1 m and
    ∀ r. (t r) = (card (defer-r r) = x) and
    x > 0 and
    profile A p and
    card (defer acc A p) ≥ x and
    non-electing acc
  shows card (defer (loop-comp-helper acc m t) A p) = x
  using assms gr-implies-not0 le-neq-implies-less less-one linorder-neqE-nat nat-neq-iff
        less-le loop-comp-helper-iter-elim-def-n-helper loop-comp-code-helper
  by (metis (no-types, lifting))

lemma iter-elim-def-n-helper:
  fixes
    m :: 'a Electoral-Module and
    t :: 'a Termination-Condition and
    A :: 'a set and
    p :: 'a Profile and
    x :: nat
  assumes
    non-electing-m: non-electing m and
    single-elimination: eliminates 1 m and
    terminate-if-n-left: ∀ r. (t r) = (card (defer-r r) = x) and
    x-greater-zero: x > 0 and
    prof: profile A p and
    enough-alternatives: card A ≥ x
  shows card (defer (m ⟳t) A p) = x
proof (cases)
```

```
  assume card A = x
  thus ?thesis
    using terminate-if-n-left
    by simp
next
  assume card-not-x: ¬ card A = x
  thus ?thesis
  proof (cases)
    assume card A < x
    thus ?thesis
      using enough-alternatives not-le
      by blast
  next
    assume ¬ card A < x
    hence card A > x
      using card-not-x
      by linarith
    moreover from this
    have card (defer m A p) = card A − 1
      using non-electing-m single-elimination single-elim-decr-def-card
            prof x-greater-zero
      by fastforce
    ultimately have card (defer m A p) ≥ x
      by linarith
    moreover have (m ↻t) A p = (loop-comp-helper m m t) A p
      using card-not-x terminate-if-n-left
      by simp
    ultimately show ?thesis
      using non-electing-m prof single-elimination terminate-if-n-left x-greater-zero
            loop-comp-helper-iter-elim-def-n
      by metis
  qed
qed
```

4.5.4 Composition Rules

The loop composition preserves defer-lift-invariance.

```
theorem loop-comp-presv-def-lift-inv[simp]:
  fixes
    m :: 'a Electoral-Module and
    t :: 'a Termination-Condition
  assumes defer-lift-invariance m
  shows defer-lift-invariance (m ↻t)
proof (unfold defer-lift-invariance-def, safe)
  have electoral-module m
    using assms
    unfolding defer-lift-invariance-def
    by simp
  thus electoral-module (m ↻t)
```

by (*simp add*: *loop-comp-sound*)
next
fix
A :: '*a set* **and**
p :: '*a Profile* **and**
q :: '*a Profile* **and**
a :: '*a*
assume
$a \in$ *defer* $(m\ \circlearrowleft_t)\ A\ p$ **and**
lifted $A\ p\ q\ a$
moreover have
$\forall\ p'\ q'\ a'.\ a' \in (\textit{defer}\ (m\ \circlearrowleft_t)\ A\ p') \wedge \textit{lifted}\ A\ p'\ q'\ a' \longrightarrow$
$(m\ \circlearrowleft_t)\ A\ p' = (m\ \circlearrowleft_t)\ A\ q'$
using *assms lifted-imp-fin-prof loop-comp-helper-def-lift-inv*
loop-composition.simps defer-module.simps
by (*metis* (*full-types*))
ultimately show $(m\ \circlearrowleft_t)\ A\ p = (m\ \circlearrowleft_t)\ A\ q$
by *metis*
qed

The loop composition preserves the property non-electing.

theorem *loop-comp-presv-non-electing*[*simp*]:
fixes
m :: '*a Electoral-Module* **and**
t :: '*a Termination-Condition*
assumes *non-electing* m
shows *non-electing* $(m\ \circlearrowleft_t)$
proof (*unfold non-electing-def*, *safe*)
show *electoral-module* $(m\ \circlearrowleft_t)$
using *loop-comp-sound assms*
unfolding *non-electing-def*
by *metis*
next
fix
A :: '*a set* **and**
p :: '*a Profile* **and**
a :: '*a*
assume
profile $A\ p$ **and**
$a \in$ *elect* $(m\ \circlearrowleft_t)\ A\ p$
thus $a \in \{\}$
using *def-mod-non-electing loop-comp-presv-non-electing-helper*
assms empty-iff loop-comp-code
unfolding *non-electing-def*
by (*metis* (*no-types*))
qed

theorem *iter-elim-def-n*[*simp*]:
fixes

```
    m :: 'a Electoral-Module and
    t :: 'a Termination-Condition and
    n :: nat
  assumes
    non-electing-m: non-electing m and
    single-elimination: eliminates 1 m and
    terminate-if-n-left: ∀ r. t r = (card (defer-r r) = n) and
    x-greater-zero: n > 0
  shows defers n (m ↻t)
proof (unfold defers-def, safe)
  show electoral-module (m ↻t)
    using loop-comp-sound non-electing-m
    unfolding non-electing-def
    by metis
next
  fix
    A :: 'a set and
    p :: 'a Profile
  assume
    n ≤ card A and
    finite A and
    profile A p
  thus card (defer (m ↻t) A p) = n
    using iter-elim-def-n-helper assms
    by metis
qed

end
```

4.6 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.

4.6.1 Definition

fun *maximum-parallel-composition* :: *'a Electoral-Module* ⇒
'a Electoral-Module ⇒ *'a Electoral-Module* **where**
maximum-parallel-composition m n =
(*let a* = *max-aggregator in* (*m* ‖$_a$ *n*))

abbreviation *max-parallel* :: *'a Electoral-Module* ⇒ *'a Electoral-Module* ⇒
'a Electoral-Module (**infix** ‖$_\uparrow$ *50*) **where**
m ‖$_\uparrow$ *n* == *maximum-parallel-composition m n*

4.6.2 Soundness

theorem *max-par-comp-sound*:
fixes
m :: *'a Electoral-Module* **and**
n :: *'a Electoral-Module*
assumes
electoral-module m **and**
electoral-module n
shows *electoral-module* (*m* ‖$_\uparrow$ *n*)
using *assms*
by *simp*

4.6.3 Lemmas

lemma *max-agg-eq-result*:
fixes
m :: *'a Electoral-Module* **and**
n :: *'a Electoral-Module* **and**
A :: *'a set* **and**
p :: *'a Profile* **and**
a :: *'a*
assumes
module-m: *electoral-module m* **and**
module-n: *electoral-module n* **and**
prof-p: *profile A p* **and**
a-in-A: *a* ∈ *A*
shows *mod-contains-result* (*m* ‖$_\uparrow$ *n*) *m A p a* ∨
mod-contains-result (*m* ‖$_\uparrow$ *n*) *n A p a*
proof (*cases*)
assume *a-elect*: *a* ∈ *elect* (*m* ‖$_\uparrow$ *n*) *A p*
hence *let* (*e*, *r*, *d*) = *m A p*;
(*e'*, *r'*, *d'*) = *n A p in*
a ∈ *e* ∪ *e'*
by *auto*
hence *a* ∈ (*elect m A p*) ∪ (*elect n A p*)
by *auto*
moreover have
∀ *m' n' A' p' a'*.

```
      mod-contains-result m' n' A' p' (a'::'a) =
        (electoral-module m' ∧ electoral-module n'
          ∧ profile A' p' ∧ a' ∈ A'
          ∧ (a' ∉ elect m' A' p' ∨ a' ∈ elect n' A' p')
          ∧ (a' ∉ reject m' A' p' ∨ a' ∈ reject n' A' p')
          ∧ (a' ∉ defer m' A' p' ∨ a' ∈ defer n' A' p'))
    unfolding mod-contains-result-def
    by simp
  moreover have module-mn: electoral-module (m ‖↑ n)
    using module-m module-n
    by simp
  moreover have a ∉ defer (m ‖↑ n) A p
    using module-mn IntI a-elect empty-iff prof-p result-disj
    by (metis (no-types))
  moreover have a ∉ reject (m ‖↑ n) A p
    using module-mn IntI a-elect empty-iff prof-p result-disj
    by (metis (no-types))
  ultimately show ?thesis
    using assms
    by blast
next
  assume not-a-elect: a ∉ elect (m ‖↑ n) A p
  thus ?thesis
  proof (cases)
    assume a-in-def: a ∈ defer (m ‖↑ n) A p
    thus ?thesis
    proof (safe)
      assume not-mod-cont-mn: ¬ mod-contains-result (m ‖↑ n) n A p a
      have par-emod: ∀ m' n'.
        electoral-module m' ∧ electoral-module n' ⟶ electoral-module (m' ‖↑ n')
        using max-par-comp-sound
        by blast
      have set-intersect: ∀ a' A' A''. (a' ∈ A' ∩ A'') = (a' ∈ A' ∧ a' ∈ A'')
        by blast
      have wf-n: well-formed A (n A p)
        using prof-p module-n
        unfolding electoral-module-def
        by blast
      have wf-m: well-formed A (m A p)
        using prof-p module-m
        unfolding electoral-module-def
        by blast
      have e-mod-par: electoral-module (m ‖↑ n)
        using par-emod module-m module-n
        by blast
      hence electoral-module (m ‖_m ax-aggregator n)
        by simp
      hence result-disj-max:
        elect (m ‖_m ax-aggregator n) A p ∩
```

