Randomised Social Choice

Manuel Eberl

April 20, 2020

Abstract

This work contains a formalisation of basic Randomised Social Choice, including Stochastic Dominance and Social Decision Schemes (SDSs) along with some of their most important properties (Anonymity, Neutrality, SD-Efficiency, SD-Strategy-Proofness) and two particular SDSs – Random Dictatorship and Random Serial Dictatorship (with proofs of the properties that they satisfy). Many important properties of these concepts are also proven such as the two equivalent characterisations of Stochastic Dominance and the fact that SD-efficiency of a lottery only depends on the support.

The entry also provides convenient commands to define Preference Profiles, prove their well-formedness, and automatically derive restrictions that sufficiently nice SDSs need to satisfy on the defined profiles. (cf. [1])

Currently, the formalisation focuses on weak preferences and Stochastic Dominance (SD), but it should be easy to extend it to other domains – such as strict preferences – or other lottery extensions – such as Bilinear Dominance or Pairwise Comparison.

Contents

1 Order Relations as Binary Predicates 4
  1.1 Basic Operations on Relations ......................... 4
  1.2 Preorders ........................................... 4
  1.3 Total preorders ....................................... 5
  1.4 Orders ............................................... 6
  1.5 Maximal elements .................................... 7
  1.6 Weak rankings ....................................... 9
  1.7 Rankings ............................................. 25

2 Preference Profiles 26
  2.1 Pareto dominance ..................................... 29
  2.2 Preferred alternatives ................................. 31
  2.3 Favourite alternatives ................................. 31
  2.4 Anonymous profiles ................................... 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5</td>
<td>Preference profiles from lists</td>
<td>35</td>
</tr>
<tr>
<td>2.6</td>
<td>Automatic evaluation of preference profiles</td>
<td>39</td>
</tr>
<tr>
<td>3</td>
<td>Auxiliary facts about PMFs</td>
<td>44</td>
</tr>
<tr>
<td>3.1</td>
<td>Definition of von Neumann–Morgenstern utility functions</td>
<td>45</td>
</tr>
<tr>
<td>4</td>
<td>Stochastic Dominance</td>
<td>48</td>
</tr>
<tr>
<td>4.1</td>
<td>Definition of Stochastic Dominance</td>
<td>48</td>
</tr>
<tr>
<td>4.2</td>
<td>Stochastic Dominance for preference profiles</td>
<td>51</td>
</tr>
<tr>
<td>4.3</td>
<td>SD efficient lotteries</td>
<td>52</td>
</tr>
<tr>
<td>4.4</td>
<td>Equivalence proof</td>
<td>54</td>
</tr>
<tr>
<td>4.5</td>
<td>Existence of SD-efficient lotteries</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>Social Decision Schemes</td>
<td>67</td>
</tr>
<tr>
<td>5.1</td>
<td>Basic Social Choice definitions</td>
<td>68</td>
</tr>
<tr>
<td>5.2</td>
<td>Social Decision Schemes</td>
<td>68</td>
</tr>
<tr>
<td>5.3</td>
<td>Anonymity</td>
<td>68</td>
</tr>
<tr>
<td>5.4</td>
<td>Neutrality</td>
<td>69</td>
</tr>
<tr>
<td>5.5</td>
<td>Ex-post efficiency</td>
<td>71</td>
</tr>
<tr>
<td>5.6</td>
<td>SD efficiency</td>
<td>72</td>
</tr>
<tr>
<td>5.7</td>
<td>Weak strategyproofness</td>
<td>73</td>
</tr>
<tr>
<td>5.8</td>
<td>Strong strategyproofness</td>
<td>74</td>
</tr>
<tr>
<td>6</td>
<td>Lowering Social Decision Schemes</td>
<td>75</td>
</tr>
<tr>
<td>7</td>
<td>Random Dictatorship</td>
<td>83</td>
</tr>
<tr>
<td>7.1</td>
<td>Anonymity</td>
<td>85</td>
</tr>
<tr>
<td>7.2</td>
<td>Neutrality</td>
<td>86</td>
</tr>
<tr>
<td>7.3</td>
<td>Strong strategyproofness</td>
<td>86</td>
</tr>
<tr>
<td>8</td>
<td>Random Serial Dictatorship</td>
<td>87</td>
</tr>
<tr>
<td>8.1</td>
<td>Auxiliary facts about RSD</td>
<td>89</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Pareto-equivalence classes</td>
<td>89</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Facts about RSD winners</td>
<td>90</td>
</tr>
<tr>
<td>8.2</td>
<td>Proofs of properties</td>
<td>95</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Well-definedness</td>
<td>95</td>
</tr>
<tr>
<td>8.2.2</td>
<td>RD extension</td>
<td>95</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Anonymity</td>
<td>95</td>
</tr>
<tr>
<td>8.2.4</td>
<td>Neutrality</td>
<td>96</td>
</tr>
<tr>
<td>8.2.5</td>
<td>Ex-post efficiency</td>
<td>96</td>
</tr>
<tr>
<td>8.2.6</td>
<td>Strong strategy-proofness</td>
<td>97</td>
</tr>
<tr>
<td>9</td>
<td>Automatic definition of Preference Profiles</td>
<td>98</td>
</tr>
<tr>
<td>9.1</td>
<td>Automatic definition of preference profiles from tables</td>
<td>100</td>
</tr>
</tbody>
</table>
1 Order Relations as Binary Predicates

theory Order-Predicates
imports
  Main
  HOL-Library.Disjoint-Sets
  HOL-Library.Permutations
  List-Index.List-Index
begin

1.1 Basic Operations on Relations

The type of binary relations

\textbf{type-synonym} \ 'a relation = 'a \Rightarrow 'a \Rightarrow bool

\textbf{definition} map-relation :: ('a \Rightarrow 'b) \Rightarrow 'b relation \Rightarrow 'a relation where

\quad \text{map-relation} f R = (\lambda x y. R (f x) (f y))

\textbf{definition} restrict-relation :: 'a set \Rightarrow 'a relation \Rightarrow 'a relation where

\quad \text{restrict-relation} A R = (\lambda x y. x \in A \land y \in A \Rightarrow R x y)

\textbf{lemma} restrict-relation-restrict-relation \ [simp]:

\quad\text{restrict-relation} A (\text{restrict-relation} B R) = \text{restrict-relation} (A \cap B) R

\quadby (intro ext) (auto simp add: restrict-relation-def refl elim: trans)

\textbf{lemma} restrict-relation-empty \ [simp]: restrict-relation \{\} R = (\lam- -. False)

\quadby (simp add: restrict-relation-def)

\textbf{lemma} restrict-relation-UNIV \ [simp]: restrict-relation UNIV R = R

\quadby (simp add: restrict-relation-def)

1.2 Preorders

Preorders are reflexive and transitive binary relations.

\textbf{locale} preorder-on =

\quad\text{fixes} carrier :: 'a set
\quad\text{fixes} le :: 'a relation
\quad\text{assumes} not-outside: le x y \Longrightarrow x \in carrier le x y \Longrightarrow y \in carrier
\quad\text{assumes} refl: x \in carrier \Longrightarrow le x x
\quad\text{assumes} trans: le x y \Longrightarrow le y z \Longrightarrow le x z

\begin{proof}

\textbf{lemma} carrier-eq: carrier = \{x. le x x\}

\quad\text{using} not-outside refl \text{ by auto}

\textbf{lemma} preorder-on-map:

\quad preorder-on (f :: 'a carrier) (\text{map-relation} f le)

\quadby unfold-locales (auto dest: not-outside simp: map-relation-def refl elim: trans)
1.3 Total preorders

Total preorders are preorders where any two elements are comparable.

locale total-preorder-on = preorder-on +
  assumes total: \( x \in \text{carrier} \Rightarrow y \in \text{carrier} \Rightarrow \leq x y \lor \leq y x \)
begin

lemma total': \( \neg \leq x y \Rightarrow x \in \text{carrier} \Rightarrow y \in \text{carrier} \Rightarrow \leq y x \)
  using total[of x y] by blast

lemma total-preorder-on-map:
  total-preorder-on (f -\ ' carrier) (map-relation f le)
proof –
  interpret R': preorder-on f -\ ' carrier map-relation f le
  using preorder-on-map[of f] .
  show \( \text{?thesis} \) by unfold-locales (simp add: map-relation-def total)
qed

lemma total-preorder-on-restrict:
  total-preorder-on (carrier \( \cap A) (\text{restrict-relation} A \leq)
proof –
  interpret R': preorder-on carrier \( \cap A \text{ restrict-relation} A \leq
  by (rule preorder-on-restrict)
  from total show \( \text{?thesis} \)
  by unfold-locales (auto simp: restrict-relation-def)
qed

lemma total-preorder-on-restrict-subset:
  \( A \subseteq \text{carrier} \Rightarrow total-preorder-on A (\text{restrict-relation} A \leq)
using total-preorder-on-restrict[of A] by (simp add: Int-absorb1)

end

Some fancy notation for order relations
abbreviation (input) weakly-preferred :: 'a ⇒ 'a relation ⇒ 'a ⇒ bool
(- ≤[-] - [51,10,51] 60) where
a ≥[R] b ≡ R a b

definition strongly-preferred (- ≺[-] - [51,10,51] 60) where
a ≺[R] b ≡ (a ≤[R] b) ∧ (b ≤[R] a)
definition indifferent (- ∼[-] - [51,10,51] 60) where
a ∼[R] b ≡ (a ≤[R] b) ∧ (b ≤[R] a)
аббревиатура (вход) слабо-приоритет :: 'a ⇒ 'a отношение ⇒ 'a ⇒ bool
(- ⪯[-] - [51,10,51] 60) где
a ⪯[R] b ≡ R a b

definition strongly-not-preferred (- ≻[-] - [51,10,51] 60) where
a ≻[R] b ≡ (a ≺[R] b) ∧ (b ≺[R] a)
definition strongly-not-preferred (- ≻[-] - [51,10,51] 60) where
a ≻[R] b ≡ (a ≺[R] b) ∧ (b ≺[R] a)
context preorder-on
begin

lemma strict-trans: a ≺[le] b ⇒ b ≺[le] c ⇒ a ≺[le] c
unfolding strongly-preferred-def by (blast intro: trans)

lemma weak-strict-trans: a ≤[le] b ⇒ b ≤[le] c ⇒ a ≤[le] c
unfolding strongly-preferred-def by (blast intro: trans)

lemma strict-weak-trans: a ≺[le] b ⇒ b ≤[le] c ⇒ a ≺[le] c
unfolding strongly-preferred-def by (blast intro: trans)
end

lemma (in total-preorder-on) not-weakly-preferred-iff:
a ∈ carrier ⇒ b ∈ carrier ⇒ ¬a ≤[le] b ←→ b ≺[le] a
using total[of a b] by (auto simp: strongly-preferred-def)

lemma (in total-preorder-on) not-strongly-preferred-iff:
a ∈ carrier ⇒ b ∈ carrier ⇒ ¬a ≺[le] b ←→ b ≺[le] a
using total[of a b] by (auto simp: strongly-preferred-def)

1.4 Orders
locale order-on = preorder-on +
  assumes antisymmetric: le x y ⇒ le y x ⇒ x = y
locale linorder-on = order-on carrier le + total-preorder-on carrier le
for carrier le
1.5 Maximal elements

Maximal elements are elements in a preorder for which there exists no strictly greater element.

definition Max-wrt-among :: 'a relation ⇒ 'a set ⇒ 'a set where
  Max-wrt-among R A = {x ∈ A. R x x ∧ (∀ y ∈ A. R x y → R y x)}

lemma Max-wrt-among-cong:
  assumes restrict-relation A R = restrict-relation A R'
  shows Max-wrt-among R A = Max-wrt-among R' A
  proof –
    from assms have R x y ⇐⇒ R' x y if x ∈ A y ∈ A for x y
      using that by (auto simp: restrict-relation-def fun_eq_iff)
    thus ?thesis unfolding Max-wrt-among-def by blast
  qed

definition Max-wrt :: 'a relation ⇒ 'a set where
  Max-wrt R = Max-wrt-among R UNIV

lemma Max-wrt-altdef: Max-wrt R = {x. R x x ∧ (∀ y. R x y → R y x)}
  unfolding Max-wrt-def Max-wrt-among-def by simp

context preorder-on
begin

lemma Max-wrt-among-preorder:
  Max-wrt-among le A = {x ∈ carrier ∩ A. ∀ y ∈ carrier ∩ A. le x y → le y x}
  unfolding Max-wrt-among-def using not-outside refl by blast

lemma Max-wrt-preorder:
  Max-wrt le = {x ∈ carrier. ∀ y ∈ carrier. le x y → le y x}
  unfolding Max-wrt-altdef using not-outside refl by blast

lemma Max-wrt-among-subset:
  Max-wrt-among le A ⊆ carrier Max-wrt-among le A ⊆ A
  unfolding Max-wrt-among-preorder by auto

lemma Max-wrt-subset:
  Max-wrt le ⊆ carrier
  unfolding Max-wrt-preorder by auto

lemma Max-wrt-among-nonempty:
  assumes B ∩ carrier ≠ {} finite (B ∩ carrier)
  shows Max-wrt-among le B ≠ {}
  proof –
    define A where A = B ∩ carrier
    have A ⊆ carrier by (simp add: A-def)
    from assms(2,1)[folded A-def] this have {x ∈ A. (∀ y ∈ A. le x y → le y x)} ≠ {}
  qed


proof (induction A rule: finite-ne-induct)
case (singleton x)
  thus case by (auto simp: refl)
next
case (insert x A)
  then obtain y where y: y ∈ A ∧ z ∈ A ⇒ le y z ⇒ le z y by blast
  thus case using insert.prems
    by (cases le y x) (blast intro: trans)+
qed
thus thesis by (simp add: A-def Max-wrt-among-preorder Int-commute)
qed

lemma Max-wrt-nonempty:
carrier ≠ {} ⇒ finite carrier ⇒ Max-wrt le ≠ {}
using Max-wrt-among-nonempty[of UNIV] by (simp add: Max-wrt-def)

lemma Max-wrt-among-map-relation-vimage:
f − ′ Max-wrt-among le A ⊆ Max-wrt-among (map-relation f le) (f − ′ A)
by (auto simp: Max-wrt-among-def map-relation-def)

lemma Max-wrt-map-relation-vimage:
f − ′ Max-wrt le ⊆ Max-wrt (map-relation f le)
by (auto simp: Max-wrt-altdef map-relation-def)

lemma image-subset-vimage-the-inv-into:
assumes inj-on f A B ⊆ A
shows f ' B ⊆ the-inv-into A f − ′ B
using assms by (auto simp: the-inv-into-f-f)

lemma Max-wrt-among-map-relation-bij-subset:
assumes bij (f :: 'a ⇒ 'b)
shows f ' Max-wrt-among le A ⊆ Max-wrt-among (map-relation (inv f) le) (f ' A)
using assms Max-wrt-among-map-relation-vimage[of inv f A]
by (simp add: bij-imp-bij-inv inv-f-f image-Int[of symmetric])

lemma Max-wrt-among-map-relation-bij:
assumes bij f
shows f ' Max-wrt-among le A = Max-wrt-among (map-relation (inv f) le) (f ' A)
proof (intro equalityI Max-wrt-among-map-relation-bij-subset assms)
interpret R: preorder-on f ' carrier map-relation (inv f) le
  using preorder-on-map[of inv f] assms
by (simp add: bij-imp-bij-inv inv-f-f image-Int[of symmetric])
show Max-wrt-among (map-relation (inv f) le) (f ' A) ⊆ f ' Max-wrt-among le A
  unfolding Max-wrt-among-preorder R.Max-wrt-among-preorder
  using assms bij-is-inj[OF assms]
by (auto simp: map-relation-def image-Int[of symmetric])
qed

lemma Max-wrt-map-relation-bij:

\[ bij f \implies f \cdot \text{Max-wrt} \text{ le} = \text{Max-wrt} (\text{map-relation} (\text{inv} f) \text{ le}) \]

proof –

assume bij: bij f

interpret \( R \): preorder-on \( f \cdot \text{carrier} \text{ map-relation} (\text{inv} f) \text{ le} \)

using preorder-on-map[of inv f] bij

by (simp add: bij-imp-bij-inv bij-vimage-eq-inv-image inv-inv-eq)

from bij show ?thesis

unfolding R.Max-wrt-preorder Max-wrt-preorder

by (auto simp: map-relation-def inv-f-f bij-is-inj)

qed

lemma Max-wrt-among-mono:

\[ \text{le} x y \implies x \in \text{Max-wrt-among} \text{ le} A \implies y \in A \implies y \in \text{Max-wrt-among} \text{ le} A \]

using not-outside by (auto simp: Max-wrt-among-preorder intro: trans)

lemma Max-wrt-mono:

\[ \text{le} x y \implies x \in \text{Max-wrt} \text{ le} \implies y \in \text{Max-wrt} \text{ le} \]

unfolding Max-wrt-def using Max-wrt-among-mono[of x y UNIV] by blast

end

context total-preorder-on

begin

lemma Max-wrt-among-total-preorder:

\[ \text{Max-wrt-among} \text{ le} A = \{ x \in \text{carrier} \cap A. \forall y \in \text{carrier} \cap A. \text{le} y x \} \]

unfolding Max-wrt-among-preorder using total by blast

lemma Max-wrt-total-preorder:

\[ \text{Max-wrt} \text{ le} = \{ x \in \text{carrier}. \forall y \in \text{carrier}. \text{le} y x \} \]

unfolding Max-wrt-preorder using total by blast

lemma decompose-Max:

assumes A: A \subseteq carrier

defines \( M \equiv \text{Max-wrt-among} \text{ le} A \)

shows restrict-relation A le = (\( \lambda x y. x \in A \land y \in M \lor (y \notin M \land \text{restrict-relation} \ (A - M) \text{ le} y) \))

using A by (intro ext) (auto simp: M-def Max-wrt-among-total-preorder

\text{restrict-relation-def Int-absorb1 intro: trans})

end

1.6 Weak rankings

inductive of-weak-ranking :: 'alt set list \Rightarrow 'alt relation where
\[ i \leq j \implies i < \text{length } \text{xs} \implies j < \text{length } \text{xs} \implies x \in \text{xs} ! i \implies y \in \text{xs} ! j \implies x \preceq \text{of-weak-ranking } \text{xs} y \]

**Lemma of-weak-ranking-Nil** [simp]: \( \text{of-weak-ranking } [] = (\lambda - . \text{False}) \)

by (intro ext) (simp add: of-weak-ranking.simps)

**Lemma of-weak-ranking-Nil’** [code]: \( \text{of-weak-ranking } [] x y = \text{False} \)

by simp

**Lemma of-weak-ranking-Cons** [code]:
\[
x \succeq \text{of-weak-ranking } (z \# \text{zs}) y \iff x \in z \land y \in \bigcup (\text{set } (z \# \text{zs})) \lor x \succeq \text{of-weak-ranking } \text{zs} y
\]

(is ?lhs \iff ?rhs)

proof
assume ?lhs
then obtain i j
  where ij: \[ i < \text{length } (z \# \text{zs}) \land j < \text{length } (z \# \text{zs}) \land i \leq j \land x \in (z \# \text{zs}) ! i \land y \in (z \# \text{zs}) ! j \]
by (blast elim: of-weak-ranking.cases)
thus ?rhs by (cases i; cases j) (force intro: of-weak-ranking.intros)+
next
assume ?rhs
thus ?lhs
proof (elim disjE conjE)
assume \( x \in z \land y \in \bigcup (\text{set } (z \# \text{zs})) \)
then obtain j where j: \[ j < \text{length } (z \# \text{zs}) \land y \in (z \# \text{zs}) ! j \]
by (subst (asm) set-conv-nth) auto
with \( x \in z \)
show \( \text{of-weak-ranking } (z \# \text{zs}) y x \)
by (intro of-weak-ranking.intros[of \( 0 \) \( j \)]) auto
next
assume of-weak-ranking \( \text{zs} y x \)
then obtain i j where i j: \[ i < \text{length } \text{zs} \land j < \text{length } \text{zs} \land i \leq j \land x \in \text{zs} ! i \land y \in \text{zs} ! j \]
by (blast elim: of-weak-ranking.cases)
thus of-weak-ranking \( (z \# \text{zs}) y x \)
by (intro of-weak-ranking.intros[of \( \text{Suc } i \) \( \text{Suc } j \)]) auto
qed

**Lemma of-weak-ranking-indifference:**

assumes \( A \in \text{set } \text{xs} \land x \in A \land y \in A \)

shows \( x \succeq \text{of-weak-ranking } \text{xs} y \)

using assms by (induction \text{xs}) (auto simp: of-weak-ranking-Cons)

**Lemma of-weak-ranking-map:**
map-relation \( f \) (of-weak-ranking \( \text{xs} \)) = of-weak-ranking (map ((\(-\)) f) \text{xs})
by (intro ext, induction \text{xs})
(simp-all add: map-relation-def of-weak-ranking-Cons)
lemma of-weak-ranking-permute':
  assumes f permutes (⋃(set xs))
  shows map-relation f (of-weak-ranking xs) = of-weak-ranking (map ((' -') (inv_f)) xs)
proof –
  have map-relation f (of-weak-ranking xs) = of-weak-ranking (map ((' -') f) xs)
    by (rule of-weak-ranking-map)
  also from assms have map ((' -') f) xs = map ((' -') (inv f)) xs
    by (intro map-cong refl) (simp add: bij-vimage-eq-inv-image permutes-bij)
  finally show ?thesis.
qed

lemma of-weak-ranking-permute:
  assumes f permutes (⋃(set xs))
  shows of-weak-ranking (map ((' -') f) xs) = map-relation (inv f) (of-weak-ranking xs)
using of-weak-ranking-permute"[OF permutes-inv[OF assms]]
by (simp add: inv-inv-eq permutes-bij)

definition is-weak-ranking where
is-weak-ranking xs ←→ ({} /∈ set xs) ∧
  (∀ i j. i < length xs ∧ j < length xs ∧ i ≠ j → xs ! i ∩ xs ! j = {})

definition is-finite-weak-ranking where
is-finite-weak-ranking xs ←→ is-weak-ranking xs ∧ (∀ x ∈ set xs. finite x)

definition weak-ranking :: 'alt relation ⇒ 'alt set list where
weak-ranking R = (SOME xs. is-weak-ranking xs ∧ R = of-weak-ranking xs)

lemma is-weak-rankingI [intro?]:
  assumes {} /∈ set xs ∧ i j. i < length xs ⇒ j < length xs ⇒ i ≠ j ⇒ xs ! i ∩ xs ! j = {}
  shows is-weak-ranking xs
using assms by (auto simp add: is-weak-ranking-def)

lemma is-weak-ranking-nonempty: is-weak-ranking xs ⇒ {} /∈ set xs
by (simp add: is-weak-ranking-def)

lemma is-weak-rankingD:
  assumes is-weak-ranking xs i < length xs j < length xs i ≠ j
  shows xs ! i ∩ xs ! j = {}
using assms by (simp add: is-weak-ranking-def)

lemma is-weak-ranking-iff:
  is-weak-ranking xs ←→ distinct xs ∧ disjoint (set xs) ∧ {} /∈ set xs
proof safe
  assume wf: is-weak-ranking xs
  from wf show disjoint (set xs)
    by (auto simp: disjoint-def is-weak-ranking-def set-conv-nth)
show distinct xs

proof (subst distinct-conv-nth, safe)
   fix i j assume ij: i \less length xs j \less length xs i \neq j xs ! i = xs ! j
   then have xs ! i \cap xs ! j = {} by (intro is-weak-rankingD wf)
   with ij have xs ! i = {} by simp
   with ij have {} \in set xs by (auto simp: set-conv-nth)
   moreover from wf ij have {} \nin set xs by (intro is-weak-ranking-nonempty wf)
   ultimately show False by contradiction
   qed
next
   assume A: distinct xs disjoint (set xs) {} \nin set xs
   thus is-weak-ranking xs
      by (intro is-weak-rankingI) (auto simp: disjoint-def distinct-conv-nth)
   qed (simp-all add: is-weak-ranking-nonempty)

lemma is-weak-ranking-rev [simp]: is-weak-ranking (rev xs) \iff is-weak-ranking xs
   by (simp add: is-weak-ranking-iff)

lemma is-weak-ranking-map-inj:
   assumes is-weak-ranking xs inj-on f (\bigcup (set xs))
   shows is-weak-ranking (map ((\') f) xs)
   using assms by (auto simp: is-weak-ranking-iff distinct-map inj-on-image disjoint-image)

lemma of-weak-ranking-rev [simp]:
   of-weak-ranking (rev xs) (x::'a) y \iff of-weak-ranking xs y x
   proof
     have of-weak-ranking (rev xs) y x if of-weak-ranking xs x y for xs and x y :: 'a
       proof
         from that obtain i j where i \less length xs j \less length xs x \in xs ! i y \in xs ! j i
           \geq j
           by (elim of-weak-ranking.cases) simp-all
         thus \thesis
           by (intro of-weak-ranking.intros[of length xs - i - 1 length xs - j - 1] diff-le-mono2)
           (auto simp: diff-le-mono2 rev-nth)
       qed
     from this[of xs y x] this[of rev xs x y] show \thesis by (intro iffI) simp-all
   qed

lemma is-weak-ranking-Nil [simp, code]: is-weak-ranking []
   by (auto simp: is-weak-ranking-def)

lemma is-finite-weak-ranking-Nil [simp, code]: is-finite-weak-ranking []
   by (auto simp: is-finite-weak-ranking-def)

lemma is-weak-ranking-Cons-empty [simp]:


\[\neg \text{is-weak-ranking } (\{\} \# xs) \text{ by (simp add: is-weak-ranking-def)}\]

**Lemma** is-finite-weak-ranking-Cons-empty [simp]:
\[\neg \text{is-finite-weak-ranking } (\{\} \# xs) \text{ by (simp add: is-weak-ranking-def)}\]

**Lemma** is-weak-ranking-singleton [simp]:
\[\text{is-weak-ranking } [x] \iff x \neq \{\} \text{ by (auto simp add: is-weak-ranking-def)}\]

**Lemma** is-finite-weak-ranking-singleton [simp]:
\[\text{is-finite-weak-ranking } [x] \iff x \neq \{\} \land \text{finite } x \text{ by (auto simp add: is-weak-ranking-def)}\]

**Lemma** is-weak-ranking-append:
\[\text{is-weak-ranking } (xs @ ys) \iff \text{is-weak-ranking } xs \land \text{is-weak-ranking } ys \land (\text{set } xs \cap \text{set } ys = \{\} \land \bigcup(\text{set } xs) \cap \bigcup(\text{set } ys) = \{\})\text{ by (simp only: is-weak-ranking-iff)}\]

\[\text{auto dest: disjointD disjoint-unionD1 disjoint-unionD2 intro: disjoint-union)}\]

**Lemma** is-weak-ranking-Cons [code]:
\[\text{is-weak-ranking } (x \# xs) \iff x \neq \{\} \land \text{is-weak-ranking } xs \land x \cap \bigcup(\text{set } xs) = \{\}\text{ by auto}\]

**Lemma** is-finite-weak-ranking-Cons [code]:
\[\text{is-finite-weak-ranking } (x \# xs) \iff x \neq \{\} \land \text{finite } x \land \text{is-finite-weak-ranking } xs \land x \cap \bigcup(\text{set } xs) = \{\}\text{ by (auto simp add: is-finite-weak-ranking-def is-weak-ranking-Cons)}\]

**Primrec** is-weak-ranking-aux where
\[\text{is-weak-ranking-aux } A [] \iff \text{True}\]
\[| \text{is-weak-ranking-aux } A (x#xs) \iff x \neq \{\} \land A \cap x = \{\} \land \text{is-weak-ranking-aux } (A \cup x) \land xs\]

**Lemma** is-weak-ranking-aux:
\[\text{is-weak-ranking-aux } A xs \iff A \cap \bigcup(\text{set } xs) = \{\} \land \text{is-weak-ranking } xs\]
\[\text{by (induction } xs \text{ arbitrary: } A) \text{ (auto simp: is-weak-ranking-Cons)}\]

**Lemma** is-weak-ranking-code [code]:
\[\text{is-weak-ranking } xs \iff \text{is-weak-ranking-aux } \{\} \land xs\]
\[\text{by (subst is-weak-ranking-aux)} \text{ auto}\]

**Lemma** of-weak-ranking-altdef:
\[\text{assumes } \text{is-weak-ranking } xs \in \bigcup(\text{set } xs) \land y \in \bigcup(\text{set } xs)\]
\[\text{shows } \text{of-weak-ranking } xs \land y \iff \text{find-index } ((\in)) \land x \geq \text{find-index } ((\in)) \land y \land xs\]

**Proof** —
from `assms`

have `A`: `find-index (\langle \in \rangle \ x) \ xs < \ length \ xs` `find-index (\langle \in \rangle \ y) \ xs < \ length \ xs`
by `(simp-all add: `find-index-less-size-conv`)

from `this[\ THEN \ nth-find-index]`

have `B`: `x \in \ xs \ ! \ find-index (\langle \in \rangle \ x) \ xs \ y \in \ xs \ ! \ find-index (\langle \in \rangle \ y) \ xs`.