```
    reject (m ‖_m ax-aggregator n) A p = {} ∧
   elect (m ‖_m ax-aggregator n) A p ∩
    defer (m ‖_m ax-aggregator n) A p = {} ∧
   reject (m ‖_m ax-aggregator n) A p ∩
    defer (m ‖_m ax-aggregator n) A p = {}
 using prof-p result-disj
 by metis
have a-not-elect: a ∉ elect (m ‖_m ax-aggregator n) A p
 using result-disj-max a-in-def
 by force
have result-m: (elect m A p, reject m A p, defer m A p) = m A p
 by auto
have result-n: (elect n A p, reject n A p, defer n A p) = n A p
 by auto
have max-pq:
 ∀ (A'::'a set) m' n'.
  elect-r (max-aggregator A' m' n') = elect-r m' ∪ elect-r n'
 by force
have a ∉ elect (m ‖_m ax-aggregator n) A p
 using a-not-elect
 by blast
hence a ∉ elect m A p ∪ elect n A p
 using max-pq
 by simp
hence b-not-elect-mn: a ∉ elect m A p ∧ a ∉ elect n A p
 by blast
have b-not-mpar-rej: a ∉ reject (m ‖_m ax-aggregator n) A p
 using result-disj-max a-in-def
 by fastforce
have mod-cont-res-fg:
 ∀ m' n' A' p' (a'::'a).
  mod-contains-result m' n' A' p' a' =
   (electoral-module m' ∧ electoral-module n'
    ∧ profile A' p' ∧ a' ∈ A'
    ∧ (a' ∈ elect m' A' p' ⟶ a' ∈ elect n' A' p')
    ∧ (a' ∈ reject m' A' p' ⟶ a' ∈ reject n' A' p')
    ∧ (a' ∈ defer m' A' p' ⟶ a' ∈ defer n' A' p'))
 by (simp add: mod-contains-result-def)
have max-agg-res:
 max-aggregator A (elect m A p, reject m A p, defer m A p)
  (elect n A p, reject n A p, defer n A p) = (m ‖_m ax-aggregator n) A p
 by simp
have well-f-max:
 ∀ r' r'' e' e'' d' d'' A'.
  well-formed A' (e', r', d') ∧ well-formed A' (e'', r'', d'') ⟶
   reject-r (max-aggregator A' (e', r', d') (e'', r'', d'')) = r' ∩ r''
 using max-agg-rej-set
 by metis
have e-mod-disj:
```

```
    ∀ m' (A'::'a set) p'. electoral-module m' ∧ profile A' p'
      ⟶ elect m' A' p' ∪ reject m' A' p' ∪ defer m' A' p' = A'
    using result-presv-alts
    by blast
  hence e-mod-disj-n: elect n A p ∪ reject n A p ∪ defer n A p = A
    using prof-p module-n
    by metis
  have ∀ m' n' A' p' (b::'a).
        mod-contains-result m' n' A' p' b =
          (electoral-module m' ∧ electoral-module n'
            ∧ profile A' p' ∧ b ∈ A'
            ∧ (b ∈ elect m' A' p' ⟶ b ∈ elect n' A' p')
            ∧ (b ∈ reject m' A' p' ⟶ b ∈ reject n' A' p')
            ∧ (b ∈ defer m' A' p' ⟶ b ∈ defer n' A' p'))
    unfolding mod-contains-result-def
    by simp
  hence a ∈ reject n A p
    using e-mod-disj-n e-mod-par prof-p a-in-A module-n not-mod-cont-mn
        a-not-elect b-not-elect-mn b-not-mpar-rej
    by auto
  hence a ∉ reject m A p
    using well-f-max max-agg-res result-m result-n set-intersect
        wf-m wf-n b-not-mpar-rej
    by (metis (no-types))
  hence a ∉ defer (m ‖↑ n) A p ∨ a ∈ defer m A p
    using e-mod-disj prof-p a-in-A module-m b-not-elect-mn
    by blast
  thus mod-contains-result (m ‖↑ n) m A p a
    using b-not-mpar-rej mod-cont-res-fg e-mod-par prof-p a-in-A
        module-m a-not-elect
    by auto
qed
next
  assume not-a-defer: a ∉ defer (m ‖↑ n) A p
  have el-rej-defer: (elect m A p, reject m A p, defer m A p) = m A p
    by auto
  from not-a-elect not-a-defer
  have a-reject: a ∈ reject (m ‖↑ n) A p
  using electoral-mod-defer-elem a-in-A module-m module-n prof-p max-par-comp-sound
    by metis
  hence case snd (m A p) of (r, d) ⇒
        case n A p of (e', r', d') ⇒
          a ∈ reject-r (max-aggregator A (elect m A p, r, d) (e', r', d'))
    using el-rej-defer
    by force
  hence let (e, r, d) = m A p;
        (e', r', d') = n A p in
          a ∈ reject-r (max-aggregator A (e, r, d) (e', r', d'))
    by (simp add: case-prod-unfold)
```

```
    hence let (e, r, d) = m A p;
            (e', r', d') = n A p in
              a ∈ A − (e ∪ e' ∪ d ∪ d')
      by simp
    hence a ∉ elect m A p ∪ (defer n A p ∪ defer m A p)
      by force
    thus ?thesis
      using mod-contains-result-comm mod-contains-result-def Un-iff
            a-reject prof-p a-in-A module-m module-n max-par-comp-sound
      by (metis (no-types))
  qed
qed

lemma max-agg-rej-iff-both-reject:
  fixes
    m :: 'a Electoral-Module and
    n :: 'a Electoral-Module and
    A :: 'a set and
    p :: 'a Profile and
    a :: 'a
  assumes
    finite-profile A p and
    electoral-module m and
    electoral-module n
  shows (a ∈ reject (m ‖↑ n) A p) = (a ∈ reject m A p ∧ a ∈ reject n A p)
proof
  assume rej-a: a ∈ reject (m ‖↑ n) A p
  hence case n A p of (e, r, d) ⇒
          a ∈ reject-r (max-aggregator A
                (elect m A p, reject m A p, defer m A p) (e, r, d))
    by auto
  hence case snd (m A p) of (r, d) ⇒
          case n A p of (e', r', d') ⇒
            a ∈ reject-r (max-aggregator A (elect m A p, r, d) (e', r', d'))
    by force
  with rej-a
  have let (e, r, d) = m A p;
          (e', r', d') = n A p in
            a ∈ reject-r (max-aggregator A (e, r, d) (e', r', d'))
    by (simp add: prod.case-eq-if)
  hence let (e, r, d) = m A p;
            (e', r', d') = n A p in
              a ∈ A − (e ∪ e' ∪ d ∪ d')
    by simp
  hence a ∈ A − (elect m A p ∪ elect n A p ∪ defer m A p ∪ defer n A p)
    by auto
  thus a ∈ reject m A p ∧ a ∈ reject n A p
    using Diff-iff Un-iff electoral-mod-defer-elem assms
    by metis
```

next
assume $a \in$ *reject* $m\ A\ p \wedge a \in$ *reject* $n\ A\ p$
moreover from *this*
have $a \notin$ *elect* $m\ A\ p \wedge a \notin$ *defer* $m\ A\ p \wedge a \notin$ *elect* $n\ A\ p \wedge a \notin$ *defer* $n\ A\ p$
using *IntI empty-iff assms result-disj*
by *metis*
ultimately show $a \in$ *reject* $(m \parallel_{\uparrow} n)\ A\ p$
using *DiffD1 max-agg-eq-result mod-contains-result-comm mod-contains-result-def reject-not-elec-or-def assms*
by (*metis* (*no-types*))
qed

lemma *max-agg-rej-fst-imp-seq-contained*:
fixes
$m :: 'a$ *Electoral-Module* **and**
$n :: 'a$ *Electoral-Module* **and**
$A :: 'a$ *set* **and**
$p :: 'a$ *Profile* **and**
$a :: 'a$
assumes
f-prof: *finite-profile* $A\ p$ **and**
module-m: *electoral-module* m **and**
module-n: *electoral-module* n **and**
rejected: $a \in$ *reject* $n\ A\ p$
shows *mod-contains-result* $m\ (m \parallel_{\uparrow} n)\ A\ p\ a$
using *assms*
proof (*unfold mod-contains-result-def*, *safe*)
show *electoral-module* $(m \parallel_{\uparrow} n)$
using *module-m module-n*
by *simp*
next
show $a \in A$
using *f-prof module-n rejected reject-in-alts*
by *auto*
next
assume *a-in-elect*: $a \in$ *elect* $m\ A\ p$
hence *a-not-reject*: $a \notin$ *reject* $m\ A\ p$
using *disjoint-iff-not-equal f-prof module-m result-disj*
by *metis*
have *reject* $n\ A\ p \subseteq A$
using *f-prof module-n*
by (*simp add*: *reject-in-alts*)
hence $a \in A$
using *in-mono rejected*
by *metis*
with *a-in-elect a-not-reject*
show $a \in$ *elect* $(m \parallel_{\uparrow} n)\ A\ p$
using *f-prof max-agg-eq-result module-m module-n rejected max-agg-rej-iff-both-reject mod-contains-result-comm*

mod-contains-result-def
by *metis*
next
assume $a \in$ *reject* $m\ A\ p$
hence $a \in$ *reject* $m\ A\ p \wedge a \in$ *reject* $n\ A\ p$
using *rejected*
by *simp*
thus $a \in$ *reject* $(m \parallel_{\uparrow} n)\ A\ p$
using *f-prof max-agg-rej-iff-both-reject module-m module-n*
by (*metis* (*no-types*))
next
assume *a-in-defer*: $a \in$ *defer* $m\ A\ p$
then obtain $d :: 'a$ **where**
defer-a: $a = d \wedge d \in$ *defer* $m\ A\ p$
by *metis*
have *a-not-rej*: $a \notin$ *reject* $m\ A\ p$
using *disjoint-iff-not-equal f-prof defer-a module-m result-disj*
by (*metis* (*no-types*))
have
$\forall\ m'\ A'\ p'.$
electoral-module $m' \wedge$ *finite* $A' \wedge$ *profile* $A'\ p' \longrightarrow$
elect $m'\ A'\ p' \cup$ *reject* $m'\ A'\ p' \cup$ *defer* $m'\ A'\ p' = A'$
using *result-presv-alts*
by *metis*
hence $a \in A$
using *a-in-defer f-prof module-m*
by *blast*
with *defer-a a-not-rej*
show $a \in$ *defer* $(m \parallel_{\uparrow} n)\ A\ p$
using *f-prof max-agg-eq-result max-agg-rej-iff-both-reject*
mod-contains-result-comm mod-contains-result-def
module-m module-n rejected
by *metis*
qed

lemma *max-agg-rej-fst-equiv-seq-contained*:
fixes
$m :: 'a$ *Electoral-Module* **and**
$n :: 'a$ *Electoral-Module* **and**
$A :: 'a$ *set* **and**
$p :: 'a$ *Profile* **and**
$a :: 'a$
assumes
finite-profile $A\ p$ **and**
electoral-module m **and**
electoral-module n **and**
$a \in$ *reject* $n\ A\ p$
shows *mod-contains-result-sym* $(m \parallel_{\uparrow} n)\ m\ A\ p\ a$
using *assms*

proof (*unfold mod-contains-result-sym-def*, *safe*)
assume $a \in$ *reject* $(m \parallel_{\uparrow} n)$ A p
thus $a \in$ *reject* m A p
using *assms max-agg-rej-iff-both-reject*
by (*metis* (*no-types*))
next
have *mod-contains-result* m $(m \parallel_{\uparrow} n)$ A p a
using *assms max-agg-rej-fst-imp-seq-contained*
by (*metis* (*full-types*))
thus
$a \in$ *elect* $(m \parallel_{\uparrow} n)$ A $p \Longrightarrow a \in$ *elect* m A p **and**
$a \in$ *defer* $(m \parallel_{\uparrow} n)$ A $p \Longrightarrow a \in$ *defer* m A p
using *mod-contains-result-comm*
unfolding *mod-contains-result-def*
by (*metis* (*full-types*), *metis* (*full-types*))
next
show
electoral-module $(m \parallel_{\uparrow} n)$ **and**
$a \in A$
using *assms max-agg-rej-fst-imp-seq-contained*
unfolding *mod-contains-result-def*
by (*metis* (*full-types*), *metis* (*full-types*))
next
show
$a \in$ *elect* m A $p \Longrightarrow a \in$ *elect* $(m \parallel_{\uparrow} n)$ A p **and**
$a \in$ *reject* m A $p \Longrightarrow a \in$ *reject* $(m \parallel_{\uparrow} n)$ A p **and**
$a \in$ *defer* m A $p \Longrightarrow a \in$ *defer* $(m \parallel_{\uparrow} n)$ A p
using *assms max-agg-rej-fst-imp-seq-contained*
unfolding *mod-contains-result-def*
by (*metis* (*no-types*), *metis* (*no-types*), *metis* (*no-types*))
qed

lemma *max-agg-rej-snd-imp-seq-contained*:
fixes
m :: *'a Electoral-Module* **and**
n :: *'a Electoral-Module* **and**
A :: *'a set* **and**
p :: *'a Profile* **and**
a :: *'a*
assumes
f-prof: *finite-profile* A p **and**
module-m: *electoral-module* m **and**
module-n: *electoral-module* n **and**
rejected: $a \in$ *reject* m A p
shows *mod-contains-result* n $(m \parallel_{\uparrow} n)$ A p a
using *assms*
proof (*unfold mod-contains-result-def*, *safe*)
show *electoral-module* $(m \parallel_{\uparrow} n)$
using *module-m module-n*

by *simp*
next
show $a \in A$
using *f-prof in-mono module-m reject-in-alts rejected*
by (*metis* (*no-types*))
next
assume $a \in$ *elect* $n\ A\ p$
thus $a \in$ *elect* $(m \parallel_{\uparrow} n)\ A\ p$
using *Un-iff elect-rej-def-combination fst-conv maximum-parallel-composition.simps*
max-aggregator.simps
unfolding *parallel-composition.simps*
by (*metis* (*mono-tags*, *lifting*))
next
assume $a \in$ *reject* $n\ A\ p$
thus $a \in$ *reject* $(m \parallel_{\uparrow} n)\ A\ p$
using *f-prof max-agg-rej-iff-both-reject module-m module-n rejected*
by *metis*
next
assume $a \in$ *defer* $n\ A\ p$
moreover have $a \in A$
using *f-prof max-agg-rej-fst-imp-seq-contained module-m rejected*
unfolding *mod-contains-result-def*
by *metis*
ultimately show $a \in$ *defer* $(m \parallel_{\uparrow} n)\ A\ p$
using *disjoint-iff-not-equal max-agg-eq-result max-agg-rej-iff-both-reject*
f-prof mod-contains-result-comm mod-contains-result-def
module-m module-n rejected result-disj
by *metis*
qed

lemma *max-agg-rej-snd-equiv-seq-contained*:
fixes
$m :: 'a$ *Electoral-Module* **and**
$n :: 'a$ *Electoral-Module* **and**
$A :: 'a$ *set* **and**
$p :: 'a$ *Profile* **and**
$a :: 'a$
assumes
finite-profile $A\ p$ **and**
electoral-module m **and**
electoral-module n **and**
$a \in$ *reject* $m\ A\ p$
shows *mod-contains-result-sym* $(m \parallel_{\uparrow} n)\ n\ A\ p\ a$
using *assms*
proof (*unfold mod-contains-result-sym-def*, *safe*)
assume $a \in$ *reject* $(m \parallel_{\uparrow} n)\ A\ p$
thus $a \in$ *reject* $n\ A\ p$
using *assms max-agg-rej-iff-both-reject*
by (*metis* (*no-types*))

next
have *mod-contains-result n* (*m* $\|_{\uparrow}$ *n*) *A p a*
using *assms max-agg-rej-snd-imp-seq-contained*
by (*metis* (*full-types*))
thus
a ∈ *elect* (*m* $\|_{\uparrow}$ *n*) *A p* ⟹ *a* ∈ *elect n A p* **and**
a ∈ *defer* (*m* $\|_{\uparrow}$ *n*) *A p* ⟹ *a* ∈ *defer n A p*
using *mod-contains-result-comm*
unfolding *mod-contains-result-def*
by (*metis* (*full-types*), *metis* (*full-types*))
next
show
electoral-module (*m* $\|_{\uparrow}$ *n*) **and**
a ∈ *A*
using *assms max-agg-rej-snd-imp-seq-contained*
unfolding *mod-contains-result-def*
by (*metis* (*full-types*), *metis* (*full-types*))
next
show
a ∈ *elect n A p* ⟹ *a* ∈ *elect* (*m* $\|_{\uparrow}$ *n*) *A p* **and**
a ∈ *reject n A p* ⟹ *a* ∈ *reject* (*m* $\|_{\uparrow}$ *n*) *A p* **and**
a ∈ *defer n A p* ⟹ *a* ∈ *defer* (*m* $\|_{\uparrow}$ *n*) *A p*
using *assms max-agg-rej-snd-imp-seq-contained*
unfolding *mod-contains-result-def*
by (*metis* (*no-types*), *metis* (*no-types*), *metis* (*no-types*))
qed

lemma *max-agg-rej-intersect*:
fixes
m :: *'a Electoral-Module* **and**
n :: *'a Electoral-Module* **and**
A :: *'a set* **and**
p :: *'a Profile*
assumes
electoral-module m **and**
electoral-module n **and**
finite-profile A p
shows *reject* (*m* $\|_{\uparrow}$ *n*) *A p* = (*reject m A p*) ∩ (*reject n A p*)
proof −
have *A* = (*elect m A p*) ∪ (*reject m A p*) ∪ (*defer m A p*) ∧
A = (*elect n A p*) ∪ (*reject n A p*) ∪ (*defer n A p*)
using *assms result-presv-alts*
by *metis*
hence *A* − ((*elect m A p*) ∪ (*defer m A p*)) = (*reject m A p*) ∧
A − ((*elect n A p*) ∪ (*defer n A p*)) = (*reject n A p*)
using *assms reject-not-elec-or-def*
by *auto*
hence *A* − ((*elect m A p*) ∪ (*elect n A p*) ∪ (*defer m A p*) ∪ (*defer n A p*)) =
(*reject m A p*) ∩ (*reject n A p*)

by *blast*
hence *let* $(e, r, d) = m\ A\ p$;
$(e', r', d') = n\ A\ p$ *in*
$A - (e \cup e' \cup d \cup d') = r \cap r'$
by *fastforce*
thus *?thesis*
by *auto*
qed

lemma *dcompat-dec-by-one-mod*:
fixes
$m :: 'a$ *Electoral-Module* **and**
$n :: 'a$ *Electoral-Module* **and**
$A :: 'a$ *set* **and**
$a :: 'a$
assumes
disjoint-compatibility $m\ n$ **and**
$a \in A$
shows
$(\forall\ p.$ *finite-profile* $A\ p \longrightarrow$ *mod-contains-result* $m\ (m \parallel_{\uparrow} n)\ A\ p\ a) \vee$
$(\forall\ p.$ *finite-profile* $A\ p \longrightarrow$ *mod-contains-result* $n\ (m \parallel_{\uparrow} n)\ A\ p\ a)$
using *DiffI assms max-agg-rej-fst-imp-seq-contained max-agg-rej-snd-imp-seq-contained*
unfolding *disjoint-compatibility-def*
by *metis*

4.6.4 Composition Rules

Using a conservative aggregator, the parallel composition preserves the property non-electing.

theorem *conserv-max-agg-presv-non-electing*[*simp*]:
fixes
$m :: 'a$ *Electoral-Module* **and**
$n :: 'a$ *Electoral-Module*
assumes
non-electing m **and**
non-electing n
shows *non-electing* $(m \parallel_{\uparrow} n)$
using *assms*
by *simp*

Using the max aggregator, composing two compatible electoral modules in parallel preserves defer-lift-invariance.

theorem *par-comp-def-lift-inv*[*simp*]:
fixes
$m :: 'a$ *Electoral-Module* **and**
$n :: 'a$ *Electoral-Module*
assumes
compatible: *disjoint-compatibility* $m\ n$ **and**