show `?thesis`
proof
assume `of-weak-ranking \ xs \ x \ y`
then obtain `i j` where `ij`: `j \leq i \ i < \ length \ xs` `\ x \in \ xs` `\ y \in \ xs` `\ j < \ length \ xs`
by `(cases rule: `of-weak-ranking.cases`) simp-all`
with `A B` have `i = \ find-index (\langle \in \rangle \ x) \ xs \ j = \ find-index (\langle \in \rangle \ y) \ xs`
using `assms(1)` unfolding `is-weak-ranking-def` by blast+
with `ij` show `\ find-index (\langle \in \rangle \ x) \ xs \geq \ find-index (\langle \in \rangle \ y) \ xs` by simp
next
assume `\ find-index (\langle \in \rangle \ x) \ xs \geq \ find-index (\langle \in \rangle \ y) \ xs`
from `this A(2,1)` `B(2,1)` show `of-weak-ranking \ xs \ x \ y`
by `(rule of-weak-ranking.intros)`
qed

qed

lemma `total-preorder-of-weak-ranking`:
assumes `\ \Union (\ set \ xs) = A`
assumes `is-weak-ranking \ xs`
shows `total-preorder-on A (of-weak-ranking \ xs)`
proof
fix `x \ y` assume `x \leq[of-weak-ranking \ xs] \ y`
with `assms` show `x \in A \ y \in A`
by `(auto elim!: `of-weak-ranking.cases`)`
next
fix `x \ y` assume `x \in A \ y \in A`
with `assms(1)` obtain `i \ j` where `ij`: `i < \ length \ xs` `\ x \in \ xs` `\ y \in \ xs` `\ j < \ length \ xs`
by `(auto simp: `set-conv-nth`)` thus `x \leq[of-weak-ranking \ xs] \ x` by `(auto intro: `of-weak-ranking.intros`)`
next
fix `x \ y` assume `x \in A \ y \in A`
with `assms(1)` obtain `i \ j \ where \ ij`: `i < \ length \ xs` `j < \ length \ xs` `\ x \in \ xs` `\ y \in \ xs` `\ j < \ length \ xs`
by `(auto simp: `set-conv-nth`)` consider `i \leq \ j | \ j < \ i` by force
thus `x \leq[of-weak-ranking \ xs] \ y \lor \ y \leq[of-weak-ranking \ xs] \ x`
by `(cases (\ insert \ ij, (\ blast intro: `of-weak-ranking.intros`)))`
next
fix `x \ y` `z`
assume `A`: `x \leq[of-weak-ranking \ xs] \ y` and `B`: `y \leq[of-weak-ranking \ xs] \ z`
from `A` obtain `i \ j`
where `ij`: `i \geq \ j \ i < \ length \ xs` `j < \ length \ xs` `\ x \in \ xs` `\ y \in \ xs` `\ j < \ length \ xs`
by `(auto elim!: `of-weak-ranking.cases`)` moreover from `B` obtain \ `j \ k`
where \(j'k: j' \geq k j' \lt \text{length} \ xs \ k \lt \text{length} \ xs \ y \in \ xs \ y' \in \ xs \ k \ \in \ k\) by (auto elim!: of-weak-ranking.cases)

moreover from \(ij j'k\ \text{is-weak-ranking}\) \([\text{OF assms} (2), \ \text{of} j j']\)

have \(j = j'\) by blast

ultimately show \(x \preceq [\text{of-weak-ranking} \ xs] z\) by (auto intro: of-weak-ranking.intros[of k \ i])

qed

lemma restrict-relation-of-weak-ranking-Cons:

assumes is-weak-ranking \((A \# A\ s)\)

shows restrict-relation \((\bigcup\ (\text{set} \ As)) \ (\text{of-weak-ranking} \ (A \# A \ s)) = \text{of-weak-ranking} \ A\)

proof —

from \(\text{assms}\) interpret \(R: \text{total-preorder-on} \ \bigcup\ (\text{set} \ As) \ \text{of-weak-ranking} \ As\)

by (intro total-preorder-of-weak-ranking)

(simp-all add: \(\text{is-weak-ranking-Cons}\))

from \(\text{assms}\) show \(?\text{thesis}\) using \(R.\text{not-outside}\)

by (intro ext) (auto simp: restrict-relation-def of-weak-ranking-Cons)

qed

lemmas of-weak-ranking-wf =

total-preorder-of-weak-ranking is-weak-ranking-code insert-commute

lemma total-preorder-on \(\{1,2,3,4::\text{nat}\}\) (of-weak-ranking \(\{1,3\},\{2\},\{4\}\))

by (simp add: of-weak-ranking-wf)

context

fixes \(x::'\prime\text{alt set}\) and \(xs::'\prime\text{alt set list}\)

assumes \(\text{wf}: \text{is-weak-ranking} \ (x\#xs)\)

begin

interpretation \(R: \text{total-preorder-on} \ \bigcup\ (\text{set} \ (x\#xs)) \ \text{of-weak-ranking} \ (x\#xs)\)

by (intro total-preorder-of-weak-ranking) (simp-all add: \(\text{wf}\))

lemma of-weak-ranking-imp-in-set:

assumes \(\text{of-weak-ranking} \ xs \ a \ b\)

shows \(a \in \bigcup\ (\text{set} \ xs) \ b \in \bigcup\ (\text{set} \ xs)\)

using \(\text{assms}\) by (fastforce elim!: of-weak-ranking.cases)+

lemma of-weak-ranking-Cons'::

assumes \(a \in \bigcup\ (\text{set} \ (x\#xs)) \ b \in \bigcup\ (\text{set} \ (x\#xs))\)

shows \(\text{of-weak-ranking} \ (x\#xs) \ a \ b \leftrightarrow b \in x \ \vee \ (a \notin x \ \wedge \ \text{of-weak-ranking} \ xs\)

15
proof
  assume of-weak-ranking \((x \# xs)\) a b
  with wf of-weak-ranking-imp-in-set[of a b]
  show \((b \in x \lor a \notin x \land of-weak-ranking xs a b)\)
  by (auto simp: is-weak-ranking-Cons of-weak-ranking-Cons)
next
  assume \(b \in x \lor a \notin x \land of-weak-ranking xs a b\)
  with assms show of-weak-ranking \((x\#xs)\) a b
  by (fastforce simp: of-weak-ranking-Cons)
qed

lemma Max-wrt-among-of-weak-ranking-Cons1:
  assumes \(x \cap A = \{\}\)
  shows Max-wrt-among \((of-weak-ranking (x\#xs))\) A = Max-wrt-among \((of-weak-ranking xs)\) A
proof
  from wf interpret R': total-preorder-on \(\bigcup\) (set xs) of-weak-ranking xs
  by (intro total-preorder-of-weak-ranking) (simp-all add: is-weak-ranking-Cons)
  from assms show ?thesis
  by (auto simp: R'.Max-wrt-among-total-preorder is-weak-ranking-Cons
                   R'.Max-wrt-among-total-preorder of-weak-ranking-Cons)
qed

lemma Max-wrt-among-of-weak-ranking-Cons2:
  assumes \(x \cap A \neq \{\}\)
  shows Max-wrt-among \((of-weak-ranking (x\#xs))\) A = \(x \cap A\)
proof
  from wf interpret R': total-preorder-on \(\bigcup\) (set xs) of-weak-ranking xs
  by (intro total-preorder-of-weak-ranking) (simp-all add: is-weak-ranking-Cons)
  from assms obtain a where \(a \in x \cap A\) by blast
  with wf R'.not-outside(1)[of a] show ?thesis
  by (auto simp: R'.Max-wrt-among-total-preorder is-weak-ranking-Cons
                   R'.Max-wrt-among-total-preorder of-weak-ranking-Cons)
qed

lemma Max-wrt-among-of-weak-ranking-Cons:
  Max-wrt-among \((of-weak-ranking (x\#xs))\) A = 
  \(\text{if } x \cap A = \{\}\ \text{then Max-wrt-among \((of-weak-ranking xs)\) A else } x \cap A\)
using Max-wrt-among-of-weak-ranking-Cons1 Max-wrt-among-of-weak-ranking-Cons2
by simp

lemma Max-wrt-of-weak-ranking-Cons:
  Max-wrt \((of-weak-ranking (x\#xs))\) = x
using wf by (simp add: is-weak-ranking-Cons Max-wrt-def Max-wrt-among-of-weak-ranking-Cons)
end

lemma Max-wrt-of-weak-ranking:
assumes is-weak-ranking xs
shows Max-wrt (of-weak-ranking xs) = (if xs = [] then {} else hd xs)
proof (cases xs)
  case Nil
  hence of-weak-ranking xs = (λ -. False) by (intro ext) simp-all
  with Nil show ?thesis by (simp add: Max-wrt-def Max-wrt-among-def)
next
  case (Cons x xs')
  with assms show ?thesis by (simp add: Max-wrt-of-weak-ranking-Cons)
qed

locale finite-total-preorder-on
  = total-preorder-on +
  assumes finite-carrier [intro]: finite carrier
begin

lemma finite-total-preorder-on-map:
  assumes finite (f − carrier)
  shows finite-total-preorder-on (f − carrier) (map-relation f le)
proof –
  interpret R': total-preorder-on f − carrier map-relation f le
  using total-preorder-on-map [of f]
  from assms show ?thesis by unfold-locales simp
qed

function weak-ranking-aux :: 'a set ⇒ 'a set list where
  weak-ranking-aux {} = []
| A ≠ {} ⇒ A ⊆ carrier ⇒ weak-ranking-aux A =
  Max-wrt-among le A ≠ weak-ranking-aux (A − Max-wrt-among le A)
| ¬(A ⊆ carrier) ⇒ weak-ranking-aux A = undefined
by blast simp-all
termination proof (relation Wellfounded.measure card)
  fix A
  let ?B = Max-wrt-among le A
  assume A: A ≠ {} A ⊆ carrier
  moreover from A(2) have finite A by (rule finite-subset) blast
  moreover from A have ?B ≠ {} ?B ⊆ A
    by (intro Max-wrt-among-nonempty Max-wrt-among-subset; force)+
  ultimately have card (A − ?B) < card A
    by (intro psubset-card-mono) auto
  thus (A − ?B, A) ∈ measure card by simp
qed simp-all

lemma weak-ranking-aux-Union:
  A ⊆ carrier ⇒ ∪(set (weak-ranking-aux A)) = A
proof (induction A rule: weak-ranking-aux.induct [case-names empty nonempty])
  case (nonempty A)
  with Max-wrt-among-subset[of A] show ?case by auto
qed simp-all
lemma weak-ranking-aux-wf:
A ⊆ carrier ⇒ is-weak-ranking (weak-ranking-aux A)

proof (induction A rule: weak-ranking-aux.induct [case-names empty nonempty])

  case (nonempty A)
  have is-weak-ranking (Max-wrt-among le A # weak-ranking-aux (A − Max-wrt-among le A))
  unfolding is-weak-ranking-Cons
  proof (intro conjI)
      from nonempty.prems nonempty.hyps show Max-wrt-among le A ≠ {} by (intro Max-wrt-among-nonempty) auto
  next
      from nonempty.prems show is-weak-ranking (weak-ranking-aux (A − Max-wrt-among le A))
      by (intro nonempty.IH) blast
  next
      from nonempty.prems nonempty.hyps have Max-wrt-among le A ≠ {}
      by (intro Max-wrt-among-nonempty) auto
      moreover from nonempty.prems
      have \( \bigcup (\text{set} (\text{weak-ranking-aux} (A − \text{Max-wrt-among le A}))) = A − \text{Max-wrt-among le A} \)
      by (intro weak-ranking-aux-Union) auto
      ultimately show Max-wrt-among le A ∩ \( \bigcup (\text{set} (\text{weak-ranking-aux} (A − \text{Max-wrt-among le A}))) \) = {} by blast
  qed

with nonempty.prems nonempty.hyps show ?case by simp
qed simp-all

lemma of-weak-ranking-weak-ranking-aux':
assumes A ⊆ carrier x ∈ A y ∈ A
shows of-weak-ranking (weak-ranking-aux A) x y ←→ restrict-relation A le x y using assms

proof (induction A rule: weak-ranking-aux.induct [case-names empty nonempty])

  case (nonempty A)
  define M where M = Max-wrt-among le A
  from nonempty.prems nonempty.hyps have M: M ⊆ A unfolding M-def
  by (intro Max-wrt-among-subset)
  from nonempty.prems have in-MD: le x y if x ∈ A y ∈ M for x y
      using that unfolding M-def Max-wrt-among-total-preorder
      by (auto simp: Int-absorb1)
  from nonempty.prems have in-MI: x ∈ M if y ∈ M x ∈ A le y x for x y
      using that unfolding M-def Max-wrt-among-total-preorder
      by (auto simp: Int-absorb1 intro: trans)
  from nonempty.prems nonempty.hyps
  have IH: of-weak-ranking (weak-ranking-aux (A − M)) x y = restrict-relation (A − M) le x y if x /∈ M y /∈ M using that unfolding M-def by (intro nonempty.IH) auto
from nonempty.prems

interpret \( R' \): total-preorder-on \( A - M \) of-weak-ranking (weak-ranking-aux \( (A - M) \))

by (intro total-preorder-of-weak-ranking weak-ranking-aux-wf weak-ranking-aux-Union)
auto

from nonempty.prems nonempty.hyps M weak-ranking-aux-Union[of A] R'.not-outside[of x y]

show ?case
by (cases \( x \in M \); cases \( y \in M \))
(auto simp: restrict-relation-def of-weak-ranking-Cons IH M-def [symmetric]
intro: in-MD dest: in-MI)

qed simp-all

lemma of-weak-ranking-weak-ranking-aux:
of-weak-ranking (weak-ranking-aux carrier) = le

proof (intro ext)
fix \( x \ y \)
have is-weak-ranking (weak-ranking-aux carrier) by (rule weak-ranking-aux-wf)
simp

then interpret \( R \): total-preorder-on carrier of-weak-ranking (weak-ranking-aux carrier)

by (intro total-preorder-of-weak-ranking weak-ranking-aux-wf weak-ranking-aux-Union)
(simp-all add: weak-ranking-aux-Union)

show of-weak-ranking (weak-ranking-aux carrier) \( x \ y = le \ x \ y \)

proof (cases \( x \in \text{carrier} \land y \in \text{carrier} \))

case True
thus ?thesis
using of-weak-ranking-weak-ranking-aux[of carrier x y] by simp

next

case False
with \( R \).not-outside have of-weak-ranking (weak-ranking-aux carrier) \( x \ y = False \)

by auto
also from not-outside False have \( \ldots = le \ x \ y \) by auto

finally show ?thesis .

qed

qed

lemma weak-ranking-aux-unique':

assumes \( \bigcup (\text{set } As) \subseteq \text{carrier is-weak-ranking } As \)
of-weak-ranking \( As = \text{restrict-relation } (\bigcup (\text{set } As)) \) le

shows \( As = \text{weak-ranking-aux } (\bigcup (\text{set } As)) \)

using assms

proof (induction \( As \))
case (Cons \( A \) \( As \))

have restrict-relation (\( \bigcup (\text{set } As) \)) (of-weak-ranking \( (A \# As) \)) = of-weak-ranking \( As \)

19
\begin{proof}

\textbf{by (intro restrict-relation-of-weak-ranking-Cons Cons.prems)}

\textbf{also have eq1: of-weak-ranking (A # As) = restrict-relation (\bigcup (set (A # As)))}

\textbf{le by fact}

\textbf{finally have eq: of-weak-ranking As = restrict-relation (\bigcup (set As)) le}

\textbf{by (simp add: Int-absorb2)}

\textbf{with Cons.prems have eq2: weak-ranking-aux (\bigcup (set As)) = As}

\textbf{by (intro sym [OF Cons.IH]) (auto simp: is-weak-ranking-Cons)}

\textbf{from eq1 have}

\textbf{Max-wrt-among le (\bigcup (set (A # As))) = Max-wrt-among (of-weak-ranking (A#As)) (\bigcup (set (A#As)))}

\textbf{by (intro Max-wrt-among-cong) simp-all}

\textbf{also from Cons.prems have … = A}

\textbf{by (subst Max-wrt-among-of-weak-ranking-Cons2)}

\textbf{simp-all add: is-weak-ranking-Cons}

\textbf{finally have Max: Max-wrt-among le (\bigcup (set (A # As))) = A .}

\textbf{moreover from Cons.prems have A \neq \{} by (simp add: is-weak-ranking-Cons)

\textbf{ultimately have weak-ranking-aux (\bigcup (set (A # As))) = A # weak-ranking-aux (A \cup \bigcup (set As) – A)}

\textbf{using Cons.prems by simp}

\textbf{also from Cons.prems have A \cup \bigcup (set As) – A = \bigcup (set As)}

\textbf{by (auto simp: is-weak-ranking-Cons)}

\textbf{also from eq2 have weak-ranking-aux … = As .}

\textbf{finally show ?case ..}

\textbf{qed simp-all}

\textbf{lemma weak-ranking-aux-unique:}

\textbf{assumes is-weak-ranking As of-weak-ranking As = le}

\textbf{shows A = weak-ranking-aux carrier}

\textbf{proof –}

\textbf{interpret R: total-preorder-on \bigcup (set As) of-weak-ranking As}

\textbf{by (intro total-preorder-of-weak-ranking assms) simp-all}

\textbf{from assms have x \in \bigcup (set As) \longleftrightarrow x \in carrier for x}

\textbf{using R.not-outside not-outside R.refl[of x] refl[of x]}

\textbf{by blast}

\textbf{hence eq: \bigcup (set As) = carrier by blast}

\textbf{from assms eq have As = weak-ranking-aux (\bigcup (set As))}

\textbf{with eq show ?thesis by simp}

\textbf{qed}

\textbf{lemma weak-ranking-total-preorder:}

\textbf{is-weak-ranking (weak-ranking le) of-weak-ranking (weak-ranking le) = le}

\textbf{proof –}

\textbf{from weak-ranking-aux-wf [of carrier] of-weak-ranking-weak-ranking-aux}

\textbf{have \exists x. is-weak-ranking x \land le = of-weak-ranking x by auto}

\textbf{hence is-weak-ranking (weak-ranking le) \land le = of-weak-ranking (weak-ranking le)}

\textbf{20}
unfolding weak-ranking-def by (rule someI-ex)
thus is-weak-ranking (weak-ranking le) of-weak-ranking (weak-ranking le) = le
by simp-all
qed

lemma weak-ranking-altdf:
weak-ranking le = weak-ranking-aux carrier
by (intro weak-ranking-aux-unique weak-ranking-total-preorder)

lemma weak-ranking-Union: \( \bigcup \{ \text{set (weak-ranking le)} \} \) = carrier
by (simp add: weak-ranking-altdf weak-ranking-aux-Union)

lemma weak-ranking-unique:
assumes is-weak-ranking As of-weak-ranking As = le
shows As = weak-ranking le
using assms unfolding weak-ranking-altdf by (rule weak-ranking-aux-unique)

lemma weak-ranking-permute:
assumes f permutes carrier
shows weak-ranking (map-relation (inv f) le) = map ((' f)) (weak-ranking le)
proof –
from assms have inv f - ' carrier = carrier
  by (simp add: permutes-vimage permutes-inv)
then interpret R: finite-total-preorder-on inv f - ' carrier map-relation (inv f) le
  by (intro finite-total-preorder-on-map) (simp-all add: finite-carrier)
from assms have is-weak-ranking (map ((' f)) (weak-ranking le))
  by (intro is-weak-ranking-map-inj)
  (simp-all add: weak-ranking-total-preorder permutes-inj-on)
with assms show ?thesis
  by (intro sym[OF R.weak-ranking-unique])
  (simp-all add: of-weak-ranking-permute weak-ranking-Union weak-ranking-total-preorder)
qed

lemma weak-ranking-index-unique:
assumes is-weak-ranking xs i < length xs j < length xs \( \in \) xs \( \in \) xs ! j
shows i = j
using assms unfolding is-weak-ranking-def by auto

lemma weak-ranking-index-unique-':
assumes is-weak-ranking xs i < length xs x \( \in \) xs ! i
shows i = find-index (\( \in \)) x xs
using assms find-index-less-size-cone nth-mem
by (intro weak-ranking-index-unique[OF assms(1,2) - assms(3)])
  nth-find-index[\( \in \)] x xs ! [blasto]

lemma weak-ranking-eqclass1:
assumes A \( \in \) set (weak-ranking le) x \( \in \) A y \( \in \) A
shows le x y

21
proof
  from assms obtain i where weak-ranking le ! i = A i < length (weak-ranking le)
    by (auto simp: set-conv-nth)
with assms have of-weak-ranking (weak-ranking le) x y
    by (intro of-weak-ranking.intros[of i i]) auto
thus ?thesis by (simp add: weak-ranking-total-preorder)
qed

lemma weak-ranking-eqclass2:
  assumes A: A ∈ set (weak-ranking le) x ∈ A and le: le x y le y x
  shows y ∈ A
proof
  from le have wf: is-weak-ranking xs by (simp add: xs-def weak-ranking-total-preorder)
  let ?le' = of-weak-ranking xs
  from le' have le': ?le' x y ?le' y x by (simp-all add: weak-ranking-total-preorder)
  from le' (1) obtain i j
    where ij: j ≤ i i < length xs j < length xs x ∈ xs ! i y ∈ xs ! j
      by (cases rule: of-weak-ranking.cases)
  from le' (2) obtain i' j'
    where ij': j' ≤ i' i' < length xs j' < length xs x ∈ xs ! j' y ∈ xs ! i'
      by (cases rule: of-weak-ranking.cases)
  from ij ij' have eq: i = j' j = i'
    by (intro weak-ranking-index-unique[OF wf]; simp)+
  moreover from A obtain k where k: k < length xs A = xs ! k
    by (auto simp: xs-def set-conv-nth)
  ultimately have of-weak-ranking (weak-ranking le) y x using assms (1)
    by (intro weak-ranking-index-unique[OF wf, of - - x]) auto
  with ij ij' k eq show ?thesis by (auto simp: xs-def)
qed

lemma hd-weak-ranking:
  assumes x ∈ hd (weak-ranking le) y ∈ carrier
  shows le x y
proof
  from weak-ranking-Union assms obtain i
    where i: i < length (weak-ranking le) y ∈ weak-ranking le ! i
      by (auto simp: set-conv-nth)
  moreover from assms(2) weak-ranking-Union have weak-ranking le ≠ [] by auto
  ultimately have of-weak-ranking (weak-ranking le) y x using assms(1)
    by (intro of-weak-ranking.intros[of 0 i]) (auto simp: hd-conv-nth)
  thus ?thesis by (simp add: weak-ranking-total-preorder)
qed

lemma last-weak-ranking:
  assumes x ∈ last (weak-ranking le) y ∈ carrier
The index in weak ranking of a given alternative. An element with index 0 is first-ranked; larger indices correspond to less-preferred alternatives.

**Definition**

\[\text{weak-ranking-index } : 'a \Rightarrow \text{nat}\]

\[\text{weak-ranking-index } x = \text{find-index} (\lambda A. x \in A) \text{(weak-ranking le)}\]

**Lemma**

\[\text{nth-weak-ranking-index:}\]

\[\text{assumes } x \in \text{carrier}\]

\[\text{shows } \text{weak-ranking-index } x < \text{length (weak-ranking le)}\]

\[\text{x \in weak-ranking le ! weak-ranking-index } x\]

**Proof**

- \[\text{from assms weak-ranking-Union show weak-ranking-index } x < \text{length (weak-ranking le)}\]

  \[\text{unfolding weak-ranking-index-def by (auto simp add: find-index-less-size-conv)}\]

  \[\text{thus } x \in \text{weak-ranking le ! weak-ranking-index } x\]

  \[\text{unfolding weak-ranking-index-def by (rule nth-find-index)}\]

**QED**

**Lemma**

\[\text{ranking-index-eqI:}\]

\[\text{assumes } x \in \text{carrier } y \in \text{carrier}\]

\[\text{shows } \text{weak-ranking-index } x \geq \text{weak-ranking-index } y \leftrightarrow \text{le } x y\]

**Proof**

- \[\text{have le } x y \leftrightarrow \text{of-weak-ranking (weak-ranking le) } x y\]

  \[\text{by (simp add: weak-ranking-total-preorder)}\]

  \[\text{also have } \ldots \leftrightarrow \text{weak-ranking-index } x \geq \text{weak-ranking-index } y\]

  **Proof**

  \[\text{assume weak-ranking-index } x \geq \text{weak-ranking-index } y\]

  \[\text{thus of-weak-ranking (weak-ranking le) } x y\]

  \[\text{by (rule of-weak-ranking.intros) (simp-all add: nth-weak-ranking-index assms)}\]

  **Next**

  \[\text{assume of-weak-ranking (weak-ranking le) } x y\]
then obtain $i \ j$ where

\[ i \leq j \ i < \text{length} \ (\text{weak-ranking le}) \ j < \text{length} \ (\text{weak-ranking le}) \]

\[ x \in \text{weak-ranking le} ! \ j y \in \text{weak-ranking le} ! \ i \]

by (elm of-weak-ranking.cases) blast

with ranking-index-eql[of i] ranking-index-eql[of j]

show weak-ranking-index $x \geq \text{weak-ranking-index y}$ by simp

qed

finally show \ ?thesis ..

qed

end

lemma weak-ranking-False [simp]: weak-ranking ($\lambda\ -\ -. \ False$) = []

proof –

interpret finite-total-preorder-on {} $\lambda\ -\ -. \ False$

by unfold-locales simp-all

have [] = weak-ranking ($\lambda\ -\ -. \ False$) by (rule weak-ranking-unique) simp-all

thus \ ?thesis ..

qed

lemmas of-weak-ranking-weak-ranking =

finite-total-preorder-on.weak-ranking-total-preorder(2)

lemma finite-total-preorder-on-iff:

finite-total-preorder-on A R $\leftrightarrow$ total-preorder-on A R $\land$ finite A

by (simp add: finite-total-preorder-on-def finite-total-preorder-on-axioms-def)

lemma finite-total-preorder-of-weak-ranking:

assumes $\bigcup (\text{set} \ \text{xs}) = A$ is-finite-weak-ranking xs

shows finite-total-preorder-on A (of-weak-ranking xs)

proof –

from assms(2) have is-weak-ranking xs by (simp add: is-weak-ranking-def)

from assms(1) and this interpret total-preorder-on A of-weak-ranking xs

by (rule total-preorder-of-weak-ranking)

from assms(2) show \ ?thesis

by unfold-locales (simp add: assms(1)[symmetric] is-finite-weak-ranking-def)

qed

lemma weak-ranking-of-weak-ranking:

assumes is-finite-weak-ranking xs

shows weak-ranking (of-weak-ranking xs) = xs

proof –

from assms interpret finite-total-preorder-on $\bigcup (\text{set} \ \text{xs})$ of-weak-ranking xs

by (intro finite-total-preorder-of-weak-ranking) simp-all

from assms show \ ?thesis

by (intro sym[OF weak-ranking-unique]) (simp-all add: is-finite-weak-ranking-def)

qed
lemma weak-ranking-eqD:
  assumes finite-total-preorder-on alts R1
  assumes finite-total-preorder-on alts R2
  assumes weak-ranking R1 = weak-ranking R2
  shows R1 = R2
proof -
  from assms have of-weak-ranking (weak-ranking R1) = of-weak-ranking (weak-ranking R2) by simp
  with assms(1,2) show ?thesis by (simp add: of-weak-ranking-weak-ranking)
qed

lemma weak-ranking-eq-iff:
  assumes finite-total-preorder-on alts R1
  assumes finite-total-preorder-on alts R2
  shows weak-ranking R1 = weak-ranking R2 ↔ R1 = R2
  using assms weak-ranking-eqD by auto

definition preferred-alts :: 'alt relation ⇒ 'alt set where
  preferred-alts R x = { y. y ⪰ [R] x }

lemma (in preorder-on) preferred-alts-refl [simp]: x ∈ carrier ⇒ x ∈ preferred-alts le x
  by (simp add: preferred-alts-def refl)

lemma (in preorder-on) preferred-alts-altdef:
  preferred-alts le x = { y∈carrier. y ⪰[le] x }
  by (auto simp: preferred-alts-def intro: not-outside)

lemma (in preorder-on) preferred-alts-subset: preferred-alts le x ⊆ carrier
  unfolding preferred-alts-def using not-outside by blast

1.7 Rankings

definition ranking :: 'a relation ⇒ 'a list where
  ranking R = map the-elem (weak-ranking R)

locale finite-linorder-on = linorder-on +
  assumes finite-carrier [intro]: finite carrier
begin

sublocale finite-total-preorder-on carrier le
  by unfold-locales (fact finite-carrier)

lemma singleton-weak-ranking:
  assumes A ∈ set (weak-ranking le)
  shows is-singleton A
proof (rule is-singletonI')
  from assms show A ≠ {}

25
using weak-ranking-total-preorder(1) is-weak-ranking-iff by auto

next
fix x y assume x ∈ A y ∈ A
with assms
have x ≤[of-weak-ranking (weak-ranking le)] y y ≤[of-weak-ranking (weak-ranking le)] x
  by (auto intro!: of-weak-ranking-indifference)
with weak-ranking-total-preorder(2)
  show x = y by (intro antisymmetric) simp-all
qed

lemma weak-ranking-ranking: weak-ranking le = map (λx. {x}) (ranking le)
unfolding ranking-def map-map o-def
proof (rule sym, rule map-idI)
  fix A assume A ∈ set (weak-ranking le)
  hence is-singleton A by (rule singleton-weak-ranking)
  thus {the-elem A} = A by (auto elim: is-singletonE)
qed

end
end

2 Preference Profiles

theory Preference-Profiles
imports
  Main
  Order-Predicates
  HOL−Library. Multiset
  HOL−Library. Disjoint-Sets
begin

The type of preference profiles

type-synonym ('agent, 'alt) pref-profile = 'agent ⇒ 'alt relation

locale preorder-family =
fixes dom :: 'a set and carrier :: 'b set and R :: 'a ⇒ 'b relation
assumes nonempty-dom: dom ≠ {}
assumes in-dom [simp]: i ∈ dom ⇒ preorder-on carrier (R i)
assumes not-in-dom [simp]: i ∉ dom ⇒ ¬R i x y
begin

lemma not-in-dom': i ∉ dom ⇒ R i = (λ- _. False)
  by (simp add: fun-eq-iff)

end
locale pref-profile-wf =
  fixes agents :: 'agent set and alts :: 'alt set and R :: ('agent, 'alt) pref-profile
  assumes nonempty-agents [simp]: agents ≠ {} and nonempty-alts [simp]: alts ≠ {} 
  assumes prefs-wf [simp]: i ∈ agents ⇒ finite-total-preorder-on alts (R i) 
  assumes prefs-undefined [simp]: i ∉ agents ⇒ ¬ R i x y

begin

lemma finite-alts [simp]: finite alts
proof –
  from nonempty-agents obtain i where i ∈ agents by blast
  then interpret finite-total-preorder-on alts R i by simp
  show ?thesis by (rule finite-carrier)
qed

lemma prefs-wf' [simp]:
i ∈ agents ⇒ total-preorder-on alts (R i) i ∈ agents ⇒ preorder-on alts (R i)
using prefs-wf [of i]
by (simp-all add: finite-total-preorder-on-def total-preorder-on-def del: prefs-wf)

lemma not-outside:
assumes x ⪯ [R i] y
shows i ∈ agents x ∈ alts y ∈ alts
proof –
  from assms show i ∈ agents by (cases i ∈ agents) auto
  then interpret preorder-on alts R i by simp
  from assms show x ∈ alts y ∈ alts by (simp-all add: not-outside)
qed

sublocale preorder-family agents alts R 
by (intro preorder-family.intro simp-all)

lemmas prefs-undefined' = not-in-dom'

lemma wf-update:
assumes i ∈ agents total-preorder-on alts Ri'
shows pref-profile-wf agents alts (R(i := Ri'))
proof –
  interpret total-preorder-on alts Ri' by fact
  from finite-alts have finite-total-preorder-on alts Ri' by unfold-locales
  with assms show ?thesis
    by (auto intro!: pref-profile-wf.intro split: if-splits)
qed

lemma wf-permute-agents:
assumes σ permutes agents
shows pref-profile-wf agents alts (R o σ)
unfolding o-def using permutes-in-image[OF assms(1)]
by (intro pref-profile-wf.intro prefs-wf) simp-all
lemma (in –) pref-profile-eqI:
  assumes pref-profile-wf agents alts R1 pref-profile-wf agents alts R2
  assumes \(\forall x. x \in \text{agents} \implies R_1 x = R_2 x\)
  shows \(R_1 = R_2\)
proof
  interpret R1: pref-profile-wf agents alts R1 by fact
  interpret R2: pref-profile-wf agents alts R2 by fact
  fix \(x\) show \(R_1 x = R_2 x\)
  by (cases \(x \in \text{agents}\); intro ext) (simp-all add: assms(3))
qed

end

Permutes a preference profile w.r.t. alternatives in the way described in the paper. This is needed for the definition of neutrality.

definition permute-profile where
  permute-profile \(\sigma\) \(R\) = \((\lambda i\ x\ y. R i (\text{inv } \sigma x) (\text{inv } \sigma y))\)

lemma permute-profile-map-relation:
  permute-profile \(\sigma\) \(R\) = \((\lambda i. \text{map-relation} (\text{inv } \sigma) (R i))\)
by (simp add: permute-profile-def map-relation-def)

lemma permute-profile-compose [simp]:
  permute-profile \(\sigma\) \((R \circ \pi)\) = permute-profile \(\sigma\) \(R \circ \pi\)
by (auto simp: fun-eq-iff permute-profile-def o-def)

lemma permute-profile-id [simp]: permute-profile id \(R\) = \(R\)
by (simp add: permute-profile-def)

lemma permute-profile-o:
  assumes bij f bij g
  shows permute-profile f (permute-profile g \(R\)) = permute-profile \((f \circ g)\) \(R\)
using assms by (simp add: permute-profile-def o-inv-distrib)

lemma (in pref-profile-wf) wf-permute-alts:
  assumes \(\sigma\) permutes alts
  shows pref-profile-wf agents alts (permute-profile \(\sigma\) \(R\))
proof (rule pref-profile-wf.intro)
  fix \(i\) assume \(i \in \text{agents}\)
  with assms interpret \(R:\) finite-total-preorder-on alts \(R\) \(i\) by simp
  from assms have [simp]: \(\text{inv } \sigma x \in \text{alts} \iff x \in \text{alts}\) for \(x\)
  by (simp add: permutes-in-image permutes-inv)
  show finite-total-preorder-on alts \(\text{permute-profile } \sigma\) \(R\) \(i\)
proof
  fix \(x\) \(y\) assume permute-profile \(\sigma\) \(R\) \(i\) \(x\) \(y\)
  thus \(x \in \text{alts} y \in \text{alts}\)
using \textit{R.not-outside[of inv }\sigma x\textit{ inv }\sigma y]\textit{ by }\text{(auto simp: permute-profile-def)}\textit{ next}\textit{ fix }x y z\textit{ assume permute-profile }\sigma R i x y\textit{ permute-profile }\sigma R i y z\textit{ thus permute-profile }\sigma R i x z\textit{ using }\textit{R.trans[of inv }\sigma x\textit{ inv }\sigma y\textit{ inv }\sigma z]\textit{ by }\text{(simp-all add: permute-profile-def)}\textit{ qed (insert }\textit{R.total R.refl R.finite-carrier, simp-all add: permute-profile-def)}\textit{ qed (insert assms, simp-all add: permute-profile-def pref-profile-wf-def)}\textit{ This shows that the above definition is equivalent to that in the paper.}\textbf{ lemma }\textit{permute-profile-iff [simp]:}\textit{ fixes }R:: ('agent, 'alt)\textit{ pref-profile}\textit{ assumes }\sigma\textit{ permutes alts }x\in\textit{alts }y\in\textit{alts}\textit{ defines }R'\equiv\textit{permute-profile }\sigma R\textit{ shows }\sigma x\preceq[R' i] \sigma y\iff x\preceq[R i] y\textit{ using assms by }\text{(simp add: permute-profile-def permutes-inverses)}\textbf{ 2.1 Pareto dominance}\textbf{ definition }\textit{Pareto} :: ('agent \Rightarrow 'alt relation) \Rightarrow 'alt relation \textit{where }x\preceq[Pareto(R)] y\iff (\exists j. x\preceq[R j] x) \land (\forall i. x\preceq[R' i] x\rightarrow x\preceq[R i] y)\textit{ A Pareto loser is an alternative that is Pareto-dominated by some other alternative.}\textbf{ definition }\textit{pareto-losers} :: ('agent, 'alt)\textit{ pref-profile} \Rightarrow 'alt set \textit{where }\textit{pareto-losers }R = \{x. \exists y. y\succ[Pareto(R)] x\}\textbf{ lemma }\textit{pareto-losersI [intro?], simp}: y\succ[Pareto(R)] x\implies x\in\textit{pareto-losers }R\textit{ by }\text{(auto simp: pareto-losers-def)}\textbf{ context preorder-family}\textbf{ begin}\textbf{ lemma }\textit{Pareto-iff}:\textit{ x\preceq[Pareto(R)] y\iff (\forall i\in\textit{dom}. x\preceq[R i] y)}\textbf{ proof}\textit{ assume }A: x\preceq[Pareto(R)] y\textit{ then obtain }j\textit{ where }j: x\preceq[R j] x\textit{ by }\text{(auto simp: Pareto-def)}\textit{ hence }j': j\in\textit{dom}\textit{ by }\text{(cases }j\in\textit{dom})\textit{ auto}\textit{ then interpret preorder-on carrier }R j\textit{ by simp}\textit{ from }j\textit{ have }x\in\textit{carrier}\textit{ by }\text{(auto simp: carrier-eq)}\textit{ with }A\textit{ preorder-on.refl[OF in-dom]}\textit{ show }\text{(\forall i\in\textit{dom}. x\preceq[R i] y)}\textit{ by }\text{(auto simp: Pareto-def)}\textit{ next}\textit{ assume }A: (\forall i\in\textit{dom}. x\preceq[R i] y)\textit{ from nonempty-dom obtain }j\textit{ where }j: j\in\textit{dom}\textit{ by blast}\textit{ then interpret preorder-on carrier }R j\textit{ by simp}\textit{ from }j A\textit{ have }x\preceq[R j] y\textit{ by simp}
hence $x \preceq [R j] x$ using not-outside refl by blast
with $A$ show $x \preceq [\text{Pareto}(R)] y$ by (auto simp: Pareto-def)
qed

lemma Pareto-strict-iff:
  $x \prec [\text{Pareto}(R)] y \iff (\forall i \in \text{dom}. x \preceq [R i] y) \land (\exists i \in \text{dom}. x \prec [R i] y)$
by (auto simp: strongly-preferred-def Pareto-iff nonempty-dom)

lemma Pareto-strictI:
  assumes $\forall i. i \in \text{dom} \implies x \preceq [R i] y$ $i \in \text{dom} x \prec [R i] y$
  shows $x \prec [\text{Pareto}(R)] y$
using assms by (auto simp: Pareto-strict-iff)

lemma Pareto-strictI':
  assumes $\forall i. i \in \text{dom} \implies x \preceq [R i] y$ $i \in \text{dom} \neg x \succeq [R i] y$
  shows $x \prec [\text{Pareto}(R)] y$
proof
  from assms interpret preorder-on carrier $R i$ by simp
  from assms have $x \prec [R i] y$ by (simp add: strongly-preferred-def)
  with assms show $\text{thesis}$ by (auto simp: Pareto-strict-iff)
qed

sublocale Pareto: preorder-on carrier $\text{Pareto}(R)$
proof
  have preorder-on carrier $(R i)$ if $i \in \text{dom}$ for $i$ using that by simp-all
  note $A = \text{preorder-on.not-outside}[OF this(1)] \text{ preorder-on.refl}[OF this(1)]$
  preorder-on.trans[OF this(1)]
  from nonempty-dom obtain $i$ where $i : i \in \text{dom}$ by blast
  show preorder-on carrier $(\text{Pareto } R)$
  proof
    fix $x y$ assume $x \preceq [\text{Pareto}(R)] y$
    with $A(1,2) [OF i]$ show $x \in \text{carrier}$ $y \in \text{carrier}$ by (auto simp: Pareto-iff)
  qed (auto simp: Pareto-iff intro: $A$)
  qed

lemma pareto-loser-in-alts:
  assumes $x \in \text{pareto-losers } R$
  shows $x \in \text{carrier}$
proof
  from assms obtain $y i$ where $i : i \in \text{dom}$ $x \prec [R i] y$
    by (auto simp: pareto-losers-def Pareto-strict-iff)
  then interpret preorder-on carrier $R i$ by simp
  from $x \prec [R i] y$ have $x \preceq [R i] y$ by (simp add: strongly-preferred-def)
  thus $x \in \text{carrier}$ using not-outside by simp
  qed

lemma pareto-losersE:
  assumes $x \in \text{pareto-losers } R$

30
obtains $y$ where $y \in \text{carrier} x \succ [\text{Pareto}(R)] x$

proof

from assms obtain $y$ where $y \succ [\text{Pareto}(R)] x$ unfolding pareto-losers-def
by blast

with $\text{Pareto}$.not-outside[of $x$ $y$] have $y \in \text{carrier}$
by (simp add: strongly-preferred-def)
with $y$ show ?thesis using that by blast
qed

end

2.2 Preferred alternatives

context pref-profile-wf
begin

lemma preferred-alts-subset-alts: preferred-alts $(R \ i) x \subseteq \text{alts} (\text{is} \ ?A)$
and finite-preferred-alts [simp,intro!]: finite (preferred-alts $(R \ i) x$) (\text{is} \ ?B)
proof

have $?A \wedge ?B$
proof (cases $i \in \text{agents}$)

assume $i \in \text{agents}$
then interpret total-preorder-on \text{alts} $i \ R \ i$ by simp

have preferred-alts $(R \ i) x \subseteq \text{alts \ not-outside}$
by (auto simp: preferred-alts-def)

thus ?thesis by (auto dest: finite-subset)
qed (auto simp: preferred-alts-def)

thus $?A \lor ?B$ by blast+
qed

lemma preferred-alts-altdef:

$i \in \text{agents} \implies \text{preferred-alts} \ (R \ i) x = \{ y \in \text{alts}. \ y \succeq [R \ i] x \}$

by (simp add: preorder-on.preferred-alts-altdef)

end

2.3 Favourite alternatives

definition favorites :: (agent, alt) pref-profile $\Rightarrow$ agent $\Rightarrow$ alt set where

favourites $R \ i = \text{Max-wrt} (R \ i)$

definition favorite :: (agent, alt) pref-profile $\Rightarrow$ agent $\Rightarrow$ alt where

favorite $R \ i = \text{the-elem} (\text{favourites} \ R \ i)$

definition has-unique-favorites :: (agent, alt) pref-profile $\Rightarrow$ bool where

has-unique-favorites $R \iff (\forall i. \text{favourites} \ R \ i = \{ \}) \lor \text{is-singleton} (\text{favourites} \ R \ i)$

context pref-profile-wf
begin
lemma favorites-altdef:
  favorites R i = Max-wrt-among (R i) alts

proof (cases i ∈ agents)
  assume i ∈ agents
  then interpret total-preorder-on alts R i by simp
  show ?thesis by (simp add: favorites-def Max-wrt-total-preorder Max-wrt-among-total-preorder)
qed (simp-all add: favorites-def Max-wrt-def Max-wrt-among-def pref-profile-wf-def)

lemma favorites-no-agent [simp]: i /∈ agents ⇒ favorites R i = {}
  by (auto simp: favorites-def Max-wrt-def Max-wrt-among-def)

lemma favorites-altdef':
  favorites R i = {x∈alts. ∀y∈alts. x ⪰[R i] y}

proof (cases i ∈ agents)
  assume i ∈ agents
  then interpret finite-total-preorder-on alts R i by simp
    by (auto simp: favorites-altdef Max-wrt-among-total-preorder)
qed simp-all

lemma favorites-subset-alts: favorites R i ⊆ alts
  by (auto simp: favorites-altdef')

lemma finite-favorites [simp, intro]: finite (favorites R i)
  using favorites-subset-alts finite-alts by (rule finite-subset)

lemma favorites-nonempty: i ∈ agents ⇒ favorites R i ≠ {}
  proof –
    assume i ∈ agents
    then interpret finite-total-preorder-on alts R i by simp
    show ?thesis unfolding favorites-def by (intro Max-wrt-nonempty) simp-all
  qed

lemma favorites-permute:
  assumes i: i ∈ agents and perm: σ permutes alts
  shows favorites (permute-profile σ R) i = σ ‘ favorites R i
  proof –
    from i interpret finite-total-preorder-on alts R i by simp
    from perm show ?thesis unfolding favorites-def
      by (subst Max-wrt-map-relation-bij)
    (simp-all add: permute-profile-def map-relation-def permutes-bij)
  qed

lemma has-unique-favorites-altdef:
  has-unique-favorites R ←→ (∀i∈agents. is-singleton (favorites R i))
proof  safe
  fix  i  assume  has-unique-favorites  R  i  ∈  agents
  thus  is-singleton  (favorites  R  i)  using  favorites-nonempty[of  i]
    by  (auto  simp:  has-unique-favorites-def)
next
  assume  ∀  i  ∈  agents.  is-singleton  (favorites  R  i)
  hence  is-singleton  (favorites  R  i)  ∨  favorites  R  i  =  {}  for  i
    by  (cases  i  ∈  agents)  (simp  add:  favorites-nonempty,  simp  add:  favorites-altdef)
  thus  has-unique-favorites  R  by  (auto  simp:  has-unique-favorites-def)
qed

locale  pref-profile-unique-favorites  =  pref-profile-wf  agents  alts  R
  for  agents  ::  'agent  set
  and  alts  ::  'alt  set
  and  R  +
  assumes  unique-favorites':  has-unique-favorites  R
begin
lemma  unique-favorites:  i  ∈  agents  ⇒  favorites  R  i  =  {favorite  R  i}
  using  unique-favorites'
    by  (auto  simp:  favorite-def  has-unique-favorites-altdef  is-singleton-the-elem)
lemma  favorite-in-alts:  i  ∈  agents  ⇒  favorite  R  i  ∈  alts
  using  favorites-subset-alts[of  i]
    by  (simp  add:  unique-favorites)
end

2.4 Anonymous profiles

definition  anonymous-profile  ::  ('agent,  'alt)  pref-profile  ⇒  ('agent,  'alt)  apref-profile
  where  anonymous-profile-auxdef:
    anonymous-profile  R  =  image-mset  (weak-ranking  o  R)  (mset-set  {i.  R  i  ≠  (λ- -.  False)})

lemma  (in  pref-profile-wf)  agents-eq:
  agents  =  {i.  R  i  ≠  (λ- -.  False)}
proof  safe
  fix  i  assume  i:  i  ∈  agents  and  Ri:  R  i  =  (λ- -.  False)
  from  i  interpret  preorder-on  alts  R  i  by  simp
  from  carrier-eq  Ri  nonempty-alts  show  False  by  simp
next
  fix  i  assume  R  i  ≠  (λ- -.  False)
  thus  i  ∈  agents  using  prefs-undefined[of  i]
    by  (cases  i  ∈  agents)  auto
qed
\begin{itemize}
  \item \textbf{lemma (in pref-profile-wf)} anonymous-profile-def:
    \begin{align*}
      \text{anonymous-profile } R &= \text{image-mset (weak-ranking } \circ R) \text{ (mset-set agents)} \\
      \text{by } &\quad (\text{simp only: agents-eq anonymous-profile-auxdef})
    \end{align*}
  \end{itemize}

\begin{itemize}
  \item \textbf{lemma (in pref-profile-wf)} anonymous-profile-permute:
    \begin{align*}
      \text{assumes } \sigma \text{ permutes alts finite agents} \\
      \text{shows } &\quad \text{anonymous-profile (permute-profile } \sigma R) = \\
                     &\quad \text{image-mset (map } ((^\prime) \sigma)) \text{ (anonymous-profile } R)
    \end{align*}
  \end{itemize}

\begin{itemize}
  \item \textbf{proof --}
    \begin{align*}
      \text{from } &\quad \text{assms(1) interpret } R': \text{pref-profile-wf agents alts permute-profile } \sigma R \\
      \text{by } &\quad (\text{rule wf-permute-alts}) \\
      \text{have } &\quad \text{anonymous-profile (permute-profile } \sigma R) = \\
                     &\quad \{\#\text{weak-ranking (map-relation } (\text{inv } \sigma) (R x)). x \in\# \text{ mset-set agents#}\} \\
      \text{unfolding } &\quad R'.\text{anonymous-profile-def} \\
      \text{by } &\quad (\text{simp add: multiset.map-comp permute-profile-map-relation o-def}) \\
      \text{also from } &\quad \text{assms have } \ldots = \{\#\text{map } ((^\prime) \sigma) \text{ (weak-ranking } (R x)). x \in\# \text{ mset-set agents#}\} \\
                     &\quad \text{by } (\text{intro image-mset-cong}) \\
                     &\quad (\text{simp add: finite-total-preorder-on.weak-ranking-permute[of alts]}), \\
      \text{also have } &\quad \ldots = \{\#\text{map } ((^\prime) \sigma) \text{ (anonymous-profile } R) \\
                     &\quad \text{by } (\text{simp add: anonymous-profile-def multiset.map-comp o-def}) \\
      \text{finally show } &\quad \text{?thesis} .
    \end{align*}
  \end{itemize}

\begin{itemize}
  \item \textbf{qed}
  \end{itemize}

\begin{itemize}
  \item \textbf{lemma (in pref-profile-wf)} anonymous-profile-update:
    \begin{align*}
      \text{assumes } i: \quad i \in \text{agents and fin } &\quad (\text{simp: finite agents and total-preorder-on alts } R_i^\prime) \\
      \text{shows } &\quad \text{anonymous-profile } (R(i := R_i^\prime)) = \\
                     &\quad \text{anonymous-profile } R - \{\#\text{weak-ranking } (R i)\#\} + \{\#\text{weak-ranking } R_i^\prime\#\}
    \end{align*}
  \end{itemize}

\begin{itemize}
  \item \textbf{proof --}
    \begin{align*}
      \text{from } &\quad \text{assms interpret } R': \text{pref-profile-wf agents alts } R(i := R_i^\prime) \\
      \text{by } &\quad (\text{simp add: finite-total-preorder-on-iff wf-update}) \\
      \text{have } &\quad \text{anonymous-profile } (R(i := R_i^\prime)) = \\
                     &\quad \{\#\text{weak-ranking } (\text{if } x = i \text{ then } R_i^\prime \text{ else } R x). x \in\# \text{ mset-set agents#}\} \\
      \text{by } &\quad (\text{simp add: } R'.\text{anonymous-profile-def o-def}) \\
      \text{also have } &\quad \ldots = \{\#\text{if } x = i \text{ then weak-ranking } R_i^\prime \text{ else weak-ranking } (R x). x \\
                     &\quad \text{\in\# mset-set agents#}\} \\
                     &\quad \text{by } (\text{intro image-mset-cong}) \text{ simp-all} \\
      \text{also have } &\quad \ldots = \{\#\text{weak-ranking } R_i^\prime, x \in\# \text{ mset-set } \{x \in \text{agents. } x = i\}\#\} + \\
                     &\quad \{\#\text{weak-ranking } (R x), x \in\# \text{ mset-set } \{x \in \text{agents. } x \neq i\}\#\} \\
      \text{by } &\quad (\text{subst image-mset-If}) ((\text{subst filter-mset-mset-set. simp}+)\text{, rule refl}) \\
      \text{also from } &\quad i \text{ have } \{x \in \text{agents. } x = i\} = \{i\} \text{ by auto} \\
      \text{also have } &\quad \{x \in \text{agents. } x \neq i\} = \text{agents} - \{i\} \text{ by auto} \\
      \text{also have } &\quad \{\#\text{weak-ranking } R_i^\prime, x \in\# \text{ mset-set } \{i\}\#\} = \{\#\text{weak-ranking } R_i^\prime\#\} \\
      \text{by simp} \\
      \text{also from } &\quad i \text{ have mset-set } \{\text{agents} - \{i\}\} = \text{mset-set agents} - \{\#i\#\} \\
      \text{by } &\quad (\text{simp add: mset-set-Diff}) \\
      \text{also from } &\quad i
    \end{align*}
  \end{itemize}
have \{\#\text{weak-ranking} \ (R \cdot \cdot x \in \# \cdot \cdot \cdot \cdot \cdot \cdot \} =
\{\#\text{weak-ranking} \ (R \cdot \cdot x \in \# \cdot \cdot \cdot \cdot \cdot mset \cdot \text{agents}\#\} - \{\#\text{weak-ranking} \ (R \cdot \cdot i \#)\}
by (subst \text{image-mset-Diff}) (\text{simp-all add: in-miset-in-set mset-subset-eq-single})
also have \{\#\text{weak-ranking} \ (R \cdot \cdot i \#)\} + \ldots =
\text{anonymous-profile} \ R - \{\#\text{weak-ranking} \ (R \cdot \cdot i \#)\} + \{\#\text{weak-ranking} \ (R \cdot \cdot i \#)\}
by (simp add: anonymous-profile-def add-ac o-def)
finally show \text{thesis} .
qed

2.5 Preference profiles from lists

definition \text{prefs-from-table} :: ('\agent \times '\alt \set \list) \list \Rightarrow ('\agent, '\alt \set \alt \set) \pref-profile
where
\text{prefs-from-table} \ xss = (\lambda i. \text{case-option} (\lambda- -. False) \text{of-weak-ranking} (\map-of \ xss \ i))

definition \text{prefs-from-table-wf} where
\text{prefs-from-table-wf} \ agents alts xss \longleftrightarrow \agents \neq \{\} \land \alts \neq \{\} \land \text{distinct} (\map \\text{fst} \ xss) \land
set (\map \\text{fst} \ xss) = \agents \land (\forall \xs \in \set (\map \\text{snd} \ xss). \bigcup (\set \xs) = \alts \land
\text{is-finite-weak-ranking} \ xs)

lemma \text{prefs-from-table-wfI}:
assumes \agents \neq \{\} \land \alts \neq \{\} \land \text{distinct} (\map \\text{fst} \ xss)
assumes \set (\map \\text{fst} \ xss) = \agents
assumes \bigwedge \xs. \xs \in \set (\map \\text{snd} \ xss) \implies \bigcup (\set \xs) = \alts
assumes \bigwedge \xs. \xs \in \set (\map \\text{snd} \ xss) \implies \text{is-finite-weak-ranking} \ xs
shows \text{prefs-from-table-wf} \ agents alts xss
using \text{assms unfolding \text{prefs-from-table-wf-def} by auto}

lemma \text{prefs-from-table-wfD}:
assumes \text{prefs-from-table-wf} \ agents alts xss
shows \agents \neq \{\} \land \alts \neq \{\} \land \text{distinct} (\map \\text{fst} \ xss)
and \set (\map \\text{fst} \ xss) = \agents
and \bigwedge \xs. \xs \in \set (\map \\text{snd} \ xss) \implies \bigcup (\set \xs) = \alts
and \bigwedge \xs. \xs \in \set (\map \\text{snd} \ xss) \implies \text{is-finite-weak-ranking} \ xs
using \text{assms unfolding \text{prefs-from-table-wf-def} by auto}

lemma \text{pref-profile-from-tableI}:
\text{prefs-from-table-wf} \ agents alts xss \implies \text{pref-profile-wf} \ agents alts (\text{prefs-from-table} \ xss)

proof (intro \text{pref-profile-wf.intro})
assume \text{wf}: \text{prefs-from-table-wf} \ agents alts xss
fix \ i \ assume \ i: \ i \in \agents
with \text{wf} have \ i \in \set (\map \\text{fst} \ xss) by (simp add: \text{prefs-from-table-wf-def})
then obtain \xs \ where \xs: \xs \in \set (\map \\text{snd} \ xss) \text{prefs-from-table} \ xss \ i =
of-weak-ranking \ xs

35
by (cases map-of xss i)
   (fastforce dest: map-of-SomeD simp: prefs-from-table-def map-of-eq-None-iff)+
with wf show finite-total-preorder-on alts (prefs-from-table xss i)
by (auto simp: prefs-from-table-wf-def intro!: finite-total-preorder-of-weak-ranking)

next
assume wf: prefs-from-table-wf agents alts xss
fix i x y assume i: i ∉ agents
with wf have i ∉ set (map fst xss) by (simp add: prefs-from-table-wf-def)
   hence map-of xss i = None by (simp add: map-of-eq-None-iff)
thus ¬ prefs-from-table xss i x y by (simp add: prefs-from-table-def)
qed (simp-all add: prefs-from-table-wf-def)

lemma prefs-from-table-eqI:
assumes distinct (map fst xs) distinct (map fst ys) set xs = set ys
shows prefs-from-table xs = prefs-from-table ys
proof
  from assms have map-of xs = map-of ys by (subst map-of-inject-set) simp-all
  thus ?thesis by (simp add: prefs-from-table-def)
qed

lemma prefs-from-table-undef:
assumes prefs-from-table-wf agents alts xss i ∉ agents
shows prefs-from-table xss i = (λ- _. False)
proof
  from assms have i ∉ fst ` set xss
      by (simp add: prefs-from-table-wf-def)
  hence map-of xss i = None by (simp add: map-of-eq-None-iff)
  thus ?thesis by (simp add: prefs-from-table-def)
qed

lemma prefs-from-table-map-of:
assumes prefs-from-table-wf agents alts xss i ∈ agents
shows prefs-from-table xss i = of-weak-ranking (the (map-of xss i))
using assms
by (auto simp: prefs-from-table-def map-of-eq-None-iff prefs-from-table-wf-def
      split: option.splits)

lemma prefs-from-table-update:
fixes x xs
assumes i ∈ set (map fst xs)
defines xs' ≡ map (λ(j, y). if j = i then (j, x) else (j, y)) xs
shows (prefs-from-table xs)(i := of-weak-ranking x) =
      prefs-from-table xs'(is ?lhs = ?rhs)
proof
  have xs': set (map fst xs') = set (map fst xs) by (force simp: xs'-def)
  fix k
  consider k = i | k ∉ set (map fst xs) | k ≠ i k ∈ set (map fst xs) by blast
  thus ?lhs k = ?rhs k
proof cases
assume \( k = i \)

moreover from \( k \) have \( y = x \) if \((i, y) \in \text{set } xs'\) for \( y \)

using (that by (auto simp: \( \text{xs'}\) def split: if-splits))

ultimately show \( \text{thesis using } \text{assms(1) } k \) \( \text{xs'} \)

by (auto simp add: \( \text{prefs-from-table-def } \) map-of-eq-None-iff dest!: map-of-\( \text{SomeD} \) split: option.splits)

next

assume \( k: k \notin \text{set } (\text{map fst } xs) \)

with \( \text{assms(1) } \) have \( k': k \neq i \) by auto

with \( k \) \( \text{xs'} \) have \( \text{map-of } k = \text{None } \) \( \text{map-of } ks' \) \( k = \text{None} \)

by (simp-all add: \( \text{map-of-eq-None-iff} \))

thus \( \text{thesis by } (\text{simp add: } \text{prefs-from-table-def } k') \) 

next

assume \( k: k \neq i \) \( k \in \text{set } (\text{map fst } xs) \)

with \( \text{k(1) } \) have \( \text{map-of } ks \) \( k = \text{map-of } ks' \) \( k \) unfolding \( \text{xs'} \)-def

by (induction \( xs \)) fastforce

with \( k \) show \( \text{thesis by } (\text{simp add: } \text{prefs-from-table-def}) \) 

qed 

qed 

lemma \( \text{prefs-from-table-swap} \):