```
    monotone-m: defer-lift-invariance m and
    monotone-n: defer-lift-invariance n
  shows defer-lift-invariance (m ∥↑ n)
proof (unfold defer-lift-invariance-def, safe)
  have electoral-module m
    using monotone-m
    unfolding defer-lift-invariance-def
    by simp
  moreover have electoral-module n
    using monotone-n
    unfolding defer-lift-invariance-def
    by simp
  ultimately show electoral-module (m ∥↑ n)
    by simp
next
  fix
    A :: 'a set and
    p :: 'a Profile and
    q :: 'a Profile and
    a :: 'a
  assume
    defer-a: a ∈ defer (m ∥↑ n) A p and
    lifted-a: Profile.lifted A p q a
  hence f-profs: finite-profile A p ∧ finite-profile A q
    unfolding lifted-def
    by simp
  from compatible
  obtain B :: 'a set where
    alts: B ⊆ A ∧
          (∀ b ∈ B. indep-of-alt m A b ∧
              (∀ p'. finite-profile A p' ⟶ b ∈ reject m A p')) ∧
          (∀ b ∈ A − B. indep-of-alt n A b ∧
              (∀ p'. finite-profile A p' ⟶ b ∈ reject n A p'))
    using f-profs
    unfolding disjoint-compatibility-def
    by (metis (no-types, lifting))
  have ∀ b ∈ A. prof-contains-result (m ∥↑ n) A p q b
  proof (cases)
    assume a-in-B: a ∈ B
    hence a ∈ reject m A p
      using alts f-profs
      by blast
    with defer-a
    have defer-n: a ∈ defer n A p
      using compatible f-profs max-agg-rej-snd-equiv-seq-contained
      unfolding disjoint-compatibility-def mod-contains-result-sym-def
      by metis
    have ∀ b ∈ B. mod-contains-result-sym (m ∥↑ n) n A p b
      using alts compatible max-agg-rej-snd-equiv-seq-contained f-profs
```

unfolding *disjoint-compatibility-def*
by *metis*
moreover have $\forall\ b \in A.$ *prof-contains-result n A p q b*
proof (*unfold prof-contains-result-def*, *clarify*)
fix $b :: 'a$
assume *b-in-A*: $b \in A$
show *electoral-module n* $\wedge$ *profile A p* $\wedge$ *profile A q* $\wedge$ $b \in A$
$\wedge$ ($b \in$ *elect n A p* $\longrightarrow$ $b \in$ *elect n A q*)
$\wedge$ ($b \in$ *reject n A p* $\longrightarrow$ $b \in$ *reject n A q*)
$\wedge$ ($b \in$ *defer n A p* $\longrightarrow$ $b \in$ *defer n A q*)
proof (*safe*)
show *electoral-module n*
using *monotone-n*
unfolding *defer-lift-invariance-def*
by *metis*
next
show *profile A p*
using *f-profs*
by *simp*
next
show *profile A q*
using *f-profs*
by *simp*
next
show $b \in A$
using *b-in-A*
by *simp*
next
assume $b \in$ *elect n A p*
thus $b \in$ *elect n A q*
using *defer-n lifted-a monotone-n f-profs*
unfolding *defer-lift-invariance-def*
by *metis*
next
assume $b \in$ *reject n A p*
thus $b \in$ *reject n A q*
using *defer-n lifted-a monotone-n f-profs*
unfolding *defer-lift-invariance-def*
by *metis*
next
assume $b \in$ *defer n A p*
thus $b \in$ *defer n A q*
using *defer-n lifted-a monotone-n f-profs*
unfolding *defer-lift-invariance-def*
by *metis*
qed
qed
moreover have $\forall\ b \in B.$ *mod-contains-result n* ($m \parallel_{\uparrow} n$) *A q b*
using *alts compatible max-agg-rej-snd-imp-seq-contained f-profs*

unfolding *disjoint-compatibility-def*
by *metis*
ultimately have *prof-contains-result-of-comps-for-elems-in-B*:
$\forall\ b \in B.$ *prof-contains-result* $(m \parallel_{\uparrow} n)\ A\ p\ q\ b$
unfolding *mod-contains-result-def mod-contains-result-sym-def*
prof-contains-result-def
by *simp*
have $\forall\ b \in A - B.$ *mod-contains-result-sym* $(m \parallel_{\uparrow} n)\ m\ A\ p\ b$
using *alts max-agg-rej-fst-equiv-seq-contained monotone-m monotone-n f-profs*
unfolding *defer-lift-invariance-def*
by *metis*
moreover have $\forall\ b \in A.$ *prof-contains-result* $m\ A\ p\ q\ b$
proof (*unfold prof-contains-result-def*, *clarify*)
fix $b :: 'a$
assume *b-in-A*: $b \in A$
show *electoral-module* $m \wedge$ *profile* $A\ p \wedge$ *profile* $A\ q \wedge b \in A \wedge$
$(b \in$ *elect* $m\ A\ p \longrightarrow b \in$ *elect* $m\ A\ q) \wedge$
$(b \in$ *reject* $m\ A\ p \longrightarrow b \in$ *reject* $m\ A\ q) \wedge$
$(b \in$ *defer* $m\ A\ p \longrightarrow b \in$ *defer* $m\ A\ q)$
proof (*safe*)
show *electoral-module* m
using *monotone-m*
unfolding *defer-lift-invariance-def*
by *metis*
next
show *profile* $A\ p$
using *f-profs*
by *simp*
next
show *profile* $A\ q$
using *f-profs*
by *simp*
next
show $b \in A$
using *b-in-A*
by *simp*
next
assume $b \in$ *elect* $m\ A\ p$
thus $b \in$ *elect* $m\ A\ q$
using *alts a-in-B lifted-a lifted-imp-equiv-prof-except-a*
unfolding *indep-of-alt-def*
by *metis*
next
assume $b \in$ *reject* $m\ A\ p$
thus $b \in$ *reject* $m\ A\ q$
using *alts a-in-B lifted-a lifted-imp-equiv-prof-except-a*
unfolding *indep-of-alt-def*
by *metis*
next

assume $b \in$ *defer* m A p
thus $b \in$ *defer* m A q
using *alts a-in-B lifted-a lifted-imp-equiv-prof-except-a*
unfolding *indep-of-alt-def*
by *metis*
qed
qed
moreover have $\forall\ b \in A - B$. *mod-contains-result* m $(m \parallel_{\uparrow} n)$ A q b
using *alts max-agg-rej-fst-imp-seq-contained monotone-m monotone-n f-profs*
unfolding *defer-lift-invariance-def*
by *metis*
ultimately have $\forall\ b \in A - B$. *prof-contains-result* $(m \parallel_{\uparrow} n)$ A p q b
unfolding *mod-contains-result-def mod-contains-result-sym-def*
prof-contains-result-def
by *simp*
thus *?thesis*
using *prof-contains-result-of-comps-for-elems-in-B*
by *blast*
next
assume $a \notin B$
hence *a-in-set-diff*: $a \in A - B$
using *DiffI lifted-a compatible f-profs*
unfolding *Profile.lifted-def*
by (*metis* (*no-types*, *lifting*))
hence $a \in$ *reject* n A p
using *alts f-profs*
by *blast*
hence *defer-m*: $a \in$ *defer* m A p
using *DiffD1 DiffD2 compatible dcompat-dec-by-one-mod f-profs defer-not-elec-or-rej*
max-agg-sound par-comp-sound disjoint-compatibility-def not-rej-imp-elec-or-def
mod-contains-result-def defer-a
unfolding *maximum-parallel-composition.simps*
by *metis*
have $\forall\ b \in B$. *mod-contains-result-sym* $(m \parallel_{\uparrow} n)$ n A p b
using *alts max-agg-rej-snd-equiv-seq-contained monotone-m monotone-n f-profs*
unfolding *defer-lift-invariance-def*
by *metis*
moreover have $\forall\ b \in A$. *prof-contains-result* n A p q b
proof (*unfold prof-contains-result-def*, *clarify*)
fix $b :: 'a$
assume *b-in-A*: $b \in A$
show *electoral-module* $n \wedge$ *profile* A $p \wedge$ *profile* A $q \wedge b \in A$
$\wedge\ (b \in$ *elect* n A $p \longrightarrow b \in$ *elect* n A $q)$
$\wedge\ (b \in$ *reject* n A $p \longrightarrow b \in$ *reject* n A $q)$
$\wedge\ (b \in$ *defer* n A $p \longrightarrow b \in$ *defer* n A $q)$
proof (*safe*)
show *electoral-module* n
using *monotone-n*
unfolding *defer-lift-invariance-def*

```
      by metis
    next
      show profile A p
        using f-profs
        by simp
    next
      show profile A q
        using f-profs
        by simp
    next
      show b ∈ A
        using b-in-A
        by simp
    next
      assume b ∈ elect n A p
      thus b ∈ elect n A q
        using alts a-in-set-diff lifted-a lifted-imp-equiv-prof-except-a
        unfolding indep-of-alt-def
        by metis
    next
      assume b ∈ reject n A p
      thus b ∈ reject n A q
        using alts a-in-set-diff lifted-a lifted-imp-equiv-prof-except-a
        unfolding indep-of-alt-def
        by metis
    next
      assume b ∈ defer n A p
      thus b ∈ defer n A q
        using alts a-in-set-diff lifted-a lifted-imp-equiv-prof-except-a
        unfolding indep-of-alt-def
        by metis
    qed
qed
moreover have ∀ b ∈ B. mod-contains-result n (m ‖↑ n) A q b
  using alts compatible max-agg-rej-snd-imp-seq-contained f-profs
  unfolding disjoint-compatibility-def
  by metis
ultimately have prof-contains-result-of-comps-for-elems-in-B:
  ∀ b ∈ B. prof-contains-result (m ‖↑ n) A p q b
    unfolding mod-contains-result-def mod-contains-result-sym-def
              prof-contains-result-def
  by simp
have ∀ b ∈ A − B. mod-contains-result-sym (m ‖↑ n) m A p b
  using alts max-agg-rej-fst-equiv-seq-contained monotone-m monotone-n f-profs
  unfolding defer-lift-invariance-def
  by metis
moreover have ∀ b ∈ A. prof-contains-result m A p q b
proof (unfold prof-contains-result-def, clarify)
  fix b :: 'a
```