\( x \neq y \Rightarrow \text{prefs-from-table } ((x,x')\#(y,y')\#xs) = \text{prefs-from-table } ((y,y')\#(x,x')\#xs) \)

by (intro ext) (auto simp: \( \text{prefs-from-table-def} \))

lemma \( \text{permute-prefs-from-table} \):

assumes \( \text{\( \sigma \) permutes \( \text{fst ' set xs} \)} \)

shows \( \text{\( \text{prefs-from-table } xs \circ \text{\( \sigma \)} = \text{prefs-from-table } (\text{map } (\lambda (x,y). (\text{inv } \sigma x, y)) ) \text{xs} \)} \)

proof

fix \( i \)

have \( (\text{prefs-from-table } xs \circ \text{\( \sigma \)} ) i = \)

(case \( \text{map-of } xs \) \( (\sigma i) \) of

None \Rightarrow \text{\( \lambda \). False} 

| Some \( x \Rightarrow \text{of-weak-ranking } x \) )

by (simp add: \( \text{prefs-from-table-def } \text{o-def} \))

also have \( \text{map-of } xs \) \( (\sigma i) = \text{map-of } (\text{map } (\lambda (x,y). (\text{inv } \sigma x, y)) ) \text{xs} \) \( i \)

using \( \text{map-of-\( \text{permute} [\text{OF } \text{assms}] \) by } (\text{simp add: } \text{o-def fun-eq-iff}) \)

finally show \( (\text{prefs-from-table } xs \circ \text{\( \sigma \)} ) i = \text{prefs-from-table } (\text{map } (\lambda (x,y). (\text{inv } \sigma x, y)) ) \text{xs} \) \( i \)

by (simp only: \( \text{prefs-from-table-def} \))

qed 

lemma \( \text{permute-profile-from-table} \):

assumes \( \text{af: } \text{prefs-from-table-wf agents alts xss} \)

assumes \( \text{perm: } \text{\( \sigma \) permutes alts} \)

shows \( \text{permute-profile } \text{\( \sigma \)} (\text{prefs-from-table xss}) = \)

\( \text{prefs-from-table } (\text{map } (\lambda (x,y). (x, \text{map } (\lambda (\cdot) \text{\( \sigma \)} y)) ) \text{xss}) \) \( (\text{is } ?f = ?g) \)

proof

fix \( i \)

37
have \( \text{wf}' \) \: \text{prefs-from-table-wf} \:\text{agents} \:\text{alts} \:\left( \lambda (x, y). \left( x, \text{map} \left( (\tau) \sigma \right) y \right) \right) \: \text{xss} \)

proof

\( \text{case (5 x)} \)

then obtain \( y \) where \( y \in \text{set} \: \text{xss} \: \text{x} = \text{map} \left( (\tau) \sigma \right) \left( \text{snd} \: y \right) \)

by (auto simp add: o-def case-prod-unfold)

with \text{assms} show ?case:

by (simp add: image-Union [symmetric] \text{prefs-from-table-wf-def} \:\text{permutes-image} \:\text{o-def} \:\text{case-prod-unfold})

next

\( \text{case (6 x)} \)

then obtain \( y \) where \( y \in \text{set} \: \text{xss} \: \text{x} = \text{map} \left( (\tau) \sigma \right) \left( \text{snd} \: y \right) \)

by (auto simp add: o-def case-prod-unfold)

with \text{assms} show ?case:

by (auto simp: is-finite-weak-ranking-def is-weak-ranking-iff \text{prefs-from-table-wf-def} \:\text{distinct-map} \:\text{permutes-inj-on} \:\text{inj-on-image} \:\text{intro} !: \text{disjoint-image})

qed (insert \text{assms}, simp-all add: image-Union [symmetric] \text{prefs-from-table-wf-def} \:\text{permutes-image} \:\text{o-def} \:\text{case-prod-unfold})

show \( \text{?f} \: i = \text{?g} \: i \)

proof

\( \text{cases i} \in \text{agents} \)

assume \( \text{i} \: \notin \:\text{agents} \)

with \text{assms} \:\text{wf}' \:\text{show} \: \text{?thesis}

by (simp add: permute-profile-def \text{prefs-from-table-undef})

next

\( \text{assume i} \: \in \:\text{agents} \)

define \( \text{xs} \) where \( \text{xs} = \text{the} \:\left( \text{map-of} \: \text{xss} \: \text{i} \right) \)

from \( \text{i} \: \text{wf} \: \text{have} \: \text{xs} \: \text{map-of} \: \text{xss} \: \text{i} = \text{Some} \: \text{xs} \)

by (cases \text{map-of} \: \text{xss} \: \text{i}) (auto simp: \text{prefs-from-table-wf-def} \:\text{xs-def})

have \( \text{xs-in-xss} \: \text{x} \in \text{snd} \:\text{set} \: \text{xss} \)

using \( \text{xs} \) by (force dest!: \text{map-of-SomeD})

with \text{wf} \:\text{have} \: \text{set} \: \text{xss} \:\text{\bigcup} \:\text{set} \: \text{xss} = \text{alts} \)

by (simp add: \text{prefs-from-table-wfD})

from \( \text{i} \) have \( \text{prefs-from-table} \; \left( \lambda (x, y). \left( x, \text{map} \left( (\tau) \sigma \right) y \right) \right) \: \text{xss} \: \text{i} = \text{of-weak-ranking} \; \left( \text{the} \: \left( \text{map-of} \; \left( \lambda (x, y). \left( x, \text{map} \left( (\tau) \sigma \right) y \right) \right) \: \text{xss} \: \text{i} \right) \right) \)

using \( \text{wf}' \) by (intro \text{prefs-from-table-map-of}) simp-all

also have \( \ldots = \text{of-weak-ranking} \; \left( \lambda (\alpha \, \beta) \; \text{of-weak-ranking} \; \text{xss} \; (\text{inv} \; \sigma \; \alpha) \; (\text{inv} \; \sigma \; \beta) \right) \)

by (subst \text{map-of-map}) (simp add: \text{xss})

also have \( \ldots = \left( \lambda a \; b \; \text{of-weak-ranking} \; \text{xss} \; (\text{inv} \; \sigma \; a) \; (\text{inv} \; \sigma \; b) \right) \)

by (intro \text{ext}) (simp add: \text{of-weak-ranking-permute map-relation-def set} \: \text{xss} \:\text{perm})

also have \( \ldots = \text{permute-profile} \; \sigma \; \left( \text{prefs-from-table} \; \text{xss} \right) \: \text{i} \)

by (simp add: \text{prefs-from-table-def} \:\text{xss} \:\text{permute-profile-def})

finally show \( \text{?thesis} \) ..

qed

qed
2.6 Automatic evaluation of preference profiles

**lemma** eval-prefs-from-table [simp]:

\[
prefs-from-table \ [i = (\lambda x. False)]
\]

\[
prefs-from-table ((i, y) \neq xs) \ i = of-weak-ranking \ y \\
i \neq j \Rightarrow prefs-from-table ((j, y) \neq xs) \ i = prefs-from-table \ xs \ i
\]

by (simp-all add: prefs-from-table-def)

**lemma** eval-of-weak-ranking [simp]:

\[
a \notin \bigcup (\set xs) \Rightarrow \neg \ of-weak-ranking \ xs \ a \ b \\
b \in x \Rightarrow a \in \bigcup (\set (x\#xs)) \Rightarrow \ of-weak-ranking \ (x \# xs) \ a \ b \\
b \notin x \Rightarrow \ of-weak-ranking \ (x \# xs) \ a \ b \mapsto \ of-weak-ranking \ xs \ a \ b
\]

by (induction xs) (simp-all add: of-weak-ranking-Cons)

**lemma** prefs-from-table-cong [cong]:

assumes \( \preservation{\text{prefs-from-table \ xs = prefs-from-table \ ys}} \)

shows \( \preservation{\text{prefs-from-table \ (x\#xs) = prefs-from-table \ (x\#ys)}} \)

proof

fix \ i

show \( \preservation{\text{prefs-from-table \ (x \# xs) \ i = prefs-from-table \ (x \# ys) \ i}} \)

using assms by (cases x, cases \( \i = \fst x \)) simp-all

qed

**definition** of-weak-ranking-Collect-ge where

of-weak-ranking-Collect-ge \( x \# x \) \( x = \{ y. \ of-weak-ranking \ xs \ y \ x \} \)

**lemma** eval-of-weak-ranking-Collect-ge:

\( \text{Collect} \ (\text{of-weak-ranking \ xs \ x}) = \text{of-weak-ranking-Collect-ge} \ (\text{rev xs}) \ x \)

by (simp add: of-weak-ranking-Collect-ge-def)

**lemma** of-weak-ranking-Collect-ge-empty [simp]:

\( \text{of-weak-ranking-Collect-ge} \ [x = \{\}] \)

by (simp add: of-weak-ranking-Collect-ge-def)

**lemma** of-weak-ranking-Collect-ge-Cons [simp]:

\( \text{y \in x \Rightarrow \ of-weak-ranking-Collect-ge \ (x\#xs) \ y = \bigcup (\set (x\#xs))} \)

\( y \notin x \Rightarrow \ of-weak-ranking-Collect-ge \ (x\#xs) \ y = \text{of-weak-ranking-Collect-ge} \ xs \ y \)

by (auto simp: of-weak-ranking-Cons of-weak-ranking-Collect-ge-def)

**lemma** of-weak-ranking-Collect-ge-Cons':

\( \text{of-weak-ranking-Collect-ge} \ (x\#xs) = (\lambda y.} \)

(if \( y \in x \) then \( \bigcup (\set (x\#xs)) \) else \( \text{of-weak-ranking-Collect-ge} \ xs \ y))

by (auto simp: of-weak-ranking-Cons of-weak-ranking-Collect-ge-def fun-eq-iff)

**lemma** anonymise-prefs-from-table:

assumes \( \preservation{\text{prefs-from-table-wf \ agents \ alts \ xs}} \)

shows \( \preservation{\text{anonymous-profile} \ (\text{prefs-from-table \ xs}) = \text{mset} \ (\text{map \ snd \ xs})} \)

proof

−

from assms interpret \( \preservation{\text{pref-profile-wf \ agents \ alts \ prefs-from-table \ xs}} \)
by (simp add: pref-profile-from-tableI)
from assms have agents: agents = fst ' set xs
  by (simp add: prefs-from-table-wf-def)

hence [simp]: finite agents by auto

have anonymous-profile (prefs-from-table xs) =
  \{ \#weak-ranking (prefs-from-table xs x). x ∈ \# mset-set agents# \}
  by (simp add: a-def anonymous-profile-def)
also from assms have \ldots = \{ \#the (map-of xs i). i ∈ \# mset-set agents# \}

proof (intro image-mset-cong)
  fix i assume i: i ∈ \# mset-set agents
  from i assms have weak-ranking (prefs-from-table xs i) = weak-ranking (of-weak-ranking (the (map-of xs i)))
    by (simp add: prefs-from-table-map-of)
  also from assms i have \ldots = the (map-of xs i)
    by (intro weak-ranking-of-weak-ranking (auto simp: prefs-from-table-wf-def))
  finally show weak-ranking (prefs-from-table xs i) = the (map-of xs i) .

qed

also from agents have mset-set agents = mset (set (map fst xs)) by simp
also from assms have \ldots = mset (map snd xs)
  by (intro mset-set-set (simp-all add: prefs-from-table-wf-def))
also from assms have \{ \#the (map-of xs i). i ∈ \# mset (map fst xs)# \} = mset (map snd xs)
  by (intro image-mset-map-of (simp-all add: prefs-from-table-wf-def))
finally show \?thesis .

qed

lemma prefs-from-table-agent-permutation:
  assumes wf: prefs-from-table-wf agents alts xs prefs-from-table-wf agents alts ys
  assumes mset-eq: mset (map snd xs) = mset (map snd ys)
  obtains π where π permutes agents prefs-from-table xs o π = prefs-from-table ys

proof ("
  from wf (1) have agents: agents = set (map fst xs)
    by (simp-all add: prefs-from-table-wf-def)
  from wf (2) have agents': agents = set (map fst ys)
    by (simp-all add: prefs-from-table-wf-def)
  from agents agents' wf (1) wf (2) have mset (map fst xs) = mset (map fst ys)
    by (subst set-eq-iff-mset-eq-distinct [symmetric]) (simp-all add: prefs-from-table-wfD)
  hence same-length: length xs = length ys by (auto dest: mset-eq-length simp del: mset-map)
  from \{mset (map fst xs) = mset (map fst ys)\}
    obtain g where g: g permutes \{..<length ys\} permute-list g (map fst ys) = map fst xs
      by (auto elim: mset-eq-permutation simp: same-length simp del: mset-map)
  from mset-eq g
have $mset (\text{map } \text{snd } \text{ys}) = mset (\text{permute-list } g (\text{map } \text{snd } \text{ys})) \text{ by simp}

with $mset$-eq obtain $f$
where $f: f \text{ permutes } \{0..<\text{length } \text{xs}\}$
permute-list $f (\text{permute-list } g (\text{map } \text{snd } \text{ys})) = \text{map } \text{snd } \text{xs}$

by (auto elim; $mset$-eq-permutation simp: same-length simp del: $mset$-map)

from permutes-in-image[OF $f(1)$]
have [simp]: $f x < \text{length } \text{zs} \leftrightarrow x < \text{length } \text{xs}$

$f x < \text{length } \text{ys} \leftrightarrow x < \text{length } \text{ys} \text{ for } x \text{ by (simp-all add: same-length)}$

define $\text{idx}$ $\text{unidx}$ where $\text{idx} = \text{index } (\text{map } \text{fst } \text{xs})$ and $\text{unidx } i = \text{map } \text{fst } \text{xs} \text{ ! } i$

for $i$

from $\text{wf}(1)$ have bij-betw $\text{idx}$ $\text{agents} \{0..<\text{length } \text{xs}\}$ unfolding $\text{idx-def}$

by (intro bij-betw-index) (simp-all add: prefs-from-table-$\text{wfD}$)

hence bij-betw-$\text{idx}$: bij-betw $\text{idx}$ $\text{agents} \{0..<\text{length } \text{xs}\}$ by (simp add: atLeast0LessThan)

have [simp]: $\text{idx } x < \text{length } \text{zs}$ if $x \in \text{agents} \text{ for } x$

using that by (simp add: $\text{idx-def}$ $\text{agents}$)

have [simp]: $\text{unidx } i \in \text{agents}$ if $i < \text{length } \text{xs}$ for $i$

using that by (simp add: $\text{agents} \text{ $\text{unidx-def}$)

have $\text{unidx}$-$\text{idx}$: $\text{unidx} \ (\text{idx } x) = x$ if $x: x \in \text{agents} \text{ for } x$

using $x$ unfolding $\text{idx-def}$ $\text{unidx-def}$ using $\text{nth-index}$-$\text{of } x \text{ $\text{map } \text{fst } \text{xs}$}$

by (simp add: $\text{agents} \text{ set-map } \text{(symmetric)} \text{ $\text{nth-map } \text{[symmetric]} \text{ $\text{del: } \text{set-map}$}$)

have $\text{idx}$-$\text{unidx}$: $\text{idx} \ (\text{unidx } i) = i$ if $i: i < \text{length } \text{xs}$ for $i$

unfolding $\text{idx-def}$ $\text{unidx-def}$ using $\text{wf}(1)$ $\text{index-nth-id}$-$\text{of } \text{map } \text{fst } \text{xs} \text{ i}$ $i$

by (simp add: prefs-from-table-$\text{wfD}(3)$)

define $\pi$ where $\pi x = (if x \in \text{agents} \text{ then } (\text{unidx } o f o \text{ idx} ) \text{ x except } x) \text{ for } x$

define $\pi'$ where $\pi' x = (if x \in \text{agents} \text{ then } (\text{unidx } o \text{ inv } f o \text{ idx} ) \text{ x except } x) \text{ for } x$

have bij-betw $(\text{unidx } o f o \text{ idx})$ $\text{agents} \text{ (is $\pi'$) unfolding $\text{unidx-def}$}

by (rule bij-betw-trans $\text{bij-betw-idx}$ permutes-imp-$\text{bij } f \text{ $g \text{ $bij-betw-nth}$}$+

(insert $\text{wf}(1) \text{ g, simp-all add: } \text{prefs-from-table-wfD}$ same-length)

also have $\text{?P } \leftrightarrow \text{bij-betw } \pi \text{ agents } \text{agents}$

by (intro $\text{bij-betw-conq}$) (simp add: $\pi$-$\text{def}$)

finally have perm: $\pi$ permutes $\text{agents}$

by (intro $\text{bij-imp-permutes}$) (simp-all add: $\pi$-$\text{def}$)

define $h$ where $h = g \circ f$

from $f \text{ g}$ have $h$: $h$ permutes $\{0..<\text{length } \text{ys}\}$ unfolding $h$-$\text{def}$

by (intro permutes-compose) (simp-all add: same-length)

have inv-$\pi$: inv $\pi = \pi'$

proof (rule permutes-invI[OF perm])

fix $x$ assume $x \in \text{agents}$

with $f(1)$ show $\pi' \ (\pi x) = x$

by (simp add: $\pi$-$\text{def}$ $\pi'$-$\text{def}$ $\text{idx-unidx}$ $\text{unidx-def}$ $\text{inv-f}$ $\text{permutes-inj}$)

qed (simp add: $\pi$-$\text{def}$ $\pi'$-$\text{def}$)

with perm have inv-$\pi$: inv $\pi' = \pi$ by (auto simp: inv-inv-eq permutes-bij)

from $\text{wf } h$ have prefs-from-table $\text{ys} = \text{prefs-from-table } (\text{permute-list } h \text{ ys}$)
\begin{verbatim}
by (intro prefs-from-table-eqI)
  (simp-all add: prefs-from-table-wfD permute-list-map [symmetric])
also have permute-list h ys = permute-list h (zip (map fst ys) (map snd ys))
  by (simp add: zip-map-fst-snd)
also from same-length f g
  have permute-list h (zip (map fst ys) (map snd ys)) =
    zip (permute-list f (map fst xs)) (map snd xs)
  by (subst permute-list-zip[OF h]) (simp-all add: h-def permute-list-compose)
also {
  fix i assume i: i < length xs
  from i have permute-list f (map fst xs) ! i = unidx (f i)
    using permutes-in-image[OF f(1)] f(1)
  by (subst permute-list-nth) (simp-all add: same-length unidx-def)
also from i have \ldots = \pi (unidx i) by (simp add: \pi-def idx-unidx)
also from i have \ldots = map \pi (map fst xs) ! i by (simp add: unidx-def)
finally have permute-list f (map fst xs) ! i = map \pi (map fst xs) ! i.
}
hence permute-list f (map fst xs) = map \pi (map fst xs)
  by (intro nth-equalityI) simp-all
also have zip (map \pi (map fst xs)) (map snd xs) =
    map (\lambda x,y. (inv \pi' x, y)) xs
  by (induction xs) (simp-all add: case-prod-unfold inv-\pi')
also from permutes-inv[OF perm] inv-\pi have
  prefs-from-table \ldots = prefs-from-table xs \circ \pi'
  by (intro permute-prefs-from-table [symmetric]) (simp-all add: agents)
finally have
  prefs-from-table xs \circ \pi' = prefs-from-table ys ..
with that[of \pi'] permutes-inv[OF perm] inv-\pi show \?thesis by auto
qed

lemma permute-list-distinct:
  assumes \{f \ldots xs \subseteq \ldots \}
  distinct xs
  shows permute-list f xs = map (\lambda x.x \circ f (index xs x)) xs
  using assms by (intro nth-equalityI) (auto simp: index-nth-id permute-list-def)

lemma image-mset-eq-permutation:
  assumes \{\#f x. x \in A\} = \{\#g x. x \in A\} finite A
  obtains \pi where \pi permutes A \\bigwedge x. x \in A \Rightarrow g (\pi x) = f x
  proof
    from assms(2) obtain xs where xs: A = set xs distinct xs
      using finite-distinct-list by blast
    with assms have mset (map f xs) = mset (map g xs)
      by (simp add: mset-set-set)
    from mset-eq-permutation[OF this] obtain \pi where
      \pi: \pi permutes \{0,..<length xs\} permute-list \pi (map g xs) = map f xs
      by (auto simp: atLeast0LessThan)
    define \pi' where \pi' x = (if x \in A then ((\! xs \circ \pi \circ index xs) x else x) for x
    have bij-betw ((\! xs \circ \pi \circ index xs) A A (is \?P)
      by (rule bij-betw-trans bij-betw-index xs refl permutes-imp-bij \pi bij-betw-nth)+
      (simp-all add: atLeast0LessThan)
  qed
\end{verbatim}
also have ?P ←→ bij-betw π' A A
  by (intro bij-betw-cong) (simp-all add: π'-def)
finally have π' permutes A
  by (rule bij-imp-permutes) (simp-all add: π'-def)
moreover from π xs(1)[symmetric] xs(2) have g(π' x) = f x if x ∈ A for x
  by (simp add: permute-list-map permute-list-distinct
       permutes-image π'-def that atLeast0LessThan)
ultimately show ?thesis by (rule that)
qed

lemma anonymous-profile-agent-permutation:
  assumes eq: anonymous-profile R1 = anonymous-profile R2
  assumes wf: pref-profile-wf agents alts R1 pref-profile-wf agents alts R2
  assumes fin: finite agents
  obtains π where π permutes agents R2 ◦ π = R1
proof –
  interpret R1: pref-profile-wf agents alts R1 by fact
  interpret R2: pref-profile-wf agents alts R2 by fact
from eq have \{#weak-ranking (R1 x). x ∈ # mset-set agents\}
  = \{#weak-ranking (R2 x). x ∈ # mset-set agents\}
  by (simp add: R1.anonymous-profile-def R2.anonymous-profile-def o-def)
from image-mset-eq-permutation[OF this fin] guess π . note π = this
from π have wf': pref-profile-wf agents alts (R2 ◦ π)
  by (intro R2.wf-permute-agents)
then interpret R2': pref-profile-wf agents alts R2 ◦ π .
have R2 ◦ π = R1
proof (intro pref-profile-eqI[OF wf' wf(1)])
  fix x assume x: x ∈ agents
  with π have weak-ranking ((R2 ◦ π) x) = weak-ranking (R1 x) by simp
  with wf' wf(1) x show (R2 ◦ π) x = R1 x
  by (intro weak-ranking-eqD[of alts] R2'.prefs-wf) simp-all
qed
from π(1) and this show ?thesis by (rule that)
qed

end
theory Elections
imports Preference-Profiles
begin
An election consists of a finite set of agents and a finite non-empty set of alternatives.
locale election =
  fixes agents :: 'agent set and alts :: 'alt set
  assumes finite-agents [simp, intro]: finite agents
  assumes finite-alts [simp, intro]: finite alts
  assumes nonempty-agents [simp]: agents ≠ {}
  assumes nonempty-alts [simp]: alts ≠ {}

begin

abbreviation is-pref-profile ≡ pref-profile-wf agents alts

lemma finite-total-preorder-on-iff' [simp]:
  finite-total-preorder-on alts R “→ total-preorder-on alts R
by (simp add: finite-total-preorder-on-iff)

lemma pref-profile-wfI' [intro]:
  (⋀ i. i ∈ agents ⇒ total-preorder-on alts (R i)) ⇒
  (⋀ i. i /∈ agents ⇒ R i = (λ -. False)) ⇒ is-pref-profile R
by (simp add: pref-profile-wf-def)

lemma is-pref-profile-update [simp.intro]:
assumes is-pref-profile R total-preorder-on alts Ri' i ∈ agents
shows is-pref-profile (R(i := Ri'))
using assms by (auto intro: pref-profile-wf.wf-update)

lemma election [simp.intro]: election agents alts
by (rule election-axioms)

context
  fixes R assumes R: total-preorder-on alts R
begin
interpretation R: total-preorder-on alts R by fact

lemma Max-wrt-prefs-finite: finite (Max-wrt R)
unfolding R.Max-wrt-preorder by simp

lemma Max-wrt-prefs-nonempty: Max-wrt R ≠ {}
using R.Max-wrt-nonempty by simp

lemma maximal-imp-preferred:
  x ∈ alts ⇒ Max-wrt R ⊆ preferred-alts R x
using R.total
by (auto simp: R.Max-wrt-total-preorder preferred-alts-def strongly-preferred-def)

end

end

end

3 Auxiliary facts about PMFs

theory Lotteries
  imports Complex-Main HOL−Probability.Probability

begin

The type of lotteries (a probability mass function)

**type-synonym** 
'alt lottery = 'alt pmf

**definition** lotteries-on :: 'a set ⇒ 'a lottery set where
lotteries-on A = {p. set-pmf p ⊆ A}

**lemma** pmf-of-set-lottery:
A ≠ {} ⇒ finite A ⇒ A ⊆ B ⇒ pmf-of-set A ∈ lotteries-on B

**unfolding** lotteries-on-def by auto

**lemma** pmf-of-list-lottery:
pmf-of-list-wf xs ⇒ set (map fst xs) ⊆ A ⇒ pmf-of-list xs ∈ lotteries-on A

**using** set-pmf-of-list[of xs] by (auto simp: lotteries-on-def)

**lemma** return-pmf-in-lotteries-on [simp,intro]:
x ∈ A ⇒ return-pmf x ∈ lotteries-on A

by (simp add: lotteries-on-def)

end

theory Utility-Functions

imports
Complex-Main
HOL—Probability
Probability
Lotteries
Preference-Profiles

begin

3.1 Definition of von Neumann–Morgenstern utility functions

**locale** vnm-utility = finite-total-preorder-on +

**fixes** u :: 'a ⇒ real

**assumes** utility-le-iff: x ∈ carrier ⇒ y ∈ carrier ⇒ u x ≤ u y ⇔ x ≺[le] y

begin

**lemma** utility-le: x ≤[le] y ⇒ u x ≤ u y

using not-outside[of x y] utility-le iff by simp

**lemma** utility-less-iff:

x ∈ carrier ⇒ y ∈ carrier ⇒ u x < u y ⇔ x ≺[le] y

using utility-le iff[of x y] utility-le iff[of y x]

by (auto simp: strongly-preferred-def)

**lemma** utility-less: x ≺[le] y ⇒ u x < u y

using not-outside[of x y] utility-less iff by (simp add: strongly-preferred-def)

The following lemma allows us to compute the expected utility by summing
over all indifference classes, using the fact that alternatives in the same indifference class must have the same utility.

**lemma** expected-utility-weak-ranking:

**assumes** \( p \in \text{lotteries-on carrier} \)

**shows** \( \text{measure-pmf.expectation} p \ u = (\sum A \{\text{weak-ranking le}. \ u (\text{SOME} \ x, x \in A) \} \ast \text{measure-pmf.prob} \ p \ A) \)

**proof** –

from **assms** have \( \text{measure-pmf.expectation} p \ u = (\sum a \in \text{carrier}. \ u a \ast \text{pmf} \ p \ a) \)

by (subst integral-measure-pmf\{\text{OF finite-carrier}\})

(auto simp: lotteries-on-def ac-simps)

also have \( \text{carrier}. \text{carrier} = \bigcup (\text{set} (\text{weak-ranking le})) \) by (simp add: weak-ranking-Union)

also from **carrier** have finite: finite \( A \) if \( A \in \text{set} \ (\text{weak-ranking le}) \) for \( A \)

using that by (blast intro!: finite-subset\{\text{OF - finite-carrier}, of \A\})

hence \( (\sum a \in (\text{set} (\text{weak-ranking le})). \ u a \ast \text{pmf} \ p \ a) = 

(\sum A \{\text{weak-ranking le}. \ \sum a \in A. \ u a \ast \text{pmf} \ p \ a\} \) (is - = sum-list ?xs)

using weak-ranking-total-preorder

by (subst sum.\text{Union-disjoint})

(auto simp: is-weak-ranking-iff disjoint-def sum.\text{distinct-set-conv-list})

also have \( ?xs = \text{map} (\lambda A. \sum a \in A. \ u (\text{SOME} a. a \in A) \ast \text{pmf} \ p \ a) \) (weak-ranking le)

**proof** (intro map-cong HOL.refl sum.cong)

fix \( x \ A \) assume \( x \in A \) and \( A: A \in \text{set} \ (\text{weak-ranking le}) \)

have \( (\text{SOME} \ x, x \in A) \in A \) by (rule someI-ex) (insert \( x \), blast)

from weak-ranking-eclass1\{\text{OF A x this}\} weak-ranking-eclass1\{\text{OF A this x}\} \ x this \( A \)

have \( u x = u (\text{SOME} \ x, x \in A) \)

by (intro antisym; subst utility-le-iff) (auto simp: carrier)

thus \( u x \ast \text{pmf} \ p \ x = u (\text{SOME} \ x, x \in A) \ast \text{pmf} \ p \ x \) by simp

**qed**

also have \( \ldots = \text{map} (\lambda A. \ u (\text{SOME} a. a \in A) \ast \text{measure-pmf.prob} \ p \ A) \)

(weak-ranking le)

using finite by (intro map-cong HOL.refl)

(auto simp: sum.distrib-left measure-measure-pmf-finite)

finally show ?thesis .

**qed**

**lemma** scaled: \( c > 0 \implies \text{vnm-utility carrier le } (\lambda x. c \ast u x) \)

by unfold-locales (insert utility-le-iff , auto)

**lemma** add-right:

**assumes** \( \forall x \ y. \ \le \ x \ y \implies f \ x \le f \ y \)

**shows** \( \text{vnm-utility carrier le } (\lambda x. u x + f x) \)

**proof**

fix \( x \ y \) assume \( xy: x \in \text{carrier} \ y \in \text{carrier} \)

from **assms[of x y] utility-le-iff \{\text{OF xy}\} **assms[of y x] utility-le-iff \{\text{OF xy(2,1)}\}

show \( u x + f x \le u y + f y \) = le \( x \ y \) by auto

**qed**

**lemma** add-left:

46
Given a consistent utility function, any function that assigns equal values to equivalent alternatives can be added to it (scaled with a sufficiently small \( \varepsilon \)), again yielding a consistent utility function.

**Lemma add-epsilon:**

assumes \( A: \forall x y. \text{le } x y \Rightarrow \text{le } y x \Rightarrow f x = f y \)

shows \( \exists \varepsilon > 0. \text{ vnm-utility carrier le } (\lambda x. u x + \varepsilon * f x) \)

proof –

let \( ?A = \{(u y - u x) / (f x - f y) \mid x y. x \prec\text{le} y \land f x > f y\} \)

have \( ?A = (\lambda(x,y). (u y - u x) / (f x - f y)) \cdot \{(x,y) \mid x y. x \prec\text{le} y \land f x > f y\} \)

by auto

also have finite \( \{(x,y) \mid x y. x \prec\text{le} y \land f x > f y\} \)

by (rule finite-subset[of - carrier \times carrier])

(insert not-outside, auto simp: strongly-preferred-def)

hence finite \( ((\lambda(x,y). (u y - u x) / (f x - f y)) \cdot \{(x,y) \mid x y. x \prec\text{le} y \land f x > f y\}) \)

by simp

finally have finite: finite \( ?A \).

define \( \varepsilon \) where \( \varepsilon = \text{Min } (\text{insert } 1 \ ?A) / 2 \)

from finite have \( \text{Min } (\text{insert } 1 \ ?A) > 0 \)

by (auto intro!: divide-pos-pos simp: utility-less)

hence \( \varepsilon: \varepsilon > 0 \) unfolding \( \varepsilon\)-def by simp

have mono: \( u x + \varepsilon * f x < u y + \varepsilon * f y \) if \( xy: x \prec\text{le} y \) for \( x y \)

proof (cases \( f x > f y \))

assume less: \( f x > f y \)

from \( \varepsilon \) have \( \varepsilon < \text{Min } (\text{insert } 1 \ ?A) \) unfolding \( \varepsilon\)-def by linarith

also from less \( xy \) finite have \( \text{Min } (\text{insert } 1 \ ?A) \leq (u y - u x) / (f x - f y) \)

unfolding \( \varepsilon\)-def

by (intro Min-le) auto

finally show \( ?\text{thesis} \) using less by (simp add: field-simps)

next

assume \( \neg f x > f y \)

with utility-less[\( OF \ xy \)] \( \varepsilon \) show \( ?\text{thesis} \)

by (simp add: algebra-simps not-less add-less-le-mono)

qed

have eq: \( u x + \varepsilon * f x = u y + \varepsilon * f y \) if \( xy: x \preceq\text{le} y \preceq\text{le} x \) for \( x y \)

using \( xy[\text{THEN utility-le-}\ A[\text{OF } xy]] \) by simp

have \( \text{vnm-utility carrier le } (\lambda x. u x + \varepsilon * f x) \)

proof

fix \( x y \) assume \( xy: x \in \text{carrier } y \in \text{carrier} \)

show \( (u x + \varepsilon * f x \leq u y + \varepsilon * f y) \leftrightarrow \text{le } x y \)

using (total[\( OF \ xy \)] mono[of \( x y \)] mono[of \( y x \)] eq[of \( x y \)]) by simp

qed

from \( \varepsilon \) this show \( ?\text{thesis} \) by blast

47
lemma \textit{diff-epsilon}:
assumes \(\forall x \ y. \ le x y \implies le y x \implies f x = f y\)
sshows \(\exists \varepsilon > 0. \ vnm\text{-}utility \ carrier \ le (\lambda x. \ u x - \varepsilon \ast f x)\)

proof
from assms have \(\exists \varepsilon > 0. \ vnm\text{-}utility \ carrier \ le (\lambda x. \ u x + \varepsilon \ast -f x)\)
by \(\text{(intro add-epsilon)} \ (\text{subst neg-equal-iff-equal})\)
thus ?thesis by simp

 qed

\begin{document}
\chapter{Stochastic Dominance}

\begin{theory}
\begin{locale}{Stochastic-Dominance}
\import{Complex-Main}
\import{HOL-Probability.Probability}
\import{Lotteries}
\import{Preference-Profiles}
\import{Utility-Functions}

\textbf{4.1 Definition of Stochastic Dominance}

This is the definition of stochastic dominance. It lifts a preference relation on alternatives to the stochastic dominance ordering on lotteries.

\textbf{definition} \(\text{SD} :: \ ('alt \ relation) \Rightarrow ('alt \ lottery \ relation)\)
where
\(p \geq[\text{SD}(R)] q \iff p \in \text{lotteries-on} \{x. \ R x x\} \land q \in \text{lotteries-on} \{x. \ R x x\} \land\)
\((\forall x. \ R x x \implies \text{measure-pmf.prob} \ p \{y. \ y \geq[R] x\} \geq \text{measure-pmf.prob} q \{y. \ y \geq[R] x\})\)

\textbf{lemma} \(\text{SD-empty} \ [\text{simpl}]; \ SD (\lambda-\ -. \ False) = (\lambda-\ -. \ False)\)
by \(\text{(auto simp; fun-eq-iff SD-def lotteries-on-def set-pmf-not-empty})\)

Stochastic dominance over any relation is a preorder.

\textbf{lemma} \(\text{SD-refl} \ [\text{simpl, trans}]; \ p \preceq[\text{SD}(R)] p \iff p \in \text{lotteries-on} \{x. \ R x x\}\)
by \(\text{(simp add: SD-def})\)

\textbf{lemma} \(\text{SD-trans} \ [\text{simpl, trans}]; \ p \preceq[\text{SD}(R)] q \implies q \preceq[\text{SD}(R)] r \implies p \preceq[\text{SD}(R)] r\)
\(\text{unfolding SD-def by (auto intro: order.trans)}\)

\textbf{lemma} \(\text{SD-is-preorder}; \ \text{preorder-on} (\text{lotteries-on} \{x. \ R x x\}) (\text{SD} R)\)
by \(\text{unfold-locale (auto simp; SD-def intro: order.trans})\)

\end{locale}
\end{theory}
\end{document}
lemma SD-preorder:
    \[ p \succeq [\text{SD}(\leq)] q \iff p \in \text{lotteries-on carrier} \land q \in \text{lotteries-on carrier} \land \\
    (\forall x \in \text{carrier}. \text{measure-pmf.prob } p \text{ (preferred-alts le } x) \geq \\
    \text{measure-pmf.prob } q \text{ (preferred-alts le } x)) \]
by (simp add: SD-def preferred-alts-def carrier-eq)

lemma SD-preorderI [intro?]:
    assumes \[ p \in \text{lotteries-on carrier} q \in \text{lotteries-on carrier} \]
    assumes \[ \forall x. x \in \text{carrier} \implies \text{measure-pmf.prob } p \text{ (preferred-alts le } x) \geq \text{measure-pmf.prob } q \]
shows \[ p \succeq [\text{SD}(\leq)] q \]
using assms by (simp add: SD-preorder)

lemma SD-preorderD:
    assumes \[ p \succeq [\text{SD}(\leq)] q \]
shows \[ p \in \text{lotteries-on carrier} q \in \text{lotteries-on carrier} \]
and \[ \forall x. x \in \text{carrier} \implies \text{measure-pmf.prob } p \text{ (preferred-alts le } x) \geq \text{measure-pmf.prob } q \]
using assms unfolding SD-preorder by simp-all

lemma SD-refl' [simp]: \[ p \preceq [\text{SD}(\leq)] p \iff p \in \text{lotteries-on carrier} \]
by (simp add: SD-def carrier-eq)

lemma SD-is-preorder': preorder-on (lotteries-on carrier) (SD(\leq))
using SD-is-preorder[of le] by (simp add: carrier-eq)

lemma SD-singleton-left:
    assumes \[ x \in \text{carrier} q \in \text{lotteries-on carrier} \]
shows \[ \text{return-pmf } x \preceq [\text{SD}(\leq)] q \iff (\forall y \in \text{set-pmf } q. x \preceq [\text{le}] y) \]
proof
assume SD: \[ \text{return-pmf } x \preceq [\text{SD}(\leq)] q \]
from assms SD-preorderD[OF SD, of x]
have \[ \text{measure-pmf.prob } (\text{return-pmf } x) \text{ (preferred-alts le } x) \leq \text{measure-pmf.prob } q \text{ (preferred-alts le } x) \]
by simp
also from assms have \[ \text{measure-pmf.prob } (\text{return-pmf } x) \text{ (preferred-alts le } x) = 1 \]
by (simp add: indicator-def)
finally have \[ AE \ y \text{ in } q. y \succeq [\text{le}] x \]
by (simp add: measure-pmf.measure-ge-1-iff measure-pmf.prob-eq-1 preferred-alts-def)
thus \[ \forall y \in \text{set-pmf } q. y \succeq [\text{le}] x \]
by (simp add: AE-measure-pmf-iff)
next
assume A: \[ \forall y \in \text{set-pmf } q. x \succeq [\text{le}] y \]
show \[ \text{return-pmf } x \preceq [\text{SD}(\leq)] q \]
proof (rule SD-preorderI)
  fix y assume y: y ∈ carrier
  show measure-pmf.prob (return-pmf x) (preferred-alts le y)
    ≤ measure-pmf.prob q (preferred-alts le y)
  proof (cases y ≤[le] x)
    case True
    from True A have measure-pmf.prob q (preferred-alts le y) = 1
    by (auto simp: AE-measure-pmf-iff measure-pmf.prob-eq-1 preferred-alts-def
     intro: trans)
    thus thesis by simp
  qed
qed (simp-all add: preferred-alts-def indicator-def measure-nonneg)

lemma SD-singleton-right:
  assumes x: x ∈ carrier and q: q ∈ lotteries-on carrier
  shows q ≤[SD(le)] return-pmf x ←→ (∀ y∈set-pmf q. y ≤[le] x)
proof safe
  fix y assume SD: q ≤[SD(le)] return-pmf x and y: y ∈ set-pmf q
  from y assms have [simp]: y ∈ carrier by (auto simp: lotteries-on-def)
  from y have 0 < measure-pmf.prob q (preferred-alts le y)
    by (rule measure-pmf-posI) simp-all
  also have ... ≤ measure-pmf.prob (return-pmf x) (preferred-alts le y)
    by (rule SD-preorderD(3)[OF SD]) simp-all
  finally show y ≤[le] x
    by (auto simp: indicator-def preferred-alts-def split: if-splits)
next
  assume A: ∀ y∈set-pmf q. y ≤[le] x
  show q ≤[SD(le)] return-pmf x
  proof (rule SD-preorderI)
    fix y assume y: y ∈ carrier
    show measure-pmf.prob q (preferred-alts le y) ≤
      measure-pmf.prob (return-pmf x) (preferred-alts le y)
    proof (cases y ≤[le] x)
      case False
      with A show thesis
        by (auto simp: preferred-alts-def indicator-def measure-le-0-iff
          measure-pmf.prob-eq-0 AE-measure-pmf-iff intro: trans)
    qed
  qed (insert assms, simp-all add: lotteries-on-def)
qed

lemma SD-strict-singleton-left:
  assumes x: x ∈ carrier q: q ∈ lotteries-on carrier
  shows return-pmf x \{SD(le)\} q ←→ (∀ y∈set-pmf q. x ≤[le] y) ∧ (∃ y∈set-pmf q. (x ≤[le] y))
using assms by (auto simp add: strongly-preferred-def SD-singleton-left SD-singleton-right)
lemma SD-strict-singleton-right:
  assumes \( x \in \text{carrier} \) \( q \in \text{lotteries-on carrier} \)
  shows \( q \prec [SD(le)] \) return-pmf \( x \leftrightarrow (\forall y \in \text{set-pmf } q. \ y \preceq [le] x) \land (\exists y \in \text{set-pmf } q. \ (y \prec [le] x)) \)
  using asms by (auto simp add: strongly-preferred-def SD-singleton-left SD-singleton-right)

lemma SD-singleton [simp]:
  \( x \in \text{carrier} \Rightarrow y \in \text{carrier} \Rightarrow \) return-pmf \( x \preceq [SD(le)] \) return-pmf \( y \leftrightarrow x \preceq [le] y \)
  by (subst SD-singleton-left) (simp-all add: lotteries-on-def)

lemma SD-strict-singleton [simp]:
  \( x \in \text{carrier} \Rightarrow y \in \text{carrier} \Rightarrow \) return-pmf \( x \prec [SD(le)] \) return-pmf \( y \leftrightarrow x \prec [le] y \)
  by (simp add: strongly-preferred-def)

end

context pref-profile-wf
begin

context
fixes i assumes i: \( i \in \text{agents} \)
begin

interpretation Ri: preorder-on alts R i by (simp add: i)

lemmas SD-singleton-left = Ri.SD-singleton-left
lemmas SD-singleton-right = Ri.SD-singleton-right
lemmas SD-strict-singleton-left = Ri.SD-strict-singleton-left
lemmas SD-strict-singleton-right = Ri.SD-strict-singleton-right
lemmas SD-singleton = Ri.SD-singleton
lemmas SD-strict-singleton = Ri.SD-strict-singleton

end

end

lemmas (in pref-profile-wf) [simp] = SD-singleton SD-strict-singleton

4.2 Stochastic Dominance for preference profiles

context pref-profile-wf
begin

lemma SD-pref-profile:
  assumes \( i \in \text{agents} \)
  shows \( p \preceq [SD(R i)] q \leftrightarrow p \in \text{lotteries-on alts} \land q \in \text{lotteries-on alts} \land \)
  \( (\forall x \in \text{alts. measure-pmf.prob } p (\text{preferred-alts } (R i) x) \geq \text{measure-pmf.prob } q (\text{preferred-alts } (R i) x)) \)

51
proof

from assms interpret total-preorder-on alts R i by simp
have preferred-alts (R i) x = {y. y ≽ [R i] x} for x using not-outside
  by (auto simp: preferred-alts-def)
thus ?thesis by (simp add: SD-preorder preferred-alts-def)
qed

lemma SD-pref-profileI [intro]:
  assumes i ∈ agents p ∈ lotteries-on alts q ∈ lotteries-on alts
  assumes ⋀ x. x ∈ alts =⇒ measure-pmf.prob p (preferred-alts (R i) x) ≥
                            measure-pmf.prob q (preferred-alts (R i) x)
  shows p ≽ [SD(R i)] q
  using assms by (simp add: SD-pref-profile)

lemma SD-pref-profileD:
  assumes i ∈ agents p ≽ [SD(R i)] q
  shows p ∈ lotteries-on alts q ∈ lotteries-on alts
  and ⋀ x. x ∈ alts =⇒ measure-pmf.prob p (preferred-alts (R i) x) ≥
                           measure-pmf.prob q (preferred-alts (R i) x)
  using assms by (simp-all add: SD-pref-profile)

end

4.3 SD efficient lotteries

definition SD-efficient :: ('agent, 'alt) pref-profile ⇒ 'alt lottery ⇒ bool where
  SD-efficient-auxdef:
    SD-efficient R p ≡ ¬(∃ q ∈ lotteries-on alts. q ≽ [Pareto (SD ◦ R)] p)

custom context pref-profile-wf

sublocale SD: preorder-family agents lotteries-on alts SD ◦ R unfolding o-def
  by (intro preorder-family.intro SD-is-preorder)
    (simp-all add: preorder-on SD-is-preorder' prefs-undefined')

A lottery is considered SD-efficient if there is no other lottery such that all
agents weakly prefer the other lottery (w.r.t. stochastic dominance) and at
least one agent strongly prefers the other lottery.

lemma SD-efficient-def:
  SD-efficient R p ≡ ¬(∃ q ∈ lotteries-on alts. q ≽ [Pareto (SD ◦ R)] p)

proof
  have SD-efficient R p ≡ ¬(∃ q ∈ lotteries-on alts. q ≽ [Pareto (SD ◦ R)] p)
    unfolding SD-efficient-auxdef ..
  also from nonempty-agents obtain i where i: i ∈ agents by blast
with preorder-on.refl[ of alts R i]
have \{ x, \exists i. R i x x \} = alts by (auto intro!: exI[of - i] not-outside)
finally show ?thesis .
qed

lemma SD-efficient-def':
SD-efficient R p \iff
\neg(\exists q \in \text{lotteries-on alts}. (\forall i \in \text{agents}. q \succeq [SD(R i)] p) \land (\exists i \in \text{agents}. q > [SD(R i)] p))
unfolding SD-efficient-def SD.Pareto-iff strongly-preferred-def [abs-def] by auto

lemma SD-inefficientI:
assumes q \in \text{lotteries-on alts} \land i \in \text{agents} \implies q \succeq [SD(R i)] p
i \in \text{agents} \implies q > [SD(R i)] p
shows \neg SD-efficient R p
using assms unfolding SD-efficient-def' by blast

lemma SD-inefficientI':
assumes q \in \text{lotteries-on alts} \land i \in \text{agents} \implies q \succeq [SD(R i)] p
\exists i \in \text{agents}. q > [SD(R i)] p
shows \neg SD-efficient R p
using assms unfolding SD-efficient-def' by blast

lemma SD-inefficientE:
assumes \neg SD-efficient R p
obtains q i where
q \in \text{lotteries-on alts} \land i \in \text{agents} \implies q \succeq [SD(R i)] p
i \in \text{agents} \implies q > [SD(R i)] p
using assms unfolding SD-efficient-def' by blast

lemma SD-efficientD:
assumes SD-efficient R p q \in \text{lotteries-on alts}
and \land i \in \text{agents} \implies q \succeq [SD(R i)] p \exists i \in \text{agents}. \neg(q \preceq [SD(R i)] p)
shows False
using assms unfolding SD-efficient-def' strongly-preferred-def by blast

lemma SD-efficient-singleton-iff:
assumes [simp]: x \in \text{alts}
shows SD-efficient R (return-pmf x) \iff x \notin \text{pareto-losers R}
proof -
{ assume x: x \in \text{pareto-losers R}
from pareto-losersE[OF x] guess y. note y = this
from y have \neg SD-efficient R (return-pmf x)
by (intro SD-inefficientI'[of return-pmf y]) (force simp: Pareto-strict-iff)+
} moreover {
assume \neg SD-efficient R (return-pmf x)
from SD-inefficientE[OF this] guess q i. note q = this

53
moreover from \( q \) obtain \( y \) where \( y \in \text{set-pmf } q \) \( y \succ [R \ i] \ x \)

by (auto simp: SD-strict-singleton-left)

ultimately have \( y \succ [\text{Pareto}(R)] \ x \)

by (fastforce simp: Pareto-strict-iff SD-singleton-left)

hence \( x \in \text{pareto-losers } R \) by simp

} ultimately show \(?thesis\) by blast

qed

end

4.4 Equivalence proof

We now show that a lottery is preferred w.r.t. Stochastic Dominance iff it yields more expected utility for all compatible utility functions.

context finite-total-preorder-on begin

abbreviation is-vnm-utility ≡ vnm-utility carrier le

lemma utility-weak-ranking-index:

is-vnm-utility \((\lambda x. \text{real} (\text{length} (\text{weak-ranking } le) - \text{weak-ranking-index } x))\)

proof

fix \( x \ y \) assume \( xy \):

\( x \in \text{carrier} \ y \in \text{carrier} \)

with this[THEN nth-weak-ranking-index(1)]

this[THEN nth-weak-ranking-index(2)]

show \((\text{real} (\text{length} (\text{weak-ranking } le) - \text{weak-ranking-index } x)) \leq \text{real} (\text{length} (\text{weak-ranking } le) - \text{weak-ranking-index } y)) \iff le x y\)

by (simp add: le-diff-iff′)

qed

lemma SD-iff-expected-utilities-le:

assumes \( p \in \text{lotteries-on carrier} \ q \in \text{lotteries-on carrier} \)

shows \( p \preceq [\text{SD}(le)] q \iff (\forall u. \text{is-vnm-utility } u \longrightarrow \text{measure-pmf.expectation } p \ u \leq \text{measure-pmf.expectation } q \ u) \)

proof

fix \( u \) assume SD: \( p \preceq [\text{SD}(le)] q \) and is-utility: \( \text{is-vnm-utility } u \)

from is-utility interpret vnm-utility carrier le u .

define \( xs \) where \( xs = \text{weak-ranking } le \)

have le: \( \text{is-weak-ranking } xs \ le = \text{af-weak-ranking } xs \)

by (simp-all add: xs-def weak-ranking-total-preorder)

let \( \text{?pref } = \lambda x. \text{measure-pmf.prob } p \{ y. \ x \preceq [le] \ y \} \) and

?pref′ = \( \lambda x. \text{measure-pmf.prob } p \{ y. \ x \prec [le] \ y \} \)

define \( f \) where \( f i = (\text{SOME } x. \ x \in xs \ ! i) \) for \( i \)

have xs-wf: \( \text{is-weak-ranking } xs \)

by (simp add: xs-def weak-ranking-total-preorder)

hence \( f : i \in xs \ ! i \) if \( i < \text{length } xs \) for \( i \)
using that unfolding f-def is-weak-ranking-def
by (intro someI-ex[\( \lambda x. x \in xs ! i \)]) (auto simp: set-cone-nth)
have f': \( i \in \text{carrier} \) if \( i < \text{length} \, xs \) for \( i \)
  using that f weak-ranking-Union unfolding xs-def by (auto simp: set-cone-nth)
define n where \( n = \text{length} \, xs - 1 \)
from assms weak-ranking-Union have carrier-nonempty: carrier \( \neq \{ \} \) and \( xs \neq [] \)
  by (auto simp: xs-def lotteries-on-def set-pmf-not-empty)
hence \( \text{length} \, xs = \text{Suc} \, n \) and \( xs\)-nonempty: \( xs \neq [] \) by (auto simp add: n-def)
have SD': \( \text{?pref} \, p \, (f \, i) \leq \text{?pref} \, q \, (f \, i) \) if \( i < \text{length} \, xs \) for \( i \)
  using f-le: \( \leq \, f \, (i \, j) \longleftrightarrow i \leq j \) if \( i < \text{length} \, xs \) and \( j < \text{length} \, xs \) for \( i \, j \)
  using that weak-ranking-index-unique[of xs-wf that(1) - f]
  weak-ranking-index-unique[of xs-wf that(2) - f]
  by (auto simp add: le intro: f elim: of-weak-ranking.cases intro!: f-weak-ranking.intros)

have measure-pmf.expectation p u =
(\( \sum_{i<n} \, (u \cdot (f \, i) - u \cdot (\text{Suc} \, i))) * \text{?pref} \, p \, (f \, i) + u \cdot (f \, n) \))
if p: \( p \in \text{lotteries-on \, carrier} \) for p
proof
  from p have measure-pmf.expectation p u =
(\( \sum_{i<\text{length} \, xs} \, u \cdot (f \, i) * \text{measure-pmf.prob} \, p \, (xs \cdot i)) \))
  by (simp add: f-def expected-utility-weak-ranking xs-def sum-list-sum-nth
   atLeast0LessThan)
  also have \( = \sum_{i<\text{length} \, xs} \, u \cdot (f \, i) * (\text{?pref} \, p \, (f \, i) - \text{?pref'} \, p \, (f \, i))) \))
proof (intro sum.cong HOL.refl)
fix i assume i: \( i \in \{..<\text{length} \, xs \}\)
have \( \text{?pref} \, p \, (f \, i) - \text{?pref'} \, p \, (f \, i) = \)
  measure-pmf.prob p \( \{(y. \, f \, i \prec \leq \, y) - \{y. \, f \, i \prec (\leq \, y) \}\} \)
  by (subst measure-pmf.finite-measure-Diff [symmetric])
  (auto simp: strongly-preferred-def)
  also have \( \{y. \, f \, i \prec \leq \, y\} - \{y. \, f \, i \prec (\leq \, y)\} = \)
  \( \{y. \, f \, i \prec \leq \, y \land y \prec \leq \, f \, i\} \) (is - = \( ?A \))
  by (auto simp: strongly-preferred-def)
also have \( = \sum_{xs \cdot ! i} \))
proof safe
fix x assume le: \( \leq \, (f \, i) \) x le x (f i)
from i f show x \( \in xs \cdot ! i \)
  by (intro weak-ranking-eqclass2[OF - - le]) (auto simp: xs-def)
next
fix x assume x \( \in xs \cdot ! i \)
from weak-ranking-eqclass1[OF - this f] weak-ranking-eqclass1[OF - f this] i
  show le x (f i) le (f i) x by (simp-all add: xs-def)
qed
finally show \( u \cdot (f \, i) * \text{measure-pmf.prob} \, p \, (xs \cdot i) = \)
  \( u \cdot (f \, i) * (\text{?pref} \, p \, (f \, i) - \text{?pref'} \, p \, (f \, i)) \) by simp
qed
also have \( \sum_{i<\text{length} \, xs} \, u \cdot (f \, i) * (\text{?pref} \, p \, (f \, i)) - \)
(\( \sum_{i<\text{length} \, xs} \, u \cdot (f \, i) * (\text{?pref'} \, p \, (f \, i)) \))
by (simp add: sum-subtractf ring-distrib) also have \((\sum i<\text{length }xs. u (f i) \ast \text{?pref }p (f i)) = \sum i<\text{n}. u (f i) \ast \text{?pref }p (f i)) + u (f n) \ast \text{?pref }p (f n)\) (is - = ?sum)

by (simp add: n)

also have \((\sum i<\text{length }xs. u (f i) \ast \text{?pref}\text{' }p (f i)) = \sum i<\text{n}. u (f (\text{Suc }i)) \ast \text{?pref}\text{' }p (f (\text{Suc }i)) + u (f 0) \ast \text{?pref}\text{' }p (f 0)\)

unfolding n sum.lessthan-Suc-shift by simp

also have \((\sum i<\text{n}. u (f (\text{Suc }i)) \ast \text{?pref}\text{' }p (f (\text{Suc }i))) = \sum i<\text{n}. u (f (\text{Suc }i)) \ast \text{?pref }p (f i)\)

proof (intro sum.cong HOL.refl)

fix \text{i} \text{ assume i : i \in } \{..<\text{n}\}

have \(f (\text{Suc }i) <\le \text{ y} \iff f i \le\le \text{ y for } y\)

proof (cases \text{ y} \in \text{carrier})

assume \text{ y} \in \text{carrier}

with weak-ranking-Union obtain \text{j} where \(j <\text{ length }xs \text{ y} \in \text{xs} ! j\)

by (auto simp: set-conv-nth xs-def)

with weak-ranking-eqclass1[OF - \text{f }j(2)] weak-ranking-eqclass1[OF - \text{f j(2)} \text{ f}]

have \(\text{iff: le } y \text{ z }\iff le (f j) \text{ z }le \text{ z } y \iff le (f (j)) \text{ for } z\)

by (auto intro: trans simp: xs-def)

thus \(?\text{thesis using } i j\) unfolding n-def

by (auto simp: iff-f-le strongly-preferred-def)

qed (insert not-outside, auto simp: strongly-preferred-def)

thus \(u (f (\text{Suc }i)) \ast \text{?pref}\text{' }p (f (\text{Suc }i)) = u (f (\text{Suc }i)) \ast \text{?pref }p (f i)\) by simp

qed

also have \(?\text{sum} - (\ldots + u (f 0) \ast \text{?pref}\text{' }p (f 0)) = \sum i<\text{n}. (u (f i) - u (f (\text{Suc }i))) \ast \text{?pref }p (f i)\) - u (f 0) \ast \text{?pref}\text{' }p (f 0) + u (f n) \ast \text{?pref }p (f n)\)

by (simp add: ring-distrib)

also have \(\{y, f 0 <\le y\} = \{\}\)

using hd-weak-ranking[of f 0] xs-nonempty f not-outside

by (auto simp: hd-conv-nth xs-def strongly-preferred-def)

also have \(\{y, \text{ le } (f n) \text{ y}\} = \text{carrier}\)

using last-weak-ranking[of f n] xs-nonempty f not-outside

by (auto simp: last-conv-nth xs-def n-def)

also from \text{p} have measure-pmf.prob \text{p} \text{carrier} = 1

by (subst measure-pmf.prob-eq-1)

(auto simp: AE-measure-pmf-iff lotteries-on-def)

finally show \(?\text{thesis}\) by simp

qed

from \text{assms}[\text{THEN this}] show measure-pmf.expectation \text{p} \text{ u} \le measure-pmf.expectation \text{q} \text{ u}

unfolding SD-preorder n-def using \text{f' }

by (auto intro!: sum-mono mult-left-mono SD simp: utility-le-iff f-le)

next
\textbf{assume} \(\forall u. \text{is-vnm-utility} u \rightarrow \text{measure-pmf.expectation} p u \leq \text{measure-pmf.expectation} q u\)

\textbf{hence} \(\text{expected-utility-le}: \text{measure-pmf.expectation} p u \leq \text{measure-pmf.expectation} q u\)

\textbf{if is-vnm-utility} \(u\) \textbf{for} \(u\) \textbf{using that by} \textbf{blast}

\textbf{define} 
\(x s\) where 
\(x s = \text{weak-ranking le}\)

\textbf{have} 
\(\text{le}: \text{le} = \text{of-weak-ranking} x s\) and \([\text{simp}]: \text{is-weak-ranking} x s\)

\textbf{by} \(\text{simp-all add:} x s-\text{def weak-ranking-total-preorder}\)

\textbf{have carrier:} carrier = \(\bigcup (\text{set} x s)\)

\textbf{by} \(\text{simp add:} x s-\text{def weak-ranking-Union}\)

\textbf{from} \(\text{assms have carrier-nonempty:} \text{carrier} \neq \{\}\)