```
  assume b-in-A: b ∈ A
  show electoral-module m ∧ profile A p ∧ profile A q ∧ b ∈ A
        ∧ (b ∈ elect m A p ⟶ b ∈ elect m A q)
        ∧ (b ∈ reject m A p ⟶ b ∈ reject m A q)
        ∧ (b ∈ defer m A p ⟶ b ∈ defer m A q)
  proof (safe)
    show electoral-module m
      using monotone-m
      unfolding defer-lift-invariance-def
      by simp
  next
    show profile A p
      using f-profs
      by simp
  next
    show profile A q
      using f-profs
      by simp
  next
    show b ∈ A
      using b-in-A
      by simp
  next
    assume b ∈ elect m A p
    thus b ∈ elect m A q
      using defer-m lifted-a monotone-m
      unfolding defer-lift-invariance-def
      by metis
  next
    assume b ∈ reject m A p
    thus b ∈ reject m A q
      using defer-m lifted-a monotone-m
      unfolding defer-lift-invariance-def
      by metis
  next
    assume b ∈ defer m A p
    thus b ∈ defer m A q
      using defer-m lifted-a monotone-m
      unfolding defer-lift-invariance-def
      by metis
  qed
qed
moreover have ∀ x ∈ A − B. mod-contains-result m (m ‖↑ n) A q x
  using alts max-agg-rej-fst-imp-seq-contained monotone-m monotone-n f-profs
  unfolding defer-lift-invariance-def
  by metis
ultimately have ∀ x ∈ A − B. prof-contains-result (m ‖↑ n) A p q x
    unfolding mod-contains-result-def mod-contains-result-sym-def
              prof-contains-result-def
```

```
    by simp
  thus ?thesis
    using prof-contains-result-of-comps-for-elems-in-B
    by blast
  qed
  thus (m ‖↑ n) A p = (m ‖↑ n) A q
    using compatible f-profs eq-alts-in-profs-imp-eq-results max-par-comp-sound
    unfolding disjoint-compatibility-def
    by metis
qed

lemma par-comp-rej-card:
  fixes
    m :: 'a Electoral-Module and
    n :: 'a Electoral-Module and
    A :: 'a set and
    p :: 'a Profile and
    c :: nat
  assumes
    compatible: disjoint-compatibility m n and
    f-prof: finite-profile A p and
    reject-sum: card (reject m A p) + card (reject n A p) = card A + c
  shows card (reject (m ‖↑ n) A p) = c
proof −
  obtain B where
    alt-set: B ⊆ A ∧
        (∀ a ∈ B. indep-of-alt m A a ∧
           (∀ q. finite-profile A q ⟶ a ∈ reject m A q)) ∧
        (∀ a ∈ A − B. indep-of-alt n A a ∧
           (∀ q. finite-profile A q ⟶ a ∈ reject n A q))
    using compatible f-prof
    unfolding disjoint-compatibility-def
    by metis
  have reject-representation:
    reject (m ‖↑ n) A p = (reject m A p) ∩ (reject n A p)
    using f-prof compatible max-agg-rej-intersect
    unfolding disjoint-compatibility-def
    by metis
  have electoral-module m ∧ electoral-module n
    using compatible
    unfolding disjoint-compatibility-def
    by simp
  hence subsets: (reject m A p) ⊆ A ∧ (reject n A p) ⊆ A
    by (simp add: f-prof reject-in-alts)
  hence finite (reject m A p) ∧ finite (reject n A p)
    using rev-finite-subset f-prof
    by metis
  hence card-difference:
    card (reject (m ‖↑ n) A p) =
```

```
    card A + c − card ((reject m A p) ∪ (reject n A p))
    using card-Un-Int reject-representation reject-sum
    by fastforce
  have ∀ a ∈ A. a ∈ (reject m A p) ∨ a ∈ (reject n A p)
    using alt-set f-prof
    by blast
  hence A = reject m A p ∪ reject n A p
    using subsets
    by force
  thus card (reject (m ‖↑ n) A p) = c
    using card-difference
    by simp
qed
```

Using the max-aggregator for composing two compatible modules in parallel, whereof the first one is non-electing and defers exactly one alternative, and the second one rejects exactly two alternatives, the composition results in an electoral module that eliminates exactly one alternative.

```
theorem par-comp-elim-one[simp]:
  fixes
    m :: 'a Electoral-Module and
    n :: 'a Electoral-Module
  assumes
    defers-m-one: defers 1 m and
    non-elec-m: non-electing m and
    rejec-n-two: rejects 2 n and
    disj-comp: disjoint-compatibility m n
  shows eliminates 1 (m ‖↑ n)
proof (unfold eliminates-def, safe)
  have electoral-module m
    using non-elec-m
    unfolding non-electing-def
    by simp
  moreover have electoral-module n
    using rejec-n-two
    unfolding rejects-def
    by simp
  ultimately show electoral-module (m ‖↑ n)
    by simp
next
  fix
    A :: 'a set and
    p :: 'a Profile
  assume
    min-card-two: 1 < card A and
    prof-A: profile A p
  hence card-geq-one: card A ≥ 1
    by presburger
  have fin-A: finite A
```

```
    using min-card-two card.infinite not-one-less-zero
    by metis
  have module: electoral-module m
    using non-elec-m
    unfolding non-electing-def
    by simp
  have elec-card-zero: card (elect m A p) = 0
    using prof-A non-elec-m card-eq-0-iff
    unfolding non-electing-def
    by simp
  moreover from card-geq-one
  have def-card-one: card (defer m A p) = 1
    using defers-m-one module prof-A fin-A
    unfolding defers-def
    by simp
  ultimately have card-reject-m: card (reject m A p) = card A − 1
  proof −
    have well-formed A (elect m A p, reject m A p, defer m A p)
      using prof-A module
      unfolding electoral-module-def
      by simp
    hence
      card A = card (elect m A p) + card (reject m A p) + card (defer m A p)
      using result-count fin-A
      by blast
    thus ?thesis
      using def-card-one elec-card-zero
      by simp
  qed
  have card A ≥ 2
    using min-card-two
    by simp
  hence card (reject n A p) = 2
    using prof-A rejec-n-two fin-A
    unfolding rejects-def
    by blast
  moreover from this
  have card (reject m A p) + card (reject n A p) = card A + 1
    using card-reject-m card-geq-one
    by linarith
  ultimately show card (reject (m ‖↑ n) A p) = 1
    using disj-comp prof-A card-reject-m par-comp-rej-card fin-A
    by blast
qed

end
```

4.7 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.

4.7.1 Definition

fun *elector* :: *'a Electoral-Module* ⇒ *'a Electoral-Module* **where**
  elector m = (*m* ▷ *elect-module*)

4.7.2 Auxiliary Lemmas

lemma *elector-seqcomp-assoc*:
  fixes
    a :: *'a Electoral-Module* **and**
    b :: *'a Electoral-Module*
  shows (*a* ▷ (*elector b*)) = (*elector* (*a* ▷ *b*))
  unfolding *elector.simps elect-module.simps sequential-composition.simps*
  using *boolean-algebra-cancel.sup2 fst-eqD snd-eqD sup-commute*
  by (*metis* (*no-types*, *opaque-lifting*))

4.7.3 Soundness

theorem *elector-sound*[*simp*]:
  fixes *m* :: *'a Electoral-Module*
  assumes *electoral-module m*
  shows *electoral-module* (*elector m*)
  using *assms*
  by *simp*

4.7.4 Electing

theorem *elector-electing*[*simp*]:
  fixes *m* :: *'a Electoral-Module*
  assumes
    module-m: *electoral-module m* **and**
    non-block-m: *non-blocking m*
  shows *electing* (*elector m*)
proof −

```
  obtain
    A :: 'a Electoral-Module ⇒ 'a set and
    p :: 'a Electoral-Module ⇒ 'a Profile where
    ∀ m'.
      (¬ electing m' ∧ electoral-module m' ⟶ elect m' (A m') (p m') = {})
      ∧ (electing m'
          ⟶ (∀ A p. A ≠ {} ∧ finite-profile A p ⟶ elect m' A p ≠ {}))
    using electing-def
    by metis
  moreover have electoral-module (elector m)
    by (simp add: module-m)
  moreover from this
  have ¬ electing (elector m)
        ⟶ elect (elector m) (A (elector m)) (p (elector m)) ≠ {}
    using Un-empty-left boolean-algebra.disj-zero-right fst-conv non-block-m
          result-presv-alts seq-comp-def-then-elect-elec-set sup-bot.eq-neutr-iff
    unfolding elect-module.simps elector.simps electing-def non-blocking-def
    by (metis (no-types, lifting))
  ultimately show ?thesis
    using non-block-m
    unfolding elector.simps
    by metis
qed
```

4.7.5 Composition Rule

If m is defer-Condorcet-consistent, then elector(m) is Condorcet consistent.

```
lemma dcc-imp-cc-elector:
  fixes m :: 'a Electoral-Module
  assumes defer-condorcet-consistency m
  shows condorcet-consistency (elector m)
proof (unfold defer-condorcet-consistency-def condorcet-consistency-def, safe)
  show electoral-module (elector m)
    using assms elector-sound
    unfolding defer-condorcet-consistency-def
    by metis
next
  fix
    A :: 'a set and
    p :: 'a Profile and
    w :: 'a
  assume c-win: condorcet-winner A p w
  have fin-A: finite A
    using condorcet-winner.simps c-win
    by metis
  have prof-A: profile A p
    using c-win
    by simp
  have max-card-w: ∀ y ∈ A − {w}.
```

```
      card {i. i < length p ∧ (w, y) ∈ (p!i)} <
        card {i. i < length p ∧ (y, w) ∈ (p!i)}
  using c-win
  by simp
have rej-is-complement: reject m A p = A − (elect m A p ∪ defer m A p)
  using double-diff sup-bot.left-neutral Un-upper2 assms fin-A prof-A
      defer-condorcet-consistency-def elec-and-def-not-rej reject-in-alts
  by (metis (no-types, opaque-lifting))
have subset-in-win-set: elect m A p ∪ defer m A p ⊆
    {e ∈ A. e ∈ A ∧ (∀ x ∈ A − {e}.
     card {i. i < length p ∧ (e, x) ∈ p!i} < card {i. i < length p ∧ (x, e) ∈ p!i})}
proof (safe-step)
  fix x :: 'a
  assume x-in-elect-or-defer: x ∈ elect m A p ∪ defer m A p
  hence x-eq-w: x = w
    using Diff-empty Diff-iff assms cond-winner-unique c-win fin-A insert-iff
        prod.sel sup-bot.left-neutral
    unfolding defer-condorcet-consistency-def
    by (metis (mono-tags, lifting))
  have ∀ x. x ∈ elect m A p ⟶ x ∈ A
    using fin-A prof-A assms elect-in-alts in-mono
    unfolding defer-condorcet-consistency-def
    by metis
  moreover have ∀ x. x ∈ defer m A p ⟶ x ∈ A
    using fin-A prof-A assms defer-in-alts in-mono
    unfolding defer-condorcet-consistency-def
    by metis
  ultimately have x ∈ A
    using x-in-elect-or-defer
    by auto
  thus x ∈ {e ∈ A. e ∈ A ∧
        (∀ x ∈ A − {e}.
          card {i. i < length p ∧ (e, x) ∈ p!i} <
            card {i. i < length p ∧ (x, e) ∈ p!i})}
    using x-eq-w max-card-w
    by auto
qed
moreover have
  {e ∈ A. e ∈ A ∧
     (∀ x ∈ A − {e}.
        card {i. i < length p ∧ (e, x) ∈ p!i} <
          card {i. i < length p ∧ (x, e) ∈ p!i})}
      ⊆ elect m A p ∪ defer m A p
proof (safe)
  fix x :: 'a
  assume
    x-not-in-defer: x ∉ defer m A p and
    x ∈ A and
    ∀ x' ∈ A − {x}.
```