\textbf{by} \(\text{(auto simp:} \text{lotteries-on-def set-pmf-not-empty)}\)

\[
\{ \\
\text{fix} x \text{ assume} x: x \in \text{carrier} \\
\text{let } ?\text{idx} = \lambda y. \text{length} x s - \text{weak-ranking-index} y \\
\text{have preferred-subset-carrier:} \text{\{y. le x y\}} \subseteq \text{carrier} \\
\text{using not-outside x by auto} \\
\text{have measure-pmf.prob p \{y. le x y\} / real (length x s)} \leq \\
\text{measure-pmf.prob q \{y. le x y\} / real (length x s)} \\
\text{proof (rule field-le-epsilon)} \\
\text{fix } \varepsilon \text{ :: real assume } \varepsilon : \varepsilon > 0 \\
\text{define} u \text{ where } u y = \text{indicator} \text{\{y. y \geq [le] x\}} y + \varepsilon * ?\text{idx} y \text{ for} y \\
\text{have is-utility:} \text{is-vnm-utility} u \text{ unfolding u-def x\text{-def}} \\
\text{proof (intro vnm-utility.add-left vnm-utility.scaled utility-weak-ranking-index)} \\
\text{fix y z assume le y z} \\
\text{thus indicator \text{\{y. y \geq [le] x\}} y \leq (indicator \text{\{y. y \geq [le] x\}} z :: real)} \\
\text{by (auto intro: trans simp: indicator-def split: if-splits)} \\
\text{qed fact+} \\
\text{have (\Sigma y)[le x y. pmf p y)} \leq \\
\text{(\Sigma y)[le x y. pmf p y) + \varepsilon * (\Sigma y)[carrier. ?\text{idx} y * pmf p y)} \\
\text{using } \varepsilon \text{ by (auto intro!: mult-nonneg-nonneg sum-nonneg pmf-nonneg)} \\
\text{also from expected-utility-le is-utility have} \\
\text{measure-pmf.expectation p u \leq measure-pmf.expectation q u} . \\
\text{with assms} \\
\text{have (\Sigma a)[carrier. a * pmf p a)} \leq (\Sigma a)[carrier. a * pmf q a) \\
\text{by (subst (asm) (1 2) integral-measure-pmf[OF finite-carrier])} \\
\text{(auto simp: lotteries-on-def set-pmf-eq ac-simps)} \\
\text{hence (\Sigma y)[le x y. pmf p y) + \varepsilon * (\Sigma y)[carrier. ?\text{idx} y * pmf p y) \leq} \\
\text{(\Sigma y)[le x y. pmf q y) + \varepsilon * (\Sigma y)[carrier. ?\text{idx} y * pmf q y)} \\
\text{using x preferred-subset-carrier not-outside} \\
\text{by (simp add: u-def sum.distrib finite-carrier algebra-simps sum-distrib-left)} \\
\text{Int-absorb1 cong: rev-conj-cong)} \\
\text{also have (\Sigma y)[carrier. ?\text{idx} y * pmf q y) \leq (\Sigma y)[carrier. length x s * pmf q y)} \\
\text{by (intro sum-mono mult-right-mono) (simp-all add: pmf-nonneg)} \\
\text{also have .. = measure-pmf.expectation q (\lambda-. length x s)} \\
\text{using assms by (subst integral-measure-pmf[OF finite-carrier])}
\]

57
also have \( \ldots = \text{length } xs \) by simp

also have \( \sum y \mid y \leq x \text{ pmf } p y = \text{measure-pmf.prob } p \{ y. \text{le } x y \} \)
using finite-subset[of preferred-subset-carrier finite-carrier]
by (simp add: measure-measure-pmf-finite)

also have \( \sum y \mid y \leq x \text{ pmf } q y = \text{measure-pmf.prob } q \{ y. \text{le } x y \} \)
using finite-subset[of preferred-subset-carrier finite-carrier]
by (simp add: measure-measure-pmf-finite)

finally show \( \text{measure-pmf.prob } p \{ y. \text{le } x y \} \leq \frac{\text{length } xs}{\text{length } xs} \leq \text{measure-pmf.prob } q \{ y. \text{le } x y \} \)
using \( \varepsilon \) by (simp add: divide-simps)

qed

moreover from carrier-nonempty carrier have \( xs \neq [] \) by auto
ultimately have \( \text{measure-pmf.prob } p \{ y. \text{le } x y \} \leq \text{measure-pmf.prob } q \{ y. \text{le } x y \} \)
by (simp add: field-simps)

with assms show \( p \preceq [SD(\leq)] q \) unfolding SD-preorder preferred-alts-def by blast

qed

lemma not-strict-SD-iff:
assumes \( p \in \text{lotteries-on } carrier q \in \text{lotteries-on } carrier \)
shows \( \neg (p < [SD(\leq)] q) \iff (\exists u. \text{is-vnm-utility } u \land \text{measure-pmf.expectation } q u \leq \text{measure-pmf.expectation } p u) \)
proof
let \( ?E = \text{measure-pmf.expectation :: 'a pmf } \Rightarrow - \Rightarrow \) real
assume \( \exists u. \text{is-vnm-utility } u \land ?E p u \geq ?E q u \)
then obtain \( u \) where \( u : \text{is-vnm-utility } u \land ?E p u \geq ?E q u \) by blast
interpret \( u : \text{vnm-utility carrier le } u \) by fact

show \( \neg p < [SD \leq] q \)
proof
assume \( \text{less. } p < [SD \leq] q \)
with assms have \( pq : ?E p u \leq ?E q u \) if \( \text{is-vnm-utility } u \) for \( u \)
using that by (auto simp: SD-iff-expected-utilities-le strongly-preferred-def)
with \( u \) have \( u\text{-eq. } ?E p u = ?E q u \) by (intro antisym) simp-all
from less assms obtain \( u' \) where \( u' : \text{is-vnm-utility } u' \land ?E p u' < ?E q u' \)
by (auto simp: SD-iff-expected-utilities-le strongly-preferred-def not-le)
interpret \( u' : \text{vnm-utility carrier le } u' \) by fact

have \( \exists \varepsilon > 0. \text{is-vnm-utility } (\lambda x. u x - \varepsilon * u' x) \)
by (intro u.diff-epsilon antisym u'.utility-le)
then guess \( \varepsilon \) by (elim \( \exists \varepsilon ) \text{ conjE) note } \varepsilon = \text{this \( \)}
define \( u'' \) where \( u'' x = u x - \varepsilon * u' x \) for \( x \)
interpret \( u'' : \text{vnm-utility carrier le } u'' \)
unfolding u''-def by fact
have \( \text{exp-u''. } ?E p u'' = ?E p u - \varepsilon * ?E p u' \) if \( p \in \text{lotteries-on } carrier \) for \( p \) using that

58
by (subst (1 2 3) integral-measure-pmf[of carrier])
(auto simp: lotteries-on-def u''-def algebra-simps sum-subtractf sum-distrib-left)
from assms ε have ?E p u'' > ?E q u''
by (simp-all add: exp-u'' algebra-simps u-eq u')
with p[OF u'':vnm-utility-axioms] show False by simp
qed
qed (insert assms utility-weak-ranking-index,
  auto simp: strongly-preferred-def SD-iff-expected-utilities-le not-le not-less intro: antisym)

lemma strict-SD-iff:
  assumes p ∈ lotteries-on carrier q ∈ lotteries-on carrier
  shows (p ≺[SD(le)] q) ←→ (∀ u. is-vnm-utility u −→ measure-pmf.expectation p u < measure-pmf.expectation q u)
  using not-strict-SD-iff[OF assms] by auto

end

end

theory SD-Efficiency
imports Complex-Main Preference-Profiles Lotteries Stochastic-Dominance
begin

context pref-profile-wf
begin

lemma SD-inefficient-support-subset:
  assumes inefficient: ¬SD-efficient R p'
  assumes support: set-pmf p' ⊆ set-pmf p
  assumes lotteries: p ∈ lotteries-on alts
  shows ¬SD-efficient R p
proof –
  from assms have p'-wf: p' ∈ lotteries-on alts by (simp add: lotteries-on-def)
  from inefficient obtain q' i where q': q' ∈ lotteries-on alts i ∈ agents
    ∧ i. i ∈ agents =⇒ q' ≥[SD(R i)] p' q' ≥[SD(R i)] p'
  unfolding SD-efficient-def' by blast
  have subset: {x. pmf p' x > pmf q' x} ⊆ set-pmf p' by (auto simp: set-pmf-eq)
  also have ... ⊆ set-pmf p by fact
  also have ... ⊆ alts using lotteries by (simp add: lotteries-on-def)
  finally have finite: finite {x. pmf p' x > pmf q' x}
    using finite-alts by (rule finite-subset)
  define ε where ε = Min (insert 1 {pmf p x / (pmf p' x - pmf q' x) |x. pmf p'}
\( x > \text{pmf } q' x )\)
define supp where supp = set-pmf \( p \cup \text{set-pmf } q' \)
from lotteries finite-alts \( q'(1) \) have finite-supp: finite supp
  by (auto simp: lotteries-on-def supp-def dest: finite-subset)
from support have [simp]: \( \text{pmf } p x = \text{0 } \) \( \text{pmf } p' x = \text{0 } \) \( \text{pmf } q' x = \text{0 } \) if \( x \notin \text{supp} \) for \( x \)
  using that by (auto simp: supp-def set-pmf-eq)
from finite support subset have \( \varepsilon : \varepsilon > \text{0 } \) unfolding \( \varepsilon \)-def
  by (auto simp: field-simps set-pmf-eq)
have nonneg: pmf \( p x + \varepsilon * (\text{pmf } q' x - \text{pmf } p' x ) \geq \text{0 } \) for \( x \)
proof (cases pmf \( p' x > \text{pmf } q' x )\)
  case True
  with finite have \( \varepsilon \leq \text{pmf } p x / (\text{pmf } p' x - \text{pmf } q' x )\)
  unfolding \( \varepsilon \)-def by (intro Min-le) auto
  with True show ?thesis by (simp add: field-simps)
next
  case False
  with pmf-nonneg[of \( p x \)] \( \varepsilon \) show ?thesis by simp
qed
define \( q \) where \( q = \text{embed-pmf } (\lambda x. \text{pmf } p x + \varepsilon * (\text{pmf } q' x - \text{pmf } p' x )) \)
have \( (\int \varepsilon x. \text{ennreal } (\text{pmf } p x + \varepsilon * (\text{pmf } q' x - \text{pmf } p' x )) \text{ count-space } \text{UNIV} ) = \text{1 }\)
proof (subst nn-integral-count-space)
  have \( (\sum x : \text{supp. } \text{ennreal } (\text{pmf } p x + \varepsilon * (\text{pmf } q' x - \text{pmf } p' x ))) = \text{ennreal } ((\sum x : \text{supp. } \text{pmf } p x ) + \varepsilon * ((\sum x : \text{supp. } \text{pmf } q' x ) - (\sum x : \text{supp. } \text{pmf } p' x )))\)
    by (subst sum-ennreal[OF nonneg], rule ennreal-cong)
    (auto simp: sum-subtractf ring-distrib sum_distrib sum_distrib_left)
  also from finite-supp support have \( \ldots = \text{1 }\)
    by (subst (1 2 3) sum-pmf-eq-1) (auto simp: supp-def)
  finally show \( (\sum x : \text{supp. } \text{ennreal } (\text{pmf } p x + \varepsilon * (\text{pmf } q' x - \text{pmf } p' x ))) = \text{1 }\).
qed (insert nonneg finite-supp, simp-all)
with nonneg have pmf-q: \( \text{pmf } q x = \text{pmf } p x + \varepsilon * (\text{pmf } q' x - \text{pmf } p' x )\) for \( x \)
  unfolding \( q \)-def by (intro pmf-embed-pmf) simp-all
with support have support-q: \( \text{set-pmf } q \subseteq \text{supp} \)
  unfolding supp-def by (auto simp: set-pmf-eq)
with lotteries support \( q'(1) \) have \( q \)-wf: \( q \in \text{lotteries-on alts} \)
  by (auto simp add: lotteries-on-def supp-def)
from support-q support have expected-utility:
  measure-pmf expectation \( q u = \text{measure-pmf } \text{expectation } p u + \varepsilon * (\text{measure-pmf } \text{expectation } q' u - \text{measure-pmf } \text{expectation } p' u ) \) for \( u \)
  by (subst (1 2 3 4) integral-measure-pmf[OF finite-supp])
  (auto simp: pmf-q supp-def sum_distrib sum_distrib_left
   sum_distrib_right sum_subtractf algebra_simps)
have \( q \succeq [SD(R \, i)] \, p \) \textbf{if} \( i: \, i \in \text{agents for } \, i \)

\begin{verbatim}
proof
  from \( i \) interpret finite-total-preorder-on alts \( R \, i \) \textbf{by} simp
  from \( i \) lotteries \( q'(1) \, q'(3) \) \textbf{[OF i]} \( q\text{-wf} \, p'^\text{-wf} \in \textbf{show} \, ?\text{thesis} \\
    \textbf{by} \, (\text{fastforce simp: SD-iff-expected-utilities-le expected-utility})
  qed

moreover from \( (i \in \text{agents}) \) interpret finite-total-preorder-on alts \( R \, i \) \textbf{by} simp
  from lotteries \( q'(1,4) \) \( q\text{-wf} \, p'^\text{-wf} \in \textbf{have} \, q \succeq [SD(R \, i)] \, p \)
  \textbf{by} \, (force simp: SD-iff-expected-utilities-le expected-utility not-le strongly-preferred-def)
  ultimately show \( ?\text{thesis} \) \textbf{using} \( q\text{-wf} \, i \in \text{agents} \) \textbf{unfolding} SD-efficient-def'
  \textbf{by} \, blast
  qed

lemma SD-efficient-support-subset:
  \textbf{assumes} \( \text{SD-efficient} \, R \, p \) \( \text{set-pmf} \, p' \subseteq \text{set-pmf} \, p \) \( p \in \text{lotteries-on alts} \)
  \textbf{shows} \( \text{SD-efficient} \, R \, p' \)
  \textbf{using} \( \text{SD-inefficient-support-subset}[\text{OF } - \text{assms}(2,3)] \) \textbf{assms}(1) \textbf{by} blast

lemma SD-efficient-same-support:
  \textbf{assumes} \( \text{set-pmf} \, p = \text{set-pmf} \, p' \) \( p \in \text{lotteries-on alts} \)
  \textbf{shows} \( \text{SD-efficient} \, R \, p \leftrightarrow \text{SD-efficient} \, R \, p' \)
  \textbf{using} \( \text{SD-inefficient-support-subset}[\text{of } p \, p'] \) \( \text{SD-inefficient-support-subset}[\text{of } p' \, p] \)
  \textbf{assms}
  \textbf{by} \, (auto simp: lotteries-on-def)

lemma SD-efficient-iff:
  \textbf{assumes} \( p \in \text{lotteries-on alts} \)
  \textbf{shows} \( \text{SD-efficient} \, R \, p \leftrightarrow \text{SD-efficient} \, R \, (\text{pmf-of-set} \, (\text{set-pmf} \, p)) \)
  \textbf{using} \textbf{assms finite-alts}
  \textbf{by} \, (intro SD-efficient-same-support)
  \textbf{(simp, subst set-pmf-of-set,}
  \textbf{auto simp: set-pmf-not-empty lotteries-on-def intro: finite-subset[OF - finite-alts])}

lemma SD-efficient-no-pareto-loser:
  \textbf{assumes} efficient: \( \text{SD-efficient} \, R \, p \) \textbf{and} \, \( p\text{-wf}: \, p \in \text{lotteries-on alts} \)
  \textbf{shows} \( \text{set-pmf} \, p \cap \text{pareto-losers} \, R \, = \, \{\} \)

\begin{verbatim}
proof
  have \( x \notin \text{pareto-losers} \, R \, \textbf{if} \, x: \, x \in \text{set-pmf} \, p \, \textbf{for} \, x \)
  proof
    from \( x \) have \( \text{set-pmf} \, (\text{return-pmf} \, x) \subseteq \text{set-pmf} \, p \) \textbf{by} auto
    from \( \text{efficient this } \, p\text{-wf} \) \textbf{have} \( \text{SD-efficient} \, R \, (\text{return-pmf} \, x) \)
    \textbf{by} \, (rule SD-efficient-support-subset)
    moreover from \textbf{assms} \( x \) have \( x \in \text{alts} \) \textbf{by} \, (auto simp: lotteries-on-def)
    ultimately \textbf{show} \( x \notin \text{pareto-losers} \, R \, \textbf{by} \, (simp add: SD-efficient-singleton-iff)
    qed
  thus \( ?\text{thesis} \) \textbf{by} blast
  qed

Given two lotteries with the same support where one is strictly Pareto-
SD-preferred to the other, one can construct a third lottery that is weakly Pareto-SD-preferred to the better lottery (and therefore strictly Pareto-SD-preferred to the worse lottery) and whose support is a strict subset of the original supports.

**Lemma** improve-lottery-support-subset:

**Assumes** $p \in \text{lotteries-on alts} \ q \in \text{lotteries-on alts} \ q \succ [\text{Pareto}(SD \circ R)] \ p$

**Obtains** $r$ where $r \in \text{lotteries-on alts} \ r \succeq [\text{Pareto}(SD \circ R)] \ q \text{ set-pmf } r \subset \text{set-pmf } p$

**Proof**
- **Have** subset: $\{ x. \ \text{pmf } p \ x > \text{pmf } q \ x \} \subseteq \text{set-pmf } p$ by (auto simp: set-pmf-eq)
- **Also have** .. $\subseteq \text{alts}$ **Using assms** by (simp add: lotteries-on-def)
- **Finally have** finite: finite $\{ x. \ \text{pmf } p \ x > \text{pmf } q \ x \}$
  **Using** finite-alts **By** (rule finite-subset)

from assms **Have** $q \neq p$ by (auto simp: strongly-preferred-def)

**Hence** ex-less: $\exists x. \text{ pmf } p \ x > \text{pmf } q \ x$ by (rule pmf-neq-exists-less)

**Define** $\varepsilon$ where $\varepsilon = \text{Min } \{ \text{pmf } p \ x / (\text{pmf } p \ x - \text{pmf } q \ x) \mid x. \ \text{pmf } p \ x > \text{pmf } q \ x \}$

**Define** supp where supp = set-pmf p

from assms **Have** finite-alts **Have** finite-supp: finite supp
  **By** (auto simp: lotteries-on-def supp-def dest: finite-subset)

from assms **Have** [simp]: pmf p x = 0 pmf q x = 0 if $x \notin \text{supp } x$
  **Using** that by (auto simp: supp-def set-pmf-eq)

from finite subset ex-less **Have** $\varepsilon: \varepsilon \geq 1$ unfolding $\varepsilon$-def
  **By** (intro Min.boundedI) (auto simp: field-simps pmf-nonneg)

**Have** nonneg: pmf p $x + \varepsilon \ast (\text{pmf } q \ x - \text{pmf } p \ x) \geq 0$ for x

**Proof** (cases pmf p x > pmf q x)
  - **Case** True
    **With** finite **Have** $\varepsilon \leq \text{pmf } p \ x / (\text{pmf } p \ x - \text{pmf } q \ x)$
    **Unfolding** $\varepsilon$-def **By** (intro Min-le) auto
    **With** True **Show** ?thesis by (simp add: field-simps)
  - **Next**
    **Case** False
    **With** pmf-nonneg[of p x] $\varepsilon$ **Show** ?thesis by simp

**Qed**

**Define** r where $r = \text{embed-pmf} (\lambda x. \text{pmf } p \ x + \varepsilon \ast (\text{pmf } q \ x - \text{pmf } p \ x))$

**Have** ( $\int x. \text{ennreal } (\text{pmf } p \ x + \varepsilon \ast (\text{pmf } q \ x - \text{pmf } p \ x)) \circ \text{count-space } \text{UNIV}$)
  = 1

**Proof** (subst nn-integral-count-space)
  **Have** ( $\sum x \in \text{supp. ennreal } (\text{pmf } p \ x + \varepsilon \ast (\text{pmf } q \ x - \text{pmf } p \ x)) =$
    ennreal (($\sum x \in \text{supp. pmf } p \ x + \varepsilon \ast ((\sum x \in \text{supp. pmf } q \ x) - (\sum x \in \text{supp. pmf } p \ x))$))
  **By** (subst sum-ennreal[OF nonneg], rule ennreal-cong)
    (auto simp: sum-subtractf ring-distrib sum-distrib-left)
  **Also from** finite-supp **Have** .. = 1

62
by (subst (1 2 3) sum-pmf-eq-1) (auto simp: supp-def assms)
finally show \((\sum x \in \text{supp. enreal (pmf p x + \varepsilon \ast (pmf q x - pmf p x))}) = 1\).
qed (insert nonneg finite-supp, simp-all)

with nonneg have pmf-r: pmf r x = pmf p x + \varepsilon \ast (pmf q x - pmf p x) for x
  unfolding r-def by (intro pmf-embed-pmf) simp-all

with assms have set-pmf r \subseteq \text{supp}
  unfolding supp-def by (auto simp: set-pmf-eq)

ultimately have \(x \in \text{set-pmf p - set-pmf r}\) by (auto simp: set-pmf-iff)
  with (set-pmf r \subseteq \text{supp}) have support-r: set-pmf r \subset set-pmf p unfolding sup-def by blast
  from this assms have \(r \in \text{lotteries-on alts}\) by (simp add: lotteries-on-def)

have \(r \succeq [\text{Pareto}(\text{SD} \circ R)] q\) unfolding SD.Pareto-iff unfolding o-def

proof
  fix \(i\) assume \(i: i \in \text{agents}\)
  then interpret finite-total-preorder-on alts R i by simp
  show \(r \succeq [\text{SD}(R i)] q\)
  proof (subst SD-iff-expected-utilities-le; safe?)
    fix \(u\) assume \(u: \text{is-vnm-utility}\ u\)
    from support-r have expected-utility-r:
      measure-pmf.expectation r u = measure-pmf.expectation p u + \(\varepsilon \ast (\text{measure-pmf.expectation q u - measure-pmf.expectation p u})\)
    by (subst (1 2 3 4) integral-measure-pmf[OF finite-supp])
      (auto simp: supp-def assms pmf-r sum-distrib sum-distrib-left
        sum-distrib-right sum-subtractf algebra-simps)
    from assms i have \(q \succeq [\text{SD}(R i)] p\) by (simp add: SD.Pareto-strict iff)
    with assms u have measure-pmf.expectation q u \geq measure-pmf.expectation p u
    proof
      by (simp add: SD-iff-expected-utilities-le r-wf)
    hence \((\varepsilon - 1) \ast \text{measure-pmf.expectation p u} \leq (\varepsilon - 1) \ast \text{measure-pmf.expectation q u}\)
      using \(\varepsilon\) by (intro mult-left-mono) simp-all
    thus measure-pmf.expectation q u \leq measure-pmf.expectation r u
      by (simp add: algebra-simps expected-utility-r)
    qed fact+
  qed
  from that[OF r-wf this support-r] show \(?thesis\).
  qed
4.5 Existence of SD-efficient lotteries

In this section, we will show that any lottery can be ‘improved’ to an SD-efficient lottery, i.e. for any lottery, there exists an SD-efficient lottery that is weakly SD-preferred to the original one by all agents.

context

fixes p :: 'alt lottery

assumes lott: p ∈ lotteries-on alts

begin

private definition improve-lottery :: 'alt lottery ⇒ 'alt lottery where
improve-lottery q = (let A = {r∈lotteries-on alts. r ∨[Pareto(SD◦R)] q} in
(SOME r. r ∈ A ∧ ¬(∃ r′∈A. set-pmf r′ ⊂ set-pmf r)))

private lemma improve-lottery:
assumes ¬SD-efficient R q
defines r ≡ improve-lottery q
shows r ∈ lotteries-on alts r ∨[Pareto( SD◦R)] q ⋀ r′ ∈ lotteries-on alts =⇒ r′ ∨[Pareto( SD◦R)] q =⇒ ¬(set-pmf r′ ⊂ set-pmf r)
proof –
define A where A = {r∈lotteries-on alts. r ∨[Pareto(SD◦R)] q}

have subset-alts: X ⊆ alts if X ∈ set-pmf A for X using that

by (auto simp: A-def lotteries-on-def)

have r-altdef: r = (SOME r. r ∈ A ∧ ¬(∃ r′∈A. set-pmf r′ ⊂ set-pmf r))

unfolding r-def improve-lottery-def Let-def A-def by simp

from assms have nonempty: A ≠ {} by (auto simp: A-def SD-efficient-def)

hence nonempty': set-pmf A ≠ {} by simp

have set-pmf ′ A ⊆ Pow alts by (auto simp: A-def lotteries-on-on-def)

from finite-alts have wf: wf {{X,Y}. X ⊆ Y ∧ Y ⊆ alts}

by (rule finite-subset-wf)

obtain X

where X ∈ set-pmf ′ A ⋀ Y. Y ⊆ X ∧ X ⊆ alts =⇒ Y ∉ set-pmf ′ A

by (rule wfE-min[OF this, folded r-altdef]) simp-all

hence ∃ r. r ∈ A ∧ ¬(∃ r′∈A. set-pmf r′ ⊂ set-pmf r)

by (auto simp: subset-alts[of X])

from someI-ex[OF this, folded r-altdef]

show r ∈ lotteries-on alts r ∨[Pareto( SD◦R)] q ⋀ r′ ∈ lotteries-on alts =⇒ r′ ∨[Pareto( SD◦R)] q =⇒ ¬(set-pmf r′ ⊂ set-pmf r)

unfolding A-def by blast+

qed

private fun sd-chain :: nat ⇒ 'alt lottery option where
sd-chain 0 = Some p
| sd-chain (Suc n) =
  (case sd-chain n of
   None ⇒ None
   Some k ⇒ Some (improve-lottery (sd-chain n)))

64
private lemma sd-chain-None-propagate:
\[ m \geq n \implies \text{sd-chain } n = \text{None} \implies \text{sd-chain } m = \text{None} \]
by (induction rule: inc-induct) simp-all

private lemma sd-chain-Some-propagate:
\[ m \geq n \implies \text{sd-chain } m = \text{Some } q \implies \exists q'. \text{sd-chain } n = \text{Some } q' \]
by (cases sd-chain n) (auto simp: sd-chain-None-propagate)

private lemma sd-chain-NoneD:
\[ \text{sd-chain } n = \text{None} \implies \exists n p. \text{sd-chain } n = \text{Some } p \land \text{SD-efficient } R p \]
by (induction n) (auto split: option.splits if-splits)

private lemma sd-chain-lottery: \[ \text{sd-chain } n = \text{Some } q \implies q \in \text{lotteries-on alts} \]
by (induction n) (insert lott, auto split: option.splits if-splits simp: improve-lottery)

private lemma sd-chain-Suc:
\[ \text{assumes } \text{sd-chain } m = \text{Some } q \]
\[ \text{assumes } \text{sd-chain } (\text{Suc } m) = \text{Some } r \]
\[ \text{shows } q \prec [\text{Pareto}(\text{SD } \circ R)] r \]
using assms by (auto split: if-splits simp: improve-lottery)

private lemma sd-chain-strictly-preferred:
\[ \text{assumes } m < n \]
\[ \text{assumes } \text{sd-chain } m = \text{Some } q \]
\[ \text{assumes } \text{sd-chain } n = \text{Some } s \]
\[ \text{shows } q \prec [\text{Pareto}(\text{SD } \circ R)] s \]
using assms
proof (induction arbitrary: q rule: strict-inc-induct)
case (base k q)
with sd-chain-Suc[of k q s] show ?case by (simp del: sd-chain.simps add: o-def)
next
case (step k q)
from step.hyps have Suc k \leq n by simp
from sd-chain-None-propagate[of this, of s] step.prems obtain r
where r: sd-chain (Suc k) = Some r by (auto simp del: sd-chain.simps)
with step.prems have \[ q \prec [\text{Pareto}(\text{SD } \circ R)] r \]
by (intro sd-chain-Suc)
moreover from r step.prems have \[ r \prec [\text{Pareto}(\text{SD } \circ R)] s \]
by (intro step.IH) simp-all
ultimately show ?case by (rule SD.Pareto.strict-trans)
qed

private lemma sd-chain-preferred:
\[ \text{assumes } m \leq n \]
\[ \text{assumes } \text{sd-chain } m = \text{Some } q \]
\[ \text{assumes } \text{sd-chain } n = \text{Some } s \]
\[ \text{shows } q \preceq [\text{Pareto}(\text{SD } \circ R)] s \]
proof (cases m < n)

| Some p \Rightarrow \text{if SD-efficient } R p \text{ then None else Some } (\text{improve-lottery } p) |
case True
  from sd-chain-strictly-preferred[OF this assms(2,3)] show ?thesis
    by (simp add: strongly-preferred-def)

next
case False
with assms show ?thesis by (auto intro: SD.Pareto.refl sd-chain-lottery)

qed

lemma SD-efficient-lottery-exists:
  obtains q where q ∈ lotteries-on alts q ⪰ Pareto(SD◦R)p SD-efficient R q
proof -
  consider ∃ n. sd-chain n = None | ∀ n. ∃ q. sd-chain n = Some q
  using option.exhaust by metis
  thus ?thesis
  proof cases
    case 1
define m where m = (LEAST m. sd-chain m = None)
define k where k = m − 1
    from LeastI-ex[OF 1] have m: sd-chain m = None by (simp add: m-def)
    from m have nz: m ≠ 0 by (intro notI) simp-all
    from nz Least-le[OF λ m. sd-chain m = None m − 1, folded m-def]
    obtain q where q: sd-chain k = Some q by (cases sd-chain (m − 1)) (auto simp: k-def)
    from sd-chain-preferred[OF sd-chain.simps(1) this] have q ⪰ Pareto(SD◦R) by simp
    moreover from q have q ∈ lotteries-on alts by (simp add: sd-chain-lottery)
    moreover from q m have SD-efficient R q by (auto split: if-splits simp: m-altdef)
    ultimately show ?thesis using that[of q] by blast

  next
case 2
  have range (set-pmf ◦ the ◦ sd-chain) ⊆ Pow alts unfolding o-def
    proof safe
    fix n x assume A: x ∈ set-pmf (the (sd-chain n))
    from 2 obtain q where sd-chain n = Some q by auto
    with sd-chain-lottery[of n q] have set-pmf (the (sd-chain n)) ⊆ alts by (simp add: lotteries-on-def)
    with A show x ∈ alts by blast
  qed

  hence finite (range (set-pmf ◦ the ◦ sd-chain)) by (rule finite-subset) simp-all
  from pigeonhole-infinite[OF infinite-UNIV-nat this]
  obtain m where infinite { n. set-pmf (the (sd-chain n)) = set-pmf (the (sd-chain m))}
    by auto
  hence infinite (⎨ n. set-pmf (the (sd-chain n)) = set-pmf (the (sd-chain m))⎬)
  − {k. ¬(k > m)}
    by (simp add: not-less)
  hence (⎨ n. set-pmf (the (sd-chain n)) = set-pmf (the (sd-chain m))⎬) − {k.
\neg (k > m)} \neq \{\}
by \text{intro notI simp-all }
\text{ then obtain } n \text{ where } mn: n > m \text{ set-pmf (the (sd-chain n)) = set-pmf (the (sd-chain m))}
by blast
from 2 obtain p q where pq: sd-chain m = Some p sd-chain n = Some q by blast
from mn pq have supp-eq: set-pmf p = set-pmf q by simp
from mn(1) pq have less: p \prec \text{[Pareto(SD \circ R)]} q by \text{(rule sd-chain-strictly-preferred)}
from (m < n) have n > 0 by simp
with \text{sd-chain n = Some q: sd-chain.simps(2)[of n - 1]}
\text{ obtain r where r: \neg SD-efficient R r q = improve-lottery r}
by \text{(auto simp del: sd-chain.simps split: if-splits option.splits)}
from pq have p \in \text{lotteries-on alts q \in \text{lotteries-on alts}}
by \text{(simp-all add: sd-chain-lottery)}
from improve-lottery-support-subset[\text{OF this less supp-eq}] guess s . note s = this
from improve-lottery(2)[of r] r s have s \prec \text{[Pareto(SD \circ R)]} r
by \text{(auto intro: SD.Pareto.strict-weak-trans)}
from improve-lottery(3)[OF r(1) s(1) this] supp-eq r
have \neg set-pmf s \subset set-pmf p by simp
with s(3) show \text{thesis by contradiction}
qed
qed
end

lemma
\text{assumes p \in \text{lotteries-on alts}}
\text{ shows } \exists q \in \text{lotteries-on alts. q \geq [Pareto(SD \circ R)] p} \land \text{SD-efficient R q}
\text{ using SD-efficient-lottery-exists[OF assms] by blast}
end
end

5 Social Decision Schemes

theory Social-Decision-Schemes
imports
Complex-Main
HOL-Probability Probability
Preference-Profiles
Elections
Order-Predicates
Stochastic-Dominance
SD-Efficiency

67
5.1 Basic Social Choice definitions

context election
begin

The set of lotteries, i.e. the probability mass functions on the type 'alt whose support is a subset of the alternative set.

abbreviation lotteries where
  lotteries ≡ lotteries-on alts

The probability that a lottery returns an alternative that is in the given set

abbreviation lottery-prob :: 'alt lottery ⇒ 'alt set ⇒ real where
  lottery-prob ≡ measure-pmf.prob

lemma lottery-prob-alts-superset:
  assumes p ∈ lotteries alts ⊆ A
  shows lottery-prob p A = 1
  using assms by (subst measure-pmf.prob-eq-1) (auto simp: AE-measure-pmf-iff lotteries-on-def)

lemma lottery-prob-alts: p ∈ lotteries ⇒ lottery-prob p alts = 1
  by (rule lottery-prob-alts-superset) simp-all

end

In the context of an election, a preference profile is a function that assigns to each agent her preference relation (which is a total preorder)

5.2 Social Decision Schemes

In the context of an election, a Social Decision Scheme (SDS) is a function that maps preference profiles to lotteries on the alternatives.

locale social-decision-scheme = election agents alts for agents :: 'agent set and alts :: 'alt set +
  fixes sds :: ('agent, 'alt) pref-profile ⇒ 'alt lottery
  assumes sds-wf: is-pref-profile R ⇒ sds R ∈ lotteries

5.3 Anonymity

An SDS is anonymous if permuting the agents in the input does not change the result.

locale anonymous-sds = social-decision-scheme agents alts sds
  for agents :: 'agent set and alts :: 'alt set and sds +
  assumes anonymous: π permutes agents ⇒ is-pref-profile R ⇒ sds (R o π) = sds R
begin

lemma anonymity-prefs-from-table:
 assumes prefs-from-table-wf agents alts xs prefs-from-table-wf agents alts ys
 shows \( \text{sds (prefs-from-table xs)} = \text{sds (prefs-from-table ys)} \)
proof
  from prefs-from-table-agent-permutation[OF assms] guess \( \pi \).
  with anonymous[of \( \pi \), of prefs-from-table xs] assms(1) show ?thesis
    by (simp add: pref-profile-from-tableI)
qed

context begin

qualified lemma anonymity-prefs-from-table-aux:
 assumes \( R_1 = \text{prefs-from-table xs} \) prefs-from-table-wf agents alts xs
 assumes \( R_2 = \text{prefs-from-table ys} \) prefs-from-table-wf agents alts ys
 assumes mset (map snd xs) = mset (map snd ys)
 shows \( \text{sds} \ R_1 = \text{sds} \ R_2 \)
  unfolding assms(1,3)
  by (rule anonymity-prefs-from-table) 
  (simp-all add: assms del: mset-map)
end

end

5.4 Neutrality

An SDS is neutral if permuting the alternatives in the input does not change the result, modulo the equivalent permutation in the output lottery.

locale neutral-sds = social-decision-scheme agents alts sds
 for agents :: 'agent set and alts :: 'alt set and sds +
 assumes neutral: \( \sigma \) permutes alts \( \implies \) is-pref-profile \( R \implies \)
\( \text{sds (permute-profile} \ \sigma \ R) = \text{map-pmf} \ \sigma \ (\text{sds} \ R) \)
begin

Alternative formulation of neutrality that shows that our definition is equivalent to that in the paper.

lemma neutral':
 assumes \( \sigma \) permutes alts
 assumes is-pref-profile \( R \)
 assumes \( a \in \text{alts} \)
 shows \( \text{pmf} \ (\text{sds (permute-profile} \ \sigma \ R)) \ (\sigma \ a) = \text{pmf} \ (\text{sds} \ R) \ a \)
proof
  from assms have A: set-pmf \( (\text{sds} \ R) \subseteq \text{alts} \) using sds-uf
    by (simp add: lotteries-on-def)
  from assms(1,2) have \( \text{pmf} \ (\text{sds (permute-profile} \ \sigma \ R)) \ (\sigma \ a) = \text{pmf} \ (\text{map-pmf} \ \sigma \ (\text{sds} \ R)) \ (\sigma \ a) \)
    by (subst neutral) simp-all
  also from assms have \( \ldots = \text{pmf} \ (\text{sds} \ R) \ a \)

end
by (intro pmf-map-inj') (simp-all add: permutes-inj)
finally show ?thesis .
qed

end

locale an-sds =
  anonymous-sds agents alts sds + neutral-sds agents alts sds
for agents :: 'agent set and alts :: 'alt set and sds
begin

lemma sds-anonymous-neutral:
  assumes perm: σ permutes alts and wf: is-pref-profile R1 is-pref-profile R2
  assumes eq: anonymous-profile R1 =
    image-mset (map (('') σ)) (anonymous-profile R2)
  shows sds R1 = map-pmf σ (sds R2)
proof
  interpret R1: pref-profile-wf agents alts R1 by fact
  interpret R2: pref-profile-wf agents alts R2 by fact
  from perm have wf': is-pref-profile (permute-profile σ R2)
    by (rule R2.wf-permute-alts)
  from eq perm have anonymous-profile R1 = anonymous-profile (permute-profile σ R2)
    by (simp add: R2.anonymous-profile-permute)
  from anonymous-profile-agent-permutation[OF this wf(1) wf']
  obtain π where π permutes agents permute-profile σ R2 o π = R1 by auto
  have sds (permute-profile σ R2 o π) = sds (permute-profile σ R2)
    by (rule anonymous) fact+
  also have ... = map-pmf σ (sds R2)
    by (rule neutral) fact+
  also have permute-profile σ R2 o π = R1 by fact
  finally show ?thesis .
qed

lemma sds-anonymous-neutral':
  assumes perm: σ permutes alts and wf: is-pref-profile R1 is-pref-profile R2
  assumes eq: anonymous-profile R1 =
    image-mset (map (('') σ)) (anonymous-profile R2)
  shows pmf (sds R1) (σ x) = pmf (sds R2) x
proof
  have sds R1 = map-pmf σ (sds R2) by (intro sds-anonymous-neutral) fact+
  also have pmf ... (σ x) = pmf (sds R2) x by (intro pmf-map-inj' permutes-inj[OF perm])
  finally show ?thesis .
qed

lemma sds-automorphism:
assumes \( \text{perm}: \sigma \text{ permutes alts and } \text{wf}: \text{is-pref-profile } R \)

assumes \( \text{eq}: \text{image-mset} (\text{map} ((\cdot) \sigma)) (\text{anonymous-profile } R) = \text{anonymous-profile } R \)

shows \( \text{map-pmf } \sigma (\text{sds } R) = \text{sds } R \)

using \( \text{sds-anonymous-neutral} [\text{OF } \text{perm } \text{wf } \text{wf } \text{eq} [\text{symmetric}]] \) ..

end

lemma \( \text{an-sds-automorphism-aux} \):

assumes \( \text{wf}: \text{prefs-from-table-wf } \text{agents alts } yss R \equiv \text{prefs-from-table } yss \)

assumes \( \text{an}: \text{an-sds } \text{agents alts } \text{sds} \)

assumes \( \text{eq}: \text{mset} (\text{map} ((\cdot) (\text{permutation-of-list } \text{xs}))) \circ \text{snd} \) \text{yss} = \text{mset} (\text{map } \text{snd } \text{yss})

assumes \( \text{perm}: \text{set} (\text{map } \text{fst } \text{xs}) \subseteq \text{alts set} (\text{map } \text{snd } \text{xs}) = \text{set} (\text{map } \text{fst } \text{xs}) \)

and \( x: x \in \text{alts } y = \text{permutation-of-list } \text{xs } x \)

shows \( \text{pmf } \text{sds } R \) \text{x} = \( \text{pmf } \text{map-pmf } \sigma (\text{sds } R) \) (\( \sigma x \))

proof –

note \( \text{perm} = \text{list-permutatesI } [\text{OF } \text{perm}] \)

let \( \sigma = \text{permutation-of-list } \text{x} \)

note \( \text{perm' } = \text{permutation-of-list } \text{permutes } [\text{OF } \text{perm}] \)

from \( \text{wf } \text{have } \text{wf' } \text{: pref-profile-wf } \text{agents alts } \text{sds } R \text{ by } (\text{simp add: pref-profile-from-tableI})) \)

then interpret \( R \text{: pref-profile-wf } \text{agents alts } \text{sds } \text{.} \)

from \( \text{perm' } \text{interpret } \text{R' } \text{: pref-profile-wf } \text{agents alts } \text{permute-profile } ?\sigma \text{ } \text{R } \text{.} \)

by (simp add: \text{R.anonymous-profile-permute pref-profile-from-tableI})

from \( \text{eq } \text{wf } \text{have } \text{eq' } \text{: image-mset} (\text{map} ((\cdot) ?\sigma)) (\text{anonymous-profile } \text{R}) = \text{anonymous-profile } \text{R } \text{.} \)

by (simp add: \text{anonymise-prefs-from-table mset-map multiset.map-comp})

from \( \text{perm' } \text{x have } \text{pmf } \text{(sds } \text{R} \text{) } \text{x } \text{=} \text{pmf } \text{(map-pmf } ?\sigma \text{ (sds } \text{R} \text{)}) (?\sigma \text{ } \text{x} ) \text{ by } (\text{simp add: pmf-map-inj' permutes-inj})) \)

also from \( \text{eq' } \text{x } \text{wf' } \text{perm' } \text{have } \text{map-pmf } ?\sigma \text{ (sds } \text{R} \text{) } \text{=} \text{sds } \text{R } \text{.} \)

by (intro \text{sdss-automorphism})

(simp-all add: \text{R.anonymous-profile-permute pref-profile-from-tableI})

finally show \( \text{thesis } \text{using } \text{x } \text{by simp } \text{.} \)

qed

5.5 Ex-post efficiency

locale \( \text{ex-post-efficient-sds } = \text{social-decision-scheme } \text{agents alts } \text{sds } \text{.} \)

for \text{agents } :: \text{'agent } \text{set and alts } :: \text{'alt } \text{set and } \text{sds } +

assumes \( \text{ex-post-efficient:} \)

\( \text{is-pref-profile } R \Rightarrow \text{set-pmf } (\text{sds } \text{R}) \cap \text{pareto-losers } \text{R } \text{=} \text{\{}} \) 

begin

lemma \( \text{ex-post-efficient' } \):

assumes \( \text{is-pref-profile } R \ y \Rightarrow[\text{Pareto } \text{R}] \text{ x } \)

shows \( \text{pmf } (\text{sds } \text{R}) \text{ x } \text{=} \text{0 } \)

71
using ex-post-efficient[of R] assms
by (auto simp: set-pmf-eq pareto-losers-def)

lemma ex-post-efficient'':
  assumes is-pref-profile R i ∈ agents ∀ i ∈ agents. y ≽ [R i] x → y ≺ [R i] x
  shows pmf (sds R) x = 0
proof
  from assms(1) interpret pref-profile-wf agents alts R .
  from assms(2−) show ?thesis
    by (intro ex-post-efficient"[OF assms(1), of - y])
        (auto simp: Pareto-iff strongly-preferred-def)
qed

end

5.6 SD efficiency

An SDS is SD-efficient if it returns an SD-efficient lottery for every preference profile, i.e. if the SDS outputs a lottery, it is never the case that there is another lottery that is weakly preferred by all agents and strictly preferred by at least one agent.

locale sd-efficient-sds =
  social-decision-scheme agents alts sds
for agents :: 'agent set and alts :: 'alt set and sds +
assumes SD-efficient: is-pref-profile R ⇒ SD-efficient R (sds R)
begin
An alternative formulation of SD-efficiency that is somewhat more convenient to use.

lemma SD-efficient':
  assumes is-pref-profile R q ∈ lotteries
  assumes ∀ i ∈ agents ⇒ q ≽ [SD(R i)] sds R i ∈ agents q ≻ [SD(R i)] sds R
  shows P
proof
  interpret pref-profile-wf agents alts R by fact
  show ?thesis
    using SD-efficient[of R] sds-wf[OF assms(1)] assms unfolding SD-efficient-def'
    by blast
qed

Any SD-efficient SDS is also ex-post efficient.

sublocale ex-post-efficient-sds
proof unfold-locales
  fix R :: ('agent, 'alt) pref-profile assume R-wf: is-pref-profile R
  interpret pref-profile-wf agents alts R by fact
  from R-wf show set-pmf (sds R) ∩ pareto-losers R = {}
    by (intro SD-efficient-no-pareto-loser SD-efficient sds-wf)
qed
The following rule can be used to derive facts from inefficient supports: If a set of alternatives is an inefficient support, at least one of the alternatives in it must receive probability 0.

**lemma** SD-inefficient-support:
assumes $A$: $A \neq \emptyset$ $A \subseteq$ alts and inefficient: $\neg$SD-efficient $R$ (pmf-of-set $A$)
assumes wf: is-pref-profile $R$
shows $\exists x \in A$. pmf (sds $R$) $x = 0$

**proof** (rule ccontr)
interpret pref-profile-wf agents alts $R$ by fact
assume $\neg (\exists x \in A$. pmf (sds $R$) $x = 0$)
with $A$ have set-pmf (pmf-of-set $A$) $\subseteq$ set-pmf (sds $R$)
by (subt set-pmf-of-set) (auto simp: set-pmf-eq intro: finite-subset[OF -finite-alts])
from inefficient and this have $\neg$SD-efficient $R$ (sds $R$)
by (rule SD-inefficient-support-subset) (simp add: wf sds-wf)
moreover from SD-efficient wf have SD-efficient $R$ (sds $R$).
ultimately show False by contradiction
qed

**lemma** SD-inefficient-support':
assumes wf: is-pref-profile $R$
assumes $A$: $A \neq \emptyset$ $A \subseteq$ alts and
wit: $p \in$ lotteries $\forall i \in$ agents. $p \succeq$[SD($R$ $i$)] pmf-of-set $A$ $i \in$ agents
shows $\exists x \in A$. pmf (sds $R$) $x = 0$

**proof** (rule SD-inefficient-support)
from wf interpret pref-profile-wf agents alts $R$.
from wit show $\neg$SD-efficient $R$ (pmf-of-set $A$)
by (intro SD-inefficientI') (auto intro!: bexI[of - i] simp: strongly-preferred-def)
qed fact+

end

5.7 Weak strategyproofness

**context** social-decision-scheme

**begin**

The SDS is said to be manipulable for a particular preference profile, a particular agent, and a particular alternative preference ordering for that agent if the lottery obtained if the agent submits the alternative preferences strictly SD-dominates that obtained if the original preferences are submitted. (SD-dominated w.r.t. the original preferences)

**definition** manipulable-profile
:: ('agent', 'alt') pref-profile $\Rightarrow$ 'agent $\Rightarrow$ 'alt relation $\Rightarrow$ bool where
manipulable-profile $R$ $i$ $R_i'$ $\longleftrightarrow$ sds ($R$($i$ := $R_i'$)) $\succ$[SD($R$ $i$)] sds $R$

end
An SDS is weakly strategyproof (or just strategyproof) if it is not manip- 
ulable for any combination of preference profiles, agents, and alternative 
preference relations.

locale strategyproof-sds = social-decision-scheme agents alts sds 
for agents :: 'agent set and alts :: 'alt set and sds + 
assumes strategyproof: 
is-pref-profile R ⇒ i ∈ agents ⇒ total-preorder-on alts Ri' ⇒ 
¬manipulable-profile R i Ri'

5.8 Strong strategyproofness

context social-decision-scheme

begin

The SDS is said to be strongly strategyproof for a particular preference 
profile, a particular agent, and a particular alternative preference ordering 
for that agent if the lottery obtained if the agent submits the alternative 
preferences is SD-dominated by the one obtained if the original preferences 
are submitted. (SD-dominated w.r.t. the original preferences)

In other words: the SDS is strategyproof w.r.t the preference profile R and 
the agent i and the alternative preference relation Ri if the lottery for ob-
tained for R is at least as good for i as the lottery obtained when i misrep-
resents her preferences as Ri'.

definition strongly-strategyproof-profile
:: ('agent, 'alt) pref-profile ⇒ 'agent ⇒ 'alt relation ⇒ bool where 
strongly-strategyproof-profile R i Ri' ←→ sds R ⪰[SD (R i)] sds (R(i := Ri'))

lemma strongly-strategyproof-profileI [intro]:
assumes is-pref-profile R total-preorder-on alts Ri' i ∈ agents 
assumes \( \forall x. x \in alts \Rightarrow lottery-prob (sds (R(i := Ri'))) (preferred-alts (R i) x) \) 
\( \leq \) lottery-prob (sds R) (preferred-alts (R i) x) 
shows strongly-strategyproof-profile R i Ri' 
proof – 
interpret pref-profile-wf agents alts R by fact 
show ?thesis 
unfolding strongly-strategyproof-profile-def 
by rule (auto intro!: sds-wf assms pref-profile-wf wf-update) 
qed

lemma strongly-strategyproof-imp-not-manipulable:
assumes strongly-strategyproof-profile R i Ri' 
sows ¬manipulable-profile R i Ri' 
using assms unfolding strongly-strategyproof-profile-def manipulable-profile-def 
by (auto simp: strongly-preferred-def)

end
An SDS is strongly strategyproof if it is strongly strategyproof for all combinations of preference profiles, agents, and alternative preference relations.

locale strongly-strategyproof-sds = social-decision-scheme agents alts sds
for agents :: 'agent set and alts :: 'alt set and sds +
assumes strongly-strategyproof:
  is-pref-profile R ⇒ i ∈ agents ⇒ total-preorder-on alts Ri' ⇒ strongly-strategyproof-profile R i Ri'

begin

Any SDS that is strongly strategyproof is also weakly strategyproof.

sublocale strategyproof-sds
by unfold-locales
  (simp add: strongly-strategyproof-imp-not-manipulable strongly-strategyproof)

end

locale strategyproof-an-sds =
strategyproof-sds agents alts sds + an-sds agents alts sds

end

6 Lowering Social Decision Schemes

theory SDS-Lowering
imports Social-Decision-Schemes
begin

definition lift-pref-profile ::
  'agent set ⇒ 'alt set ⇒ 'agent set ⇒ 'alt set ⇒
  ('agent, 'alt) pref-profile ⇒ ('agent, 'alt) pref-profile
where
lift-pref-profile agents alts agents' alts' R = (λi x y.
  x ∈ alts' ∧ y ∈ alts' ∧ i ∈ agents' ∧
  (x = y ∨ x ∉ alts ∨ i ∉ agents ∨ (y ∈ alts ∧ R i x y)))

lemma lift-pref-profile-wf:
  assumes pref-profile-wf agents alts R
  assumes agents ⊆ agents' alts ⊆ alts' finite alts'
  defines R' ≡ lift-pref-profile agents alts agents' alts' R
  shows pref-profile-wf agents' alts' R'
proof –
  from assms interpret R: pref-profile-wf agents alts by simp
  have finite-total-preorder-on alts' (R' i)
  if i: i ∈ agents' for i
  proof (cases i ∈ agents)
    case True
    then interpret finite-total-preorder-on alts R i by simp
from True assms show ?thesis
  by unfold-locales (auto simp: lift-pref-profile-def dest: total intro: trans)
next
case False
  with assms i show ?thesis
    by unfold-locales (simp-all add: lift-pref-profile-def)
qed
moreover have $R' \equiv (\lambda\cdot\cdot\cdot. \text{False})$ if $i \notin \text{agents'}$ for $i$
  unfolding lift-pref-profile-def $R'$-def using that by simp
ultimately show ?thesis unfolding pref-profile-wf-def using assms by auto
qed

lemma lift-pref-profile-permute-agents:
  assumes $\pi$ permutes agents agents $\subseteq$ agents'
  shows lift-pref-profile agents alts agents' $R \circ \pi) =$
    lift-pref-profile agents alts agents' $R \circ \pi$
  using assms permutes-subset[OF assms]
  by (auto simp add: lift-pref-profile-def $R'$-def permutes-in-image)

lemma lift-pref-profile-permute-alts:
  assumes $\sigma$ permutes alts alts $\subseteq$ alts'
  shows lift-pref-profile agents alts agents' $\text{permute-profile} \sigma R) =$
    $\text{permute-profile} \sigma (\text{lift-pref-profile agents alts agents'}$ $R)$
proof
from assms have inv: inv $\sigma$ permutes alts by (intro permutes-inv)
from this assms(2) have inv $\sigma$ permutes alts' by (rule permutes-subset)
with inv show ?thesis using assms permutes-inj[OF (inv $\sigma$ permutes alts)]
  by (fastforce simp add: lift-pref-profile-def permutes-in-image
    fun-eq-iff dest: injD)
qed

lemma lotteries-on-subset: $A \subseteq B \implies p \in \text{lotteries-on} A \implies p \in \text{lotteries-on} B$
  unfolding lotteries-on-def by blast

lemma lottery-prob-carrier: $p \in \text{lotteries-on} A \implies \text{measure-pmf.prob} p A = 1$
  by (auto simp: measure-pmf.prob-eq-1 lotteries-on-def AE-measure-pmf-iff)

context
fixes agents alts $R$ agents' alts' $R'$
assumes $R\text{-uf}: \text{pref-profile-wf agents alts} R$
assumes election: agents $\subseteq$ agents' alts $\subseteq$ alts' alts $\neq \{\}$ agents $\neq \{\}$ finite alts'
defines $R' \equiv \text{lift-pref-profile agents alts agents'}$ alts' $R$
begin
interpretation $R: \text{pref-profile-wf agents alts} R$
interpretation $R': \text{pref-profile-wf agents'}$ alts' $R'$
using election $R$-uf by (simp add: $R'$-def lift-pref-profile-wf)
lemma lift-pref-profile-strict-iff:  
  \( x \prec ([\text{lift-pref-profile agents alts agents' alts'} R i]) y \longleftrightarrow \)  
  \( i \in \text{agents} \land ((y \in \text{alts} \land x \in \text{alts'}) \lor x \prec (R i) y) \)  
proof (cases \( i \in \text{agents} \))  
  case True  
  then interpret total-preorder-on alts R i by simp  
  show ?thesis using \( \text{not-outside election} \)  
  by (auto simp: lift-pref-profile-def strongly-preferred-def)  
qed (simp-all add: lift-pref-profile-def strongly-preferred-def)

lemma preferred-alts-lift-pref-profile:  
  assumes \( i \in \text{agents'} \) and \( x \in \text{alts'} \)  
  shows \( \text{preferred-alts} (R' i) x = \)  
  \( \text{if } i \in \text{agents} \land x \in \text{alts} \text{ then } \text{preferred-alts} (R i) x \text{ else alts'} \)  
proof (cases \( i \in \text{agents} \))  
  assume \( i \in \text{agents} \)  
  then interpret \( R i : \text{total-preorder-on alts R i} \) by simp  
  show ?thesis using \( i x \text{ election R i. not-outside} \)  
  by (auto simp: preferred-alts-def R' def lift-pref-profile-def R refl)  
qed (auto simp: preferred-alts-def R' def lift-pref-profile-def i x)

lemma lift-pref-profile-Pareto-iff:  
  \( x \preceq ([\text{Pareto}] (R' i)) y \longleftrightarrow x \in \text{alts'} \land y \in \text{alts'} \land (x \notin \text{alts} \land y \in \text{alts'}) \)  
proof  
  from \( \text{R. nonempty-agents obtain i where } i : i \in \text{agents} \) by blast  
  then interpret \( R i : \text{finite-total-preorder-on alts R i} \) by simp  
  show ?thesis unfolding \( R'. \text{ Pareto-iff} R. \text{ Pareto-iff} \) unfolding \( R' \text{-def lift-pref-profile-def} \)  
  using \( \text{election i by (auto simp: preorder-on. refl[OF R.in-dom]} \)  
  simp del: \( \text{R. nonempty-agents R. nonempty-agents intro: R. not-outside} \)  
qed

lemma lift-pref-profile-Pareto-strict-iff:  
  \( x \prec ([\text{Pareto}] (R')) y \longleftrightarrow x \in \text{alts'} \land y \in \text{alts'} \land (x \notin \text{alts} \land y \in \text{alts} \lor x \prec (\text{Pareto}(R)) y) \)  
by (auto simp: strongly-preferred-def lift-pref-profile-Pareto-iff R. Pareto. not-outside)

lemma pareto-losers-lift-pref-profile:  
  shows \( \text{pareto-losers} R' = \text{pareto-losers} R \cup (\text{alts'} - \text{alts}) \)  
proof  
  have \( A: x \in \text{alts} \land y \in \text{alts} \text{ if } x \prec (\text{Pareto}(R)) y \text{ for } x y \)  
  using that \( R. \text{ Pareto. not-outside unfolding} \) strongly-preferred-def by auto  
  have \( B: x \in \text{alts'} \text{ if } x \in \text{alts} \text{ for } x \text{ using } \text{election that by blast} \)  
  from \( \text{R. nonempty-alts obtain x where } x : x \in \text{alts} \text{ by blast} \)  
  thus ?thesis unfolding \( \text{pareto-losers-def lift-pref-profile-Pareto-strict-iff} \) [abs-def]  
  by (auto dest: A B)  
qed

context
begin
private lemma lift-SD-iff-agent:
assumes \( p \in \text{lotteries-on alts} \) \( q \in \text{lotteries-on alts} \) \( i : i \in \text{agents} \)
shows \( p \preceq [SD(R^i)] q \iff p \preceq [SD(R^i)] q \)
proof –
  from \( i \) interpret \( R_i : \text{preorder-on alts} \) \( R^i \) by simp
  from \( i \) election have \( i' : i \in \text{agents}' \) by blast
  then interpret \( R^i : \text{preorder-on alts} \) \( R^i \) by simp
  from \( \text{assms election} \) have \( p \in \text{lotteries-on alts}' q \in \text{lotteries-on alts}' \)
  by (auto intro: lotteries-on-subset)
  with \( \text{assms election i'} \) show \( ?\text{thesis} \)
  by (auto simp: \( R^i\).SD-preorder \( R^i\).SD-preorder preferred-alts-lift-pref-profile lottery-prob-carrier)
qed

private lemma lift-SD-iff-nonagent:
assumes \( p \in \text{lotteries-on alts} q \in \text{lotteries-on alts} \) \( i : i \in \text{agents}' - \text{agents} \)
shows \( p \preceq [SD(R^i)] q \)
proof –
  from \( i \) election have \( i' : i \in \text{agents}' \) by blast
  then interpret \( R^i : \text{preorder-on alts} \) \( R^i \) by simp
  from \( \text{assms election} \) have \( p \in \text{lotteries-on alts}' q \in \text{lotteries-on alts}' \)
  by (auto intro: lotteries-on-subset)
  with \( \text{assms election i'} \) show \( ?\text{thesis} \)
  by (auto simp: \( R^i\).SD-preorder preferred-alts-lift-pref-profile lottery-prob-carrier)
qed

lemmas lift-SD-iff = lift-SD-iff-agent lift-SD-iff-nonagent

lemma lift-SD-iff':
\( p \in \text{lotteries-on alts} \implies q \in \text{lotteries-on alts} \implies i \in \text{agents}' \implies \)
\( p \preceq [SD(R^i)] q \iff i \notin \text{agents} \lor p \preceq [SD(R^i)] q \)
by (cases \( i \in \text{agents} \)) (simp-all add: lift-SD-iff)
end