```
      card {i. i < length p ∧ (x, x') ∈ p!i} <
        card {i. i < length p ∧ (x', x) ∈ p!i}
    hence c-win-x: condorcet-winner A p x
      using fin-A prof-A
      by simp
    have (electoral-module m ∧ ¬ defer-condorcet-consistency m ⟶
          (∃ A rs a. condorcet-winner A rs a ∧
            m A rs ≠ ({}, A − defer m A rs, {a ∈ A. condorcet-winner A rs a}))) ∧
        (defer-condorcet-consistency m ⟶
          (∀ A rs a. finite A ⟶ condorcet-winner A rs a ⟶
            m A rs = ({}, A − defer m A rs, {a ∈ A. condorcet-winner A rs a})))
      unfolding defer-condorcet-consistency-def
      by blast
    hence m A p = ({}, A − defer m A p, {a ∈ A. condorcet-winner A p a})
      using c-win-x assms fin-A
      by blast
    thus x ∈ elect m A p
    using assms x-not-in-defer fin-A cond-winner-unique defer-condorcet-consistency-def
          insertCI snd-conv c-win-x
      by (metis (no-types, lifting))
  qed
  ultimately have
    elect m A p ∪ defer m A p =
      {e ∈ A. e ∈ A ∧
        (∀ x ∈ A − {e}.
          card {i. i < length p ∧ (e, x) ∈ p!i} <
            card {i. i < length p ∧ (x, e) ∈ p!i})}
    by blast
  thus elector m A p =
        ({e ∈ A. condorcet-winner A p e}, A − elect (elector m) A p, {})
    using fin-A prof-A rej-is-complement
    by simp
qed

end
```

4.8 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.

4.8.1 Definition

fun *iter* :: *'a Electoral-Module* ⇒ *'a Electoral-Module* **where**
  iter m =
    (*let t* = *defer-equal-condition 1 in*
      (*m* ↻$_t$))

abbreviation *defer-one-loop* ::
  'a Electoral-Module ⇒ *'a Electoral-Module*
    (-↻$_{\exists\,!d}$ *50*) **where**
  m ↻$_{\exists\,!d}$ ≡ *iter m*

fun *iterelect* :: *'a Electoral-Module* ⇒ *'a Electoral-Module* **where**
  iterelect m = *elector* (*m* ↻$_{\exists\,!d}$)

end

Chapter 5

Voting Rules

5.1 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.

5.1.1 Definition

fun *plurality-rule* :: *'a Electoral-Module* **where**
  plurality-rule A p = *elector plurality A p*

fun *plurality-rule'* :: *'a Electoral-Module* **where**
  plurality-rule' A p =
    ({*a* ∈ *A*. ∀ *x* ∈ *A*. *win-count p x* ≤ *win-count p a*},
     {*a* ∈ *A*. ∃ *x* ∈ *A*. *win-count p x* > *win-count p a*},
     {})

lemma *plurality-revision-equiv*:
  fixes
    A :: *'a set* **and**
    p :: *'a Profile*
  shows *plurality' A p* = (*plurality-rule'*↓) *A p*
proof (*unfold plurality-rule'.simps plurality'.simps revision-composition.simps*,
    standard, *clarsimp*, *standard*, *safe*)
  fix
    a :: *'a* **and**
    b :: *'a*
  assume

```
    b ∈ A and
    card {i. i < length p ∧ above (p!i) a = {a}} <
      card {i. i < length p ∧ above (p!i) b = {b}} and
    ∀ a' ∈ A. card {i. i < length p ∧ above (p!i) a' = {a'}} ≤
      card {i. i < length p ∧ above (p!i) a = {a}}
  thus False
    using leD
    by blast
next
  fix
    a :: 'a and
    b :: 'a
  assume
    b ∈ A and
    ¬ card {i. i < length p ∧ above (p!i) b = {b}} ≤
      card {i. i < length p ∧ above (p!i) a = {a}}
  thus ∃ x ∈ A.
        card {i. i < length p ∧ above (p!i) a = {a}}
        < card {i. i < length p ∧ above (p!i) x = {x}}
    using linorder-not-less
    by blast
qed

lemma plurality-elim-equiv:
  fixes
    A :: 'a set and
    p :: 'a Profile
  assumes
    A ≠ {} and
    finite-profile A p
  shows plurality A p = (plurality-rule'↓) A p
  using assms plurality-mod-elim-equiv plurality-revision-equiv
  by (metis (full-types))
```

5.1.2 Soundness

```
theorem plurality-rule-sound[simp]: electoral-module plurality-rule
  unfolding plurality-rule.simps
  using elector-sound plurality-sound
  by metis

theorem plurality-rule'-sound[simp]: electoral-module plurality-rule'
proof (unfold electoral-module-def, safe)
  fix
    A :: 'a set and
    p :: 'a Profile
  have disjoint3 (
      {a ∈ A. ∀ a' ∈ A. win-count p a' ≤ win-count p a},
      {a ∈ A. ∃ a' ∈ A. win-count p a < win-count p a'},
```

{})
by *auto*
moreover have
{$a \in A.\ \forall\ x \in A.\ \textit{win-count}\ p\ x \leq \textit{win-count}\ p\ a$} ∪
{$a \in A.\ \exists\ x \in A.\ \textit{win-count}\ p\ a < \textit{win-count}\ p\ x$} = A
using *not-le-imp-less*
by *auto*
ultimately show *well-formed A* (*plurality-rule′ A p*)
by *simp*
qed

5.1.3 Electing

lemma *plurality-rule-elect-non-empty*:
fixes
A :: *'a set* **and**
p :: *'a Profile*
assumes
A-non-empty: $A \neq \{\}$ **and**
fin-prof-A: *finite-profile A p*
shows *elect plurality-rule A p* ≠ {}
proof
assume *plurality-elect-none*: *elect plurality-rule A p* = {}
obtain *max* **where**
max: *max* = *Max* (*win-count p* ‘ *A*)
by *simp*
then obtain *a* **where**
max-a: *win-count p a* = *max* ∧ *a* ∈ *A*
using *Max-in A-non-empty fin-prof-A empty-is-image finite-imageI imageE*
by (*metis* (*no-types*, *lifting*))
hence ∀ *a′* ∈ *A*. *win-count p a′* ≤ *win-count p a*
using *fin-prof-A max*
by *simp*
moreover have *a* ∈ *A*
using *max-a*
by *simp*
ultimately have *a* ∈ {*a′* ∈ *A*. ∀ *c* ∈ *A*. *win-count p c* ≤ *win-count p a′*}
by *blast*
hence *a* ∈ *elect plurality-rule A p*
by *auto*
thus *False*
using *plurality-elect-none all-not-in-conv*
by *metis*
qed

The plurality module is electing.

theorem *plurality-rule-electing*[*simp*]: *electing plurality-rule*
proof (*unfold electing-def*, *safe*)
show *electoral-module plurality-rule*

using *plurality-rule-sound*
by *simp*
next
fix
$A :: 'a\ set$ **and**
$p :: 'a\ Profile$ **and**
$a :: 'a$
assume
fin-A: *finite A* **and**
prof-p: *profile A p* **and**
elect-none: *elect plurality-rule A p* $= \{\}$ **and**
a-in-A: $a \in A$
have $\forall\ A\ p.\ A \neq \{\} \wedge$ *finite-profile A p* $\longrightarrow$ *elect plurality-rule A p* $\neq \{\}$
using *plurality-rule-elect-non-empty*
by (*metis* (*no-types*))
hence *empty-A*: $A = \{\}$
using *fin-A prof-p elect-none*
by (*metis* (*no-types*))
thus $a \in \{\}$
using *a-in-A*
by *simp*
qed

5.1.4 Property

lemma *plurality-rule-inv-mono-eq*:
fixes
$A :: 'a\ set$ **and**
$p :: 'a\ Profile$ **and**
$q :: 'a\ Profile$ **and**
$a :: 'a$
assumes
elect-a: $a \in$ *elect plurality-rule A p* **and**
lift-a: *lifted A p q a*
shows *elect plurality-rule A q* = *elect plurality-rule A p* $\vee$
elect plurality-rule A q $= \{a\}$
proof −
have $a \in$ *elect* (*elector plurality*) *A p*
using *elect-a*
by *simp*
moreover have *eq-p*: *elect* (*elector plurality*) *A p* = *defer plurality A p*
by *simp*
ultimately have $a \in$ *defer plurality A p*
by *blast*
hence *defer plurality A q* = *defer plurality A p* $\vee$ *defer plurality A q* $= \{a\}$
using *lift-a plurality-def-inv-mono-alts*
by *metis*
moreover have *elect* (*elector plurality*) *A q* = *defer plurality A q*
by *simp*

ultimately show
elect plurality-rule A q = elect plurality-rule A p ∨
elect plurality-rule A q = {a}
using *eq-p*
by *simp*
qed

The plurality rule is invariant-monotone.

theorem *plurality-rule-inv-mono*[*simp*]: *invariant-monotonicity plurality-rule*
proof (*unfold invariant-monotonicity-def*, *intro conjI impI allI*)
show *electoral-module plurality-rule*
by *simp*
next
fix
A :: *'a set* **and**
p :: *'a Profile* **and**
q :: *'a Profile* **and**
a :: *'a*
assume *a ∈ elect plurality-rule A p ∧ Profile.lifted A p q a*
thus *elect plurality-rule A q = elect plurality-rule A p ∨*
elect plurality-rule A q = {a}
using *plurality-rule-inv-mono-eq*
by *metis*
qed

end

5.2 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.

5.2.1 Definition

fun *borda-rule* :: *'a Electoral-Module* **where**
borda-rule A p = elector borda A p

fun *borda-rule*$_{\mathcal{R}}$ :: *'a Electoral-Module* **where**
borda-rule$_{\mathcal{R}}$ *A p* = *swap-*$\mathcal{R}$ *unanimity A p*

5.2.2 Soundness

theorem *borda-rule-sound*: *electoral-module borda-rule*
unfolding *borda-rule.simps*
using *elector-sound borda-sound*
by *metis*

theorem *borda-rule*$_{\mathcal{R}}$*-sound*: *electoral-module borda-rule*$_{\mathcal{R}}$
unfolding *borda-rule*$_{\mathcal{R}}$*.simps swap-*$\mathcal{R}$*.simps*
using $\mathcal{R}$*-sound*
by *metis*

5.2.3 Anonymity Property

theorem *borda-rule*$_{\mathcal{R}}$*-anonymous*: *anonymity borda-rule*$_{\mathcal{R}}$
proof (*unfold borda-rule*$_{\mathcal{R}}$*.simps swap-*$\mathcal{R}$*.simps*)
let *?swap-dist* = *votewise-distance swap l-one*
from *l-one-is-sym*
have *distance-anonymity ?swap-dist*
using *symmetric-norm-imp-distance-anonymous*[*of l-one*]
by *simp*
with *unanimity-anonymous*
show *anonymity* (*distance-*$\mathcal{R}$ *?swap-dist unanimity*)
using *anonymous-distance-and-consensus-imp-rule-anonymity*
by *metis*
qed

end

5.3 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.

5.3.1 Definition

fun *pairwise-majority-rule* :: *'a Electoral-Module* **where**

pairwise-majority-rule A p = elector condorcet A p

fun *condorcet′* :: *'a Electoral-Module* **where**
condorcet′ A p =
(((*min-eliminator condorcet-score*) $\circlearrowleft_{\exists!d}$) *A p*

fun *pairwise-majority-rule′* :: *'a Electoral-Module* **where**
pairwise-majority-rule′ A p = iterelect condorcet′ A p

5.3.2 Soundness

theorem *pairwise-majority-rule-sound*: *electoral-module pairwise-majority-rule*
unfolding *pairwise-majority-rule.simps*
using *condorcet-sound elector-sound*
by *metis*

theorem *condorcet′-rule-sound*: *electoral-module condorcet′*
unfolding *condorcet′.simps*
by (*simp add*: *loop-comp-sound*)

theorem *pairwise-majority-rule′-sound*: *electoral-module pairwise-majority-rule′*
unfolding *pairwise-majority-rule′.simps*
using *condorcet′-rule-sound elector-sound iter.simps iterelect.simps loop-comp-sound*
by *metis*

5.3.3 Condorcet Consistency Property

theorem *condorcet-condorcet*: *condorcet-consistency pairwise-majority-rule*
proof (*unfold pairwise-majority-rule.simps*)
show *condorcet-consistency* (*elector condorcet*)
using *condorcet-is-dcc dcc-imp-cc-elector*
by *metis*
qed

end

5.4 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.

5.4.1 Definition

fun *copeland-rule* :: *'a Electoral-Module* **where**
  copeland-rule A p = *elector copeland A p*

5.4.2 Soundness

theorem *copeland-rule-sound*: *electoral-module copeland-rule*
  unfolding *copeland-rule.simps*
  using *elector-sound copeland-sound*
  by *metis*

5.4.3 Condorcet Consistency Property

theorem *copeland-condorcet*: *condorcet-consistency copeland-rule*
proof (*unfold copeland-rule.simps*)
  show *condorcet-consistency* (*elector copeland*)
    using *copeland-is-dcc dcc-imp-cc-elector*
    by *metis*
qed

end

5.5 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.

5.5.1 Definition

fun *minimax-rule* :: *'a Electoral-Module* **where**
  minimax-rule A p = *elector minimax A p*

5.5.2 Soundness

theorem *minimax-rule-sound*: *electoral-module minimax-rule*
  unfolding *minimax-rule.simps*
  using *elector-sound minimax-sound*
  by *metis*

5.5.3 Condorcet Consistency Property

theorem *minimax-condorcet*: *condorcet-consistency minimax-rule*
proof (*unfold minimax-rule.simps*)
  show *condorcet-consistency* (*elector minimax*)
    using *minimax-is-dcc dcc-imp-cc-elector*
    by *metis*
qed

end

5.6 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.

5.6.1 Definition

declare *seq-comp-alt-eq*[*simp*]

fun *black* :: *'a Electoral-Module* **where**
  black A p = (*condorcet* ▷ *borda*) *A p*

fun *blacks-rule* :: *'a Electoral-Module* **where**
  blacks-rule A p = *elector black A p*

declare *seq-comp-alt-eq*[*simp del*]

5.6.2 Soundness

theorem *blacks-sound*: *electoral-module black*
  unfolding *black.simps*
  using *seq-comp-sound condorcet-sound borda-sound*
  by *metis*

theorem *blacks-rule-sound*: *electoral-module blacks-rule*
  unfolding *blacks-rule.simps*
  using *blacks-sound elector-sound*
  by *metis*

5.6.3 Condorcet Consistency Property

theorem *black-is-dcc*: *defer-condorcet-consistency black*
unfolding *black.simps*
using *condorcet-is-dcc borda-mod-non-blocking borda-mod-non-electing seq-comp-dcc*
by *metis*

theorem *black-condorcet*: *condorcet-consistency blacks-rule*
unfolding *blacks-rule.simps*
using *black-is-dcc dcc-imp-cc-elector*
by *metis*

end

5.7 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.

5.7.1 Definition

fun *nanson-baldwin-rule* :: *'a Electoral-Module* **where**
nanson-baldwin-rule A p =
((*min-eliminator borda-score*) $\circlearrowleft_{\exists !d}$) *A p*

5.7.2 Soundness

theorem *nanson-baldwin-rule-sound*: *electoral-module nanson-baldwin-rule*
unfolding *nanson-baldwin-rule.simps*
by (*simp add*: *loop-comp-sound*)

end

5.8 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.

5.8.1 Definition

fun *classic-nanson-rule* :: *'a Electoral-Module* **where**
classic-nanson-rule A p =
((*leq-average-eliminator borda-score*) $\circlearrowleft_{\exists !d}$) *A p*

5.8.2 Soundness

theorem *classic-nanson-rule-sound*: *electoral-module classic-nanson-rule*
unfolding *classic-nanson-rule.simps*
by (*simp add*: *loop-comp-sound*)

end

5.9 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.

5.9.1 Definition

fun *schwartz-rule* :: *'a Electoral-Module* **where**
schwartz-rule A p =
((*less-average-eliminator borda-score*) $\circlearrowleft_{\exists !d}$) *A p*

5.9.2 Soundness

theorem *schwartz-rule-sound*: *electoral-module schwartz-rule*

unfolding *schwartz-rule.simps*
by (*simp add*: *loop-comp-sound*)

end

5.10 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.

5.10.1 Definition

fun *smc* :: *'a Preference-Relation* ⇒ *'a Electoral-Module* **where**
smc x A p =
((*elector* ((((*pass-module 2 x*) ▷ ((*plurality-rule*↓) ▷ (*pass-module 1 x*))) ‖↑
(*drop-module 2 x*)) ⟳∃ !*d*)) *A p*)

5.10.2 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*
assumes *linear-order x*
shows *electoral-module* (*smc x*)
proof (*unfold electoral-module-def*, *simp*, *safe*, *simp-all*)
fix
A :: *'a set* **and**
p :: *'a Profile* **and**
x' :: *'a*
let *?a* = *max-aggregator*
let *?t* = *defer-equal-condition*

```
  let ?smc =
    pass-module 2 x ▷
      ((plurality-rule↓) ▷ pass-module (Suc 0) x) ∥? a
        drop-module 2 x ⟲? t (Suc 0)
  assume
    profile A p and
    x' ∈ reject (?smc) A p and
    x' ∈ elect (?smc) A p
  thus False
    using IntI drop-mod-sound emptyE loop-comp-sound max-agg-sound assms
          par-comp-sound pass-mod-sound plurality-rule-sound rev-comp-sound
          result-disj seq-comp-sound
    by metis
next
  fix
    A :: 'a set and
    p :: 'a Profile and
    x' :: 'a
  let ?a = max-aggregator
  let ?t = defer-equal-condition
  let ?smc =
    pass-module 2 x ▷
      ((plurality-rule↓) ▷ pass-module (Suc 0) x) ∥? a
        drop-module 2 x ⟲? t (Suc 0)
  assume
    profile A p and
    x' ∈ reject (?smc) A p and
    x' ∈ defer (?smc) A p
  thus False
    using IntI assms result-disj emptyE drop-mod-sound loop-comp-sound
          max-agg-sound par-comp-sound pass-mod-sound plurality-rule-sound
          rev-comp-sound seq-comp-sound
    by metis
next
  fix
    A :: 'a set and
    p :: 'a Profile and
    x' :: 'a
  let ?a = max-aggregator
  let ?t = defer-equal-condition
  let ?smc =
    pass-module 2 x ▷
      ((plurality-rule↓) ▷ pass-module (Suc 0) x) ∥? a
        drop-module 2 x ⟲? t (Suc 0)
  assume
    profile A p and
    x' ∈ elect (?smc) A p
  thus x' ∈ A
    using drop-mod-sound elect-in-alts in-mono assms loop-comp-sound
```

```
      max-agg-sound par-comp-sound pass-mod-sound plurality-rule-sound
      rev-comp-sound seq-comp-sound
  by metis
next
 fix
  A :: 'a set and
  p :: 'a Profile and
  x' :: 'a
 let ?a = max-aggregator
 let ?t = defer-equal-condition
 let ?smc =
  pass-module 2 x ▷
    ((plurality-rule↓) ▷ pass-module (Suc 0) x) ∥? a
     drop-module 2 x ↻? t (Suc 0)
 assume
  profile A p and
  x' ∈ defer (?smc) A p
 thus x' ∈ A
  using drop-mod-sound defer-in-alts in-mono assms loop-comp-sound
      max-agg-sound par-comp-sound pass-mod-sound plurality-rule-sound
      rev-comp-sound seq-comp-sound
  by (metis (no-types, lifting))
next
 fix
  A :: 'a set and
  p :: 'a Profile and
  x' :: 'a
 let ?a = max-aggregator
 let ?t = defer-equal-condition
 let ?smc =
  pass-module 2 x ▷
    ((plurality-rule↓) ▷ pass-module (Suc 0) x) ∥? a
     drop-module 2 x ↻? t (Suc 0)
 assume
  prof-A: profile A p and
  reject-x': x' ∈ reject (?smc) A p
 have electoral-module (plurality-rule↓)
  by simp
 moreover have electoral-module (drop-module 2 x)
  by simp
 ultimately show x' ∈ A
  using reject-x' prof-A in-mono assms reject-in-alts loop-comp-sound
      max-agg-sound par-comp-sound pass-mod-sound seq-comp-sound
  by (metis (no-types))
next
 fix
  A :: 'a set and
  p :: 'a Profile and
  x' :: 'a
```

```
  let ?a = max-aggregator
  let ?t = defer-equal-condition
  let ?smc =
    pass-module 2 x ▷
      ((plurality-rule↓) ▷ pass-module (Suc 0) x) ‖?a
        drop-module 2 x ⟲?t (Suc 0)
  assume
    profile A p and
    x' ∈ A and
    x' ∉ defer (?smc) A p and
    x' ∉ reject (?smc) A p
  thus x' ∈ elect (?smc) A p
    using assms electoral-mod-defer-elem drop-mod-sound loop-comp-sound
          max-agg-sound par-comp-sound pass-mod-sound plurality-rule-sound
          rev-comp-sound seq-comp-sound
    by metis
qed
```

5.10.3 Electing

The sequential majority comparison electoral module is electing. This property is needed to convert electoral modules to a social choice function. Apart from the very last proof step, it is a part of the monotonicity proof below.

```
theorem smc-electing:
  fixes x :: 'a Preference-Relation
  assumes linear-order x
  shows electing (smc x)
proof −
  let ?pass2 = pass-module 2 x
  let ?tie-breaker = (pass-module 1 x)
  let ?plurality-defer = (plurality-rule↓) ▷ ?tie-breaker
  let ?compare-two = ?pass2 ▷ ?plurality-defer
  let ?drop2 = drop-module 2 x
  let ?eliminator = ?compare-two ‖↑ ?drop2
  let ?loop =
    let t = defer-equal-condition 1 in (?eliminator ⟲t)

  have 00011: non-electing (plurality-rule↓)
    by simp
  have 00012: non-electing ?tie-breaker
    using assms
    by simp
  have 00013: defers 1 ?tie-breaker
    using assms pass-one-mod-def-one
    by simp
  have 20000: non-blocking (plurality-rule↓)
    by simp

  have 0020: disjoint-compatibility ?pass2 ?drop2
```

using *assms*
by *simp*
have *1000*: *non-electing ?pass2*
using *assms*
by *simp*
have *1001*: *non-electing ?plurality-defer*
using *00011 00012*
by *simp*
have *2000*: *non-blocking ?pass2*
using *assms*
by *simp*
have *2001*: *defers 1 ?plurality-defer*
using *20000 00011 00013 seq-comp-def-one*
by *blast*

have *002*: *disjoint-compatibility ?compare-two ?drop2*
using *assms 0020*
by *simp*
have *100*: *non-electing ?compare-two*
using *1000 1001*
by *simp*
have *101*: *non-electing ?drop2*
using *assms*
by *simp*
have *102*: *agg-conservative max-aggregator*
by *simp*
have *200*: *defers 1 ?compare-two*
using *2000 1000 2001 seq-comp-def-one*
by *simp*
have *201*: *rejects 2 ?drop2*
using *assms*
by *simp*

have *10*: *non-electing ?eliminator*
using *100 101 102*
by *simp*
have *20*: *eliminates 1 ?eliminator*
using *200 100 201 002 par-comp-elim-one*
by *simp*

have *2*: *defers 1 ?loop*
using *10 20*
by *simp*
have *3*: *electing elect-module*
by *simp*

show *?thesis*
using *2 3 assms seq-comp-electing smc-sound*
unfolding *Defer-One-Loop-Composition.iter.simps*

smc.simps elector.simps electing-def
by *metis*
qed

5.10.4 (Weak) Monotonicity Property

The following proof is a fully modular proof for weak monotonicity of sequential majority comparison. It is composed of many small steps.

theorem *smc-monotone*:
fixes *x* :: *'a Preference-Relation*
assumes *linear-order x*
shows *monotonicity* (*smc x*)
proof −
let *?pass2* = *pass-module 2 x*
let *?tie-breaker* = *pass-module 1 x*
let *?plurality-defer* = (*plurality-rule*↓) ▷ *?tie-breaker*
let *?compare-two* = *?pass2* ▷ *?plurality-defer*
let *?drop2* = *drop-module 2 x*
let *?eliminator* = *?compare-two* $\|_{\uparrow}$ *?drop2*
let *?loop* =
let t = *defer-equal-condition 1 in* (*?eliminator* $\circlearrowleft_t$)

have *00010*: *defer-invariant-monotonicity* (*plurality-rule*↓)
by *simp*
have *00011*: *non-electing* (*plurality-rule*↓)
by *simp*
have *00012*: *non-electing ?tie-breaker*
using *assms*
by *simp*
have *00013*: *defers 1 ?tie-breaker*
using *assms pass-one-mod-def-one*
by *simp*
have *00014*: *defer-monotonicity ?tie-breaker*
using *assms*
by *simp*
have *20000*: *non-blocking* (*plurality-rule*↓)
by *simp*

have *0000*: *defer-lift-invariance ?pass2*
using *assms*
by *simp*
have *0001*: *defer-lift-invariance ?plurality-defer*
using *00010 00011 00012 00013 00014*
by *simp*
have *0020*: *disjoint-compatibility ?pass2 ?drop2*
using *assms*
by *simp*
have *1000*: *non-electing ?pass2*
using *assms*

by *simp*
have *1001*: *non-electing ?plurality-defer*
using *00011 00012*
by *simp*
have *2000*: *non-blocking ?pass2*
using *assms*
by *simp*
have *2001*: *defers 1 ?plurality-defer*
using *20000 00011 00013 seq-comp-def-one*
by *blast*

have *000*: *defer-lift-invariance ?compare-two*
using *0000 0001*
by *simp*
have *001*: *defer-lift-invariance ?drop2*
using *assms*
by *simp*
have *002*: *disjoint-compatibility ?compare-two ?drop2*
using *assms 0020*
by *simp*

have *100*: *non-electing ?compare-two*
using *1000 1001*
by *simp*
have *101*: *non-electing ?drop2*
using *assms*
by *simp*
have *102*: *agg-conservative max-aggregator*
by *simp*
have *200*: *defers 1 ?compare-two*
using *2000 1000 2001 seq-comp-def-one*
by *simp*
have *201*: *rejects 2 ?drop2*
using *assms*
by *simp*

have *00*: *defer-lift-invariance ?eliminator*
using *000 001 002 par-comp-def-lift-inv*
by *blast*
have *10*: *non-electing ?eliminator*
using *100 101 102*
by *simp*
have *20*: *eliminates 1 ?eliminator*
using *200 100 201 002 par-comp-elim-one*
by *simp*

have *0*: *defer-lift-invariance ?loop*
using *00*
by *simp*

```
  have 1: non-electing ?loop
    using 10
    by simp
  have 2: defers 1 ?loop
    using 10 20
    by simp
  have 3: electing elect-module
    by simp

  show ?thesis
    using 0 1 2 3 assms seq-comp-mono
    unfolding Electoral-Module.monotonicity-def elector.simps
              Defer-One-Loop-Composition.iter.simps
              smc-sound smc.simps
    by (metis (full-types))
qed

end
```

5.11 Kemeny Rule

```
theory Kemeny-Rule
  imports Compositional-Structures/Basic-Modules/Component-Types/Votewise-Distance-Rationalization
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.

5.11.1 Definition

```
fun kemeny-rule :: 'a Electoral-Module where
  kemeny-rule A p = swap-𝓡 strong-unanimity A p
```

5.11.2 Soundness

```
theorem kemeny-rule-sound: electoral-module kemeny-rule
  unfolding kemeny-rule.simps swap-𝓡.simps
  using 𝓡-sound
  by metis
```

5.11.3 Anonymity Property

```
theorem kemeny-rule-anonymous: anonymity kemeny-rule
```

proof (*unfold kemeny-rule.simps swap-$\mathcal{R}$.simps*)
  let *?swap-dist = votewise-distance swap l-one*
  have *distance-anonymity ?swap-dist*
    using *l-one-is-sym symmetric-norm-imp-distance-anonymous*[*of l-one*]
    by *simp*
  thus *anonymity* (*distance-$\mathcal{R}$?swap-dist strong-unanimity*)
   using *strong-unanimity-anonymous anonymous-distance-and-consensus-imp-rule-anonymity*
    by *metis*
qed

end

Bibliography

[1] K. Diekhoff, M. Kirsten, and J. Krämer. Formal property-oriented design of voting rules using composable modules. In S. Pekeč and K. Venable, editors, *6th International Conference on Algorithmic Decision Theory (ADT 2019)*, volume 11834 of *Lecture Notes in Artificial Intelligence*, pages 164–166. Springer, 2019.

[2] K. Diekhoff, M. Kirsten, and J. Krämer. Verified construction of fair voting rules. In M. 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.