lemma lift-SD-strict-iff:
assumes \( p \in \text{lotteries-on alts} q \in \text{lotteries-on alts} \) \( i : i \in \text{agents} \)
shows \( p \prec [SD(R^i)] q \iff p \preceq [SD(R^i)] q \)
using \( \text{assms} \) by (simp add: strongly-preferred-def lift-SD-iff)

lemma lift-Pareto-SD-iff:
assumes \( p \in \text{lotteries-on alts} q \in \text{lotteries-on alts} \)
shows \( p \preceq [\text{Pareto}(SD \circ R^i)] q \iff p \preceq [\text{Pareto}(SD \circ R)] q \)
using \( \text{assms election} \) by (auto simp: R.SD.Pareto-iff R'.SD.Pareto-iff lift-SD-iff')

lemma lift-Pareto-SD-strict-iff:
assumes \( p \in \text{lotteries-on alts} q \in \text{lotteries-on alts} \)
shows \( p \prec [\text{Pareto}(SD \circ R^i)] q \iff p \prec [\text{Pareto}(SD \circ R)] q \)
using assms by (simp add: strongly-preferred-def lift-Pareto-SD-iff)

lemma lift-SD-efficient-iff:
assumes p: p ∈ lotteries-on alts
shows SD-efficient R' p ←→ SD-efficient R p
proof
  assume eff: SD-efficient R' p
  have ¬(q >:[Pareto(SD R)] p) if q: q ∈ lotteries-on alts for q
  proof
    from q election have q': q ∈ lotteries-on alts by (blast intro: lotteries-on-subset)
    with eff have ¬(q >:[Pareto(SD R')] p) by (simp add: R'.SD-efficient-def)
    with p q show thesis by (simp add: lift-Pareto-SD-strict-iff)
  qed
  thus SD-efficient R p by (simp add: R'.SD-efficient-def)
next
  assume eff: SD-efficient R p
  have ¬(q >:[Pareto(SD R')] p) if q: q ∈ lotteries-on alts' for q
  proof
    assume less: q >:[Pareto(SD R')] p
    from R'.SD-efficient-lottery-exists[OF q] guess q'. note q' = this
    have x /∈ set-pmf q' if x: x ∈ alts' − alts for x
    proof
      from x have x ∈ pareto-losers R' by (simp add: pareto-losers-lift-pref-profile)
      with R'.SD-efficient-no-pareto-loser[OF q'(3,1)] show x /∈ set-pmf q' by blast
    qed
    with q' have q' ∈ lotteries-on alts by (auto simp: lotteries-on-def)
    moreover from q' less have q' >:[Pareto(SD R')] p
    by (auto intro: R'.SD.Pareto.strict-weak-trans)
    with q' ∈ lotteries-on alts: p have q' >:[Pareto(SD R)] p
    by (subst (asm) lift-Pareto-SD-strict-iff)
    ultimately have ¬SD-efficient R p by (auto simp: R'.SD-efficient-def)
    with eff show False by contradiction
  qed
  thus SD-efficient R' p by (simp add: R'.SD-efficient-def)
qed
end

locale sds-lowering =
ex-post-efficient-sds agents alts sds
for agents :: 'agent set and alts :: 'alt set and sds +
fixes agents' alts'
assumes agents'-subset: agents' ⊆ agents and alts'-subset: alts' ⊆ alts
and agents'-nonempty [simp]: agents' ≠ {} and alts'-nonempty [simp]: alts' ≠ {}
lemma finite-agents' [simp]: finite agents'
  using agents'-'subset finite-agents by (rule finite-subset)

lemma finite-alts' [simp]: finite alts'
  using alts'-'subset finite-alts by (rule finite-subset)

abbreviation lift :: ('agent, 'alt) pref-profile ⇒ ('agent, 'alt) pref-profile
  where
  lift ≡ lift-pref-profile agents alts

definition lowered :: ('agent, 'alt) pref-profile ⇒ 'alt lottery
  where
  lowered = sds ◦ lift

lemma lift-wf [simp, intro]:
  pref-profile-wf agents alts' R ⇒ is-pref-profile (lift R)
  using alts'-'subset agents'-'subset by (intro lift-pref-profile-wf) simp-all

sublocale lowered: election agents' alts'
  by unfold-locales simp-all

lemma preferred-alts-lift:
  lowered.is-pref-profile R ⇒ i ∈ agents ⇒ x ∈ alts ⇒
  preferred-alts (lift R i) x =
  (if i ∈ agents' ∧ x ∈ alts' then preferred-alts (R i) x else alts)
  using alts'-'subset agents'-'subset
  by (intro preferred-alts-lift-pref-profile) simp-all

lemma pareto-losers-lift:
  lowered.is-pref-profile R ⇒ pareto-losers (lift R) = pareto-losers R ∪ (alts − alts')
  using alts'-'subset alts'-'subset by (intro pareto-losers-lift-pref-profile) simp-all

lemma lowered-lotteries:
  lowered.lotteries ⊆ lotteries
  unfolding lotteries-on-def using alts'-'subset by blast

sublocale lowered: social-decision-scheme agents' alts' lowered

proof
  fix R assume R-wf: pref-profile-wf agents' alts' R
  from R-wf have R'-wf: pref-profile-wf agents alts (lift R) by (rule lift-wf)
  show lowered R ∈ lowered.lotteries unfolding lotteries-on-def
  proof safe
    fix x assume x ∈ set-prof (lowered R)
    hence x: x ∈ set-prof (sds (lift R)) by (simp add: lowered-def)
    with ex-post-efficient[OF R'-wf]
    have x /∈ pareto-losers (lift R) by blast
    with pareto-losers-lift[OF R'-wf]
    have x /∈ alts − alts' by blast
    moreover from x have x ∈ alts using sds-wf[OF R'-wf]
    by (auto simp: lotteries-on-def)
    ultimately show x ∈ alts' by simp
qed

qed

sublocale \textit{ex-post-efficient-sds agents' alts' lowered}
proof
\begin{itemize}
  \item \textbf{fix} $R$ \textbf{assume} $R\text{-wf}$ \textbf{lowered.is-pref-profile} $R$
  \item \textbf{hence} \textbf{is-pref-profile} $(\text{lift } R)$ \textbf{by simp}
  \item \textbf{have} \textbf{set-pmf} $(\text{lowered } R) \cap \text{pareto-losers} (\text{lift } R) = \{\}$ \textbf{unfolding} \textbf{lowered-def o-def} \textbf{by} (intro \textit{ex-post-efficient lift-wf} $R\text{-wf}$)
  \item \textbf{also have} \textbf{pareto-losers} $(\text{lift } R) = \text{pareto-losers } R \cup (\text{alts} - \text{alts'}$
  \item \textbf{by} (intro \textit{pareto-losers-lift} $R\text{-wf}$)
  \item \textbf{finally show} \textbf{set-pmf} $(\text{lowered } R) \cap \text{pareto-losers } R = \{\}$ \textbf{by blast}
\end{itemize}

qed

lemma \textit{lowered-in-lotteries} [simp]: \textbf{lowered.is-pref-profile} $R \implies \text{lowered } R \in \text{lotteries}$
\begin{itemize}
  \item \textbf{using} \textit{lowered.sds-wf [of }$R$\text{]} \textbf{lowered-lotteries} \textbf{by blast}
\end{itemize}
end

locale \textit{sds-lowering-anonymous} =
\begin{itemize}
  \item \textit{anonymous-sds agents alts sds + \textit{sds-lowering agents alts sds agents' alts'}}
  \item \textbf{for agents :: 'agent set \textbf{and alts :: 'alt set and sds agents' alts'}}
\end{itemize}
begin

sublocale \textit{lowered: anonymous-sds agents' alts' lowered}
proof
\begin{itemize}
  \item \textbf{fix} $\pi$ $R$ \textbf{assume} $\text{perm:\, }\pi$ \textbf{permutes agents' and }$R\text{-wf}$ \textbf{lowered.is-pref-profile} $R$
  \item \textbf{from} \textbf{perm} \textbf{have} \textbf{lift} $(R \circ \pi) = \text{lift } R \circ \pi$
  \item \textbf{using} \textbf{agents'-subset} \textbf{by} (rule \textit{lift-pref-profile-permute-agents})
  \item \textbf{hence} \textbf{sds} $(\text{lift } (R \circ \pi)) = \text{sds } (\text{lift } R \circ \pi)$ \textbf{by simp}
  \item \textbf{also from} \textbf{perm} $R\text{-wf}$ \textbf{have} $\pi$ \textbf{permutes agents is-pref-profile} $(\text{lift } R)$
  \item \textbf{using} \textbf{agents'-subset} \textbf{by} (auto \textbf{dest: permutes-subset})
  \item \textbf{from} \textbf{anonymous[OF this]} \textbf{have} \textbf{sds} $(\text{lift } R \circ \pi) = \text{sds } (\text{lift } R)$
  \item \textbf{by} (simp \textbf{add: lowered-def})
  \item \textbf{finally show} \textbf{lowered } $(R \circ \pi) = \text{lowered } R$ \textbf{unfolding} \textbf{lowered-def o-def} .
\end{itemize}

qed

end

locale \textit{sds-lowering-neutral} =
\begin{itemize}
  \item \textit{neutral-sds agents alts sds + \textit{sds-lowering agents alts sds agents' alts'}}
  \item \textbf{for agents :: 'agent set \textbf{and alts :: 'alt set and sds agents' alts'}}
\end{itemize}
begin

sublocale \textit{lowered: neutral-sds agents' alts' lowered}

end

81
proof

fix \sigma R assume perm: \sigma permutes alts' and R-wf: lowered.is-pref-profile R
from perm alts'-subset
  have lift (permute-profile \sigma R) = permute-profile \sigma (lift R)
  by (rule lift-pref-profile-permute-alts)
hence sds (lift (permute-profile \sigma R)) = sds (permute-profile \sigma (lift R)) by simp
also from R-wf perm have is-pref-profile (lift R) by simp
with perm alts'-subset
  have sds (permute-profile \sigma (lift R)) = map-pmf \sigma (sds (lift R))
  by (intro neutral) (auto intro: permutes-subset)
finally show lowered (permute-profile \sigma R) = map-pmf \sigma (lowered R)
  by (simp add: lowered-def o-def)
qed

end

locale sds-lowering-sd-efficient =
  sd-efficient-sds agents alts sds +
  sds-lowering agents alts sds agents' alts'
begin

sublocale sd-efficient-sds agents alts' lowered
proof
  fix R assume R-wf: lowered.is-pref-profile R
  interpret R: pref-profile-uf agents' alts' R by fact
  from R-wf agents'-subset alts'-subset show SD-efficient R (lowered R)
  unfolding lowered-def o-def
  by (subst lift-SD-efficient-iff [symmetric])
  (insert SD-efficient R-wf lowered.sds-wf[OF R-wf], auto simp: lowered-def)
qed

end

locale sds-lowering-strategyproof =
  strategyproof-sds agents alts sds +
  sds-lowering agents alts sds agents' alts'
begin

sublocale strategyproof-sds agents alts' lowered
proof (unfold-locales, safe)
  fix R i Ri'
  assume R-wf: lowered.is-pref-profile R and i: i \in agents'
  assume Ri': total-preorder-on alts' Ri'
  assume manipulable: lowered.manipulable-profile R i Ri'
  from i agents'-subset have i': i \in agents by blast
  interpret R: pref-profile-uf agents' alts' R by fact
from R-wf interpret liftR: pref-profile-wf agents alts lift R by simp

define lift-Ri' where
lift-Ri' x y ←→ x ∈ alts ∧ y ∈ alts ∧ (x = y ∨ x ∉ alts' ∨ (y ∈ alts' ∧ Ri' x y))

for x y define S where S = (lift R)(i := lift-Ri')
from Ri' interpret Ri': total-preorder-on alts' Ri'.

have wf-lift-Ri'': total-preorder-on alts lift-Ri' using Ri'.total
  by unfold-locales (auto simp: lift-Ri'-def intro: Ri'.trans)
from agents'-subset i have S-altdef: S = lift (R(i := Ri'))
  by (auto simp: fun-eq-iff lift-pref-profile-def lift-Ri'-def S-def)
have lowered (R(i := Ri')) ∈ lowered.lotteries
  by (intro lowered.sds-wf R.wf-update i Ri')

hence sds-S-wf: sds S ∈ lowered.lotteries by (simp add: S-altdef lowered-def)

from manipulable have lowered R ≺[SD (R i)] sds (lift (R(i := Ri')))
  unfolding lowered.manipulable-profile-def by (simp add: lowered-def)
also note S-altdef [symmetric]
finally have lowered R ≺[SD (lift R i)] sds S
  using R-wf i lowered.sds-wf[OF R-wf] sds-S-wf
  by (subst lift-SD-strict iff) (simp-all add: agents'-subset alts'-subset)

hence manipulable-profile (lift R) i lift-Ri'
  by (simp add: manipulable-profile-def lowered-def S-def)
with strategyproof[OF lift-wf[OF R-wf] i' wf-lift-Ri'] show False by contradiction
qed

end

locale sds-lowering-anonymous-neutral-sd-efficient-strategyproof =
  sds-lowering-anonymous + sds-lowering-neutral +
  sds-lowering-sd-efficient + sds-lowering-strategyproof

end

7 Random Dictatorship

theory Random-Dictatorship
imports
  Complex-Main
  Social-Decision-Schemes
begin

We define Random Dictatorship as a social decision scheme on total preorders (i.e. agents are allowed to have ties in their rankings) by first selecting an agent uniformly at random and then selecting one of that agents’ most preferred alternatives uniformly at random. Note that this definition also works for weak preferences.
definition random-dictatorship :: 'agent set ⇒ 'alt set ⇒ ('agent, 'alt) pref-profile ⇒ 'alt lottery where
random-dictatorship-auxdef:
random-dictatorship agents alts R =
do {i ← pmf-of-set agents;
pmf-of-set (Max-wrt-among (R i) alts)}

context election
begin

abbreviation RD :: ('agent, 'alt) pref-profile ⇒ 'alt lottery where
RD ≡ random-dictatorship agents alts

lemma random-dictatorship-def:
assumes is-pref-profile R
shows RD R =
do {
i ← pmf-of-set agents;
pmf-of-set (favorites R i)}

proof –
from assms interpret pref-profile-wf agents alts R .
show ?thesis by (simp add: random-dictatorship-auxdef favorites-altdef)
qed

lemma random-dictatorship-unique-favorites:
assumes is-pref-profile R has-unique-favorites R
shows RD R = map-pmf (favorite R) (pmf-of-set agents)

proof –
from assms(1) interpret pref-profile-wf agents alts R .
from assms(2) interpret pref-profile-unique-favorites agents alts R by unfold-locales
show ?thesis unfolding random-dictatorship-def[OF assms(1)] map-pmf-def
by (intro bind-pmf-cong) (auto simp: unique-favorites pmf-of-set-singleton)
qed

lemma random-dictatorship-unique-favorites':
assumes is-pref-profile R has-unique-favorites R
shows RD R = pmf-of-multiset (image-mset (favorite R) (mset-set agents))
using assms by (simp add: random-dictatorship-unique-favorites map-pmf-of-set)

lemma pmf-random-dictatorship:
assumes is-pref-profile R
shows pmf (RD R) x =
(∑ i∈agents. indicator (favorites R i) x / real (card (favorites R i))) / real (card agents)

proof –
from assms(1) interpret pref-profile-wf agents alts R .
from nonempty-dom have card agents > 0 by (auto simp del: nonempty-agents)

hence ennreal (pmf (RD R) x) = ennreal (sum i∈agents. pmf (pmf-of-set (favorites R i)) x) / real (card agents)

(is _ = ennreal (?p / -)) unfolding random-dictatorship-def[OF assms]


also have ?p = (sum i∈agents. indicator (favorites R i) x / real (card (favorites R i)))

by (intro sum.cong) (simp-all add: favorites-nonempty)

finally show ?thesis

by (subst (asm) ennreal-inj) (auto intro!: sum-nonneg divide-nonneg-nonneg)

qed

sublocale RD: social-decision-scheme agents alts RD

proof

fix R assume R-wf: is-pref-profile R

then interpret pref-profile-wf agents alts R .

from R-wf show RD R ∈ lotteries

using favorites-subset-alts favorites-nonempty

by (auto simp: lotteries-on-def random-dictatorship-def)

qed

We now show that Random Dictatorship fulfils anonymity, neutrality, and strong strategyproofness. At the very least, this shows that the definitions of these notions are consistent.

7.1 Anonymity

The following proof is essentially the following: In Random Dictatorship, permuting the agents in the preference profile is the same as applying the permutation to the agent that was picked uniformly at random in the first step. However, uniform distributions are invariant under permutation, therefore the outcome is totally unchanged.

sublocale RD: anonymous-sds agents alts RD

proof

fix R π assume uf: is-pref-profile R and perm: π permutes agents

interpret pref-profile-uf agents alts R by fact

from uf-permute-agents[OF perm]

have RD (R ◦ π) = map-pmf π (pmf-of-set agents) ≫= (∃ i. pmf-of-set (favorites R i))

by (simp add: bind-map-pmf random-dictatorship-def o_def favorites-def)

also from perm uf have ... = RD R

by (simp add: map-pmf-of-set-inj permutes-inj-on permutes-image random-dictatorship-def)

finally show RD (R ◦ π) = RD R .

qed

85
7.2 Neutrality

The proof of neutrality is similar to that of anonymity. We have proven elsewhere that the most preferred alternatives of an agent in a profile with permuted alternatives are simply the image of the originally preferred alternatives. Since we pick one alternative from the most preferred alternatives of the selected agent uniformly at random, this means that we effectively pick an agent, then pick one of her most preferred alternatives, and then apply the permutation to that alternative, which is simply Random Dictatorship transformed with the permutation.

sublocale RD: neutral-sds agents alts RD
proof
  fix σ R
  assume perm: σ permutes alts and R-wf: is-pref-profile R
  from R-wf interpret pref-profile-wf agents alts R .
  from wf-permute-alts[OF perm] R-wf perm show RD (permute-profile σ R) = map-pmf σ (RD R)
    by (subst random-dictatorship-def)
      (auto intro!: bind-pmf-cong simp: random-dictatorship-def map-bind-pmf
                   favorites-permute map-pmf-of-set-inj permutes-inj-on favorites-nonempty)
qed

7.3 Strong strategyproofness

The argument for strategyproofness is quite simple: Since the preferences submitted by an agent \(i\) only influence the outcome when that agent is picked in the first process, it suffices to focus on this case. When the agent \(i\) submits her true preferences, the probability of obtaining a result at least as good as \(x\) (for any alternative \(x\)) is 1, since the outcome will always be one of her most-preferred alternatives. Obviously, the probability of obtaining such a result cannot exceed 1 no matter what preferences she submits instead, and thus, RD is strategyproof.

sublocale RD: strongly-strategyproof-sds agents alts RD
proof (unfold-locales, unfold RD, strongly-strategyproof-profile-def)
  fix R i Ri' assume R-wf: is-pref-profile R and i: i ∈ agents
    and Ri'-wf: total-preorder-on alts Ri'
  interpret R: pref-profile-wf agents alts R by fact
  from R-wf Ri'-wf i have R'-wf: is-pref-profile (R(i := Ri'))
    by (simp add: R_wf-update)
  interpret R': pref-profile-wf agents alts R(i := Ri') by fact

  show SD (R i) (RD (R(i := Ri')))) (RD R)
  proof (rule R_SD-pref-profile1)
    fix x assume x ∈ alts
    hence emeasure (measure-pmf (RD (R(i := Ri')))) (preferred-alts (R i) x)
      ≤ emeasure (measure-pmf (RD R)) (preferred-alts (R i) x)
    using Ri'-wf maximal-imp-preferred[of R i x]
Random Serial Dictatorship is an anonymous, neutral, strongly strategy-proof, and ex-post efficient Social Decision Scheme that extends Random Dictatorship to the domain of weak preferences.

We define RSD using a fold over a random permutation. Effectively, we choose a random order of the agents (in the form of a list) and then traverse that list from left to right, where each agent in turn removes all the alternatives that are not top-ranked among the remaining ones.

The following two facts correspond give an alternative recursive definition to the above definition, which uses random permutations and list folding.

Theorem 8 Random Serial Dictatorship

theory Random-Serial-Dictatorship
imports
  Complex-Main
  Social-Decision-Schemes
  Random-Dictatorship
begin

Random Serial Dictatorship is an anonymous, neutral, strongly strategy-proof, and ex-post efficient Social Decision Scheme that extends Random Dictatorship to the domain of weak preferences.

We define RSD using a fold over a random permutation. Effectively, we choose a random order of the agents (in the form of a list) and then traverse that list from left to right, where each agent in turn removes all the alternatives that are not top-ranked among the remaining ones.

definition random-serial-dictatorship ::
  'agent set ⇒ 'alt set ⇒ ('agent, 'alt) pref-profile ⇒ 'alt lottery where
  random-serial-dictatorship agents alts R =
  fold-bind-random-permutation (λi alts. Max-wrt-among (R i) alts) pmf-of-set alts agents

The following two facts correspond give an alternative recursive definition to the above definition, which uses random permutations and list folding.

lemma random-serial-dictatorship-empty [simp]:
  random-serial-dictatorship {} alts R = pmf-of-set alts
  by (simp add: random-serial-dictatorship-def)

lemma random-serial-dictatorship-nonempty:
  finite agents ⇒ agents ≠ {} ⇒⇒
  random-serial-dictatorship agents alts R =
do {
  i ← pmf-of-set agents;
  random-serial-dictatorship (agents − {i}) (Max-wrt-among (R i) alts) R
}

by (simp add: random-serial-dictatorship-def)

We define the RSD winners w.r.t. a given set of alternatives and a fixed permutation (i.e. list) of agents. In contrast to the above definition, the RSD winners are determined by traversing the list of agents from right to left. This may seem strange, but it makes induction much easier, since induction over foldr does not require generalisation over the set of alternatives and is therefore much easier than over foldl.

definition rsd-winners where
rsd-winners R alts agents = foldr (λi alts. Max-wrt-among (R i) alts) agents alts

lemma rsd-winners-empty [simp]: rsd-winners R alts [] = alts
by (simp add: rsd-winners-def)

lemma rsd-winners-Cons [simp]:
rsd-winners R alts (i # agents) = Max-wrt-among (R i) (rsd-winners R alts agents)
by (simp add: rsd-winners-def)

lemma rsd-winners-map [simp]:
rsd-winners R alts (map f agents) = rsd-winners (R ◦ f) alts agents
by (simp add: rsd-winners-def foldr-map o-def)

There is now another alternative definition of RSD in terms of the RSD winners. This will mostly be used for induction.

lemma random-serial-dictatorship-altdef:
assumes finite agents
shows random-serial-dictatorship agents alts R =
do {
  agents' ← pmf-of-set (permutations-of-set agents);
  pmf-of-set (rsd-winners R alts agents')
}
by (simp add: random-serial-dictatorship-def fold-bind-random-permutation-foldr assms rsd-winners-def)

The following lemma shows that folding from left to right yields the same distribution. This is probably the most commonly used definition in the literature, along with the recursive one.

lemma random-serial-dictatorship-foldl:
assumes finite agents
shows random-serial-dictatorship agents alts R =
do {
  agents' ← pmf-of-set (permutations-of-set agents);
  pmf-of-set (foldl (λalts i. Max-wrt-among (R i) alts) alts agents')
}
8.1 Auxiliary facts about RSD

8.1.1 Pareto-equivalence classes

First of all, we introduce the auxiliary notion of a Pareto-equivalence class. A non-empty set of alternatives is a Pareto equivalence class if all agents are indifferent between all alternatives in it, and if some alternative \( x \) is contained in the set, any other alternative \( y \) is contained in it if and only if, to all agents, \( y \) is at least as good as \( x \). The importance of this notion lies in the fact that the set of RSD winners is always a Pareto-equivalence class, which we will later use to show ex-post efficiency and strategy-proofness.

**Definition**  

\[
\text{RSD-pareto-eqclass} \quad \text{where} \\
\text{RSD-pareto-eqclass} \text{ agents alts R A} \iff A \neq \emptyset \land A \subseteq \text{alts} \land (\forall x \in A. \forall y \in \text{alts}. \ y \in A \iff (\forall i \in \text{agents}. \ R \ i \ x \ y))
\]

**Lemma**  

\[
\text{RSD-pareto-eqclassI}:
\text{assumes} \ A \neq \emptyset A \subseteq \text{alts} \land (\forall x \in A. \forall y \in \text{alts}. \ y \in A \iff (\forall i \in \text{agents}. \ R \ i \ x \ y))
\text{shows} \ \text{RSD-pareto-eqclass} \text{ agents alts R A}
\text{using} \ \text{assms} \ \text{unfolding} \ \text{RSD-pareto-eqclass-def} \ \text{by} \ \text{simp-all}
\]

**Lemma**  

\[
\text{RSD-pareto-eqclassD}:
\text{assumes} \ A \neq \emptyset A \subseteq \text{alts} \land (\forall x \in A. \forall y \in \text{alts}. \ y \in A \iff (\forall i \in \text{agents}. \ R \ i \ x \ y))
\text{shows} \ A \neq \emptyset A \subseteq \text{alts} \land (\forall x \in A. \forall y \in \text{alts}. \ y \in A \iff (\forall i \in \text{agents}. \ R \ i \ x \ y))
\text{using} \ \text{assms} \ \text{unfolding} \ \text{RSD-pareto-eqclass-def} \ \text{by} \ \text{simp-all}
\]

**Lemma**  

\[
\text{RSD-pareto-eqclass-indiff-set}:
\text{assumes} \ \text{RSD-pareto-eqclass} \text{ agents alts R A i} \in \text{agents} x \in A y \in A
\text{shows} \ R \ i \ x \ y
\text{using} \ \text{assms} \ \text{unfolding} \ \text{RSD-pareto-eqclass-def} \ \text{by} \ \text{blast}
\]

**Lemma**  

\[
\text{RSD-pareto-eqclass-empty} \ [\text{simp, intro}!]:
\text{alts} \neq \emptyset \implies \text{RSD-pareto-eqclass} \{\} \text{ alts} \text{ R alts}
\text{by} \ (\text{auto intro!: RSD-pareto-eqclassI})
\]

**Lemma**  

\[
\text{(in pref-profile-wf) RSD-pareto-eqclass-insert}:
\text{assumes} \ \text{RSD-pareto-eqclass} \text{ agents alts} \text{ R A finite alts}
\text{i} \in \text{agents} \text{agents} \subseteq \text{agents}
\text{shows} \ \text{RSD-pareto-eqclass} \text{ (insert i agents) alts} \text{ R (Max-wrt-among (R i) A)}
\text{proof} –
\text{from} \ \text{assms} \ \text{interpret} \ \text{total-preorder-on alts} \text{ R i by} \ \text{simp}
\text{show} \ \text{thesis}
\text{proof} \ (\text{intro RSD-pareto-eqclassI} \text{ Max-wrt-among-nonempty} \text{ Max-wrt-among-subset},
\]

89
8.1.2 Facts about RSD winners

context pref-profile-wf
begin

Any RSD winner is a valid alternative.

lemma rsd-winners-subset:
assumes set agents′ ⊆ agents
shows rsd-winners R alts′ agents′ ⊆ alts′
proof –
{  fix i assume i ∈ agents  then interpret total-preorder-on alts R i by simp  have Max-wrt-among (R i) A ⊆ A for A  using Max-wrt-among-subset by blast  } note A = this  from ⟨set agents′ ⊆ agents⟩ show rsd-winners R alts′ agents′ ⊆ alts′  using A by (induction agents′) auto
qed

There is always at least one RSD winner.

lemma rsd-winners-nonempty:
assumes finite: finite alts and alts′ ≠ {}  set agents′ ⊆ agents alts′ ⊆ alts
shows rsd-winners R alts′ agents′ ≠ {}  proof –
{  fix i assume i ∈ agents  then interpret total-preorder-on alts R i by simp  have Max-wrt-among (R i) A ≠ {} if A ⊆ alts A ≠ {} for A  using that assms by (intro Max-wrt-among-nonempty) (auto simp: Int-absorb)  } note B = this  with ⟨set agents′ ⊆ agents⟩ ⟨alts′ ⊆ alts⟩ ⟨alts′ ≠ {}⟩ show rsd-winners R alts′ agents′ ≠ {}  proof (induction agents′)
  case (Cons i agents′)
    with B[of i rsd-winners R alts′ agents′] rsd-winners-subset[of agents′ alts′] finite wf  show ?case by auto
Obviously, the set of RSD winners is always finite.

**lemma** rsd-winners-finite:

**assumes** set agents' ⊆ agents finite alts alts' ⊆ alts

**shows** finite (rsd-winners R alts' agents')

**by** (rule finite-subset[OF subset-trans[OF rsd-winners-subset]]) fact+

**lemmas** rsd-winners-uf =

rsd-winners-subset rsd-winners-nonempty rsd-winners-finite

The set of RSD winners is a Pareto-equivalence class.

**lemma** RSD-pareto-eqclass-rsd-winners-aux:

**assumes** finite: finite alts and alts ≠ {} and set agents' ⊆ agents

**shows** RSD-pareto-eqclass (set agents') alts R (rsd-winners R alts agents')

**using** (set agents' ⊆ agents)

**proof** (induction agents')

**case** (Cons i agents')

**from** Cons.prems **show** ?case

by (simp only: set-simps rsd-winners-Cons, intro RSD-pareto-eqclass-insert[OF Cons.IH finite]) simp-all

**qed** (insert assms, simp-all)

**lemma** RSD-pareto-eqclass-rsd-winners:

**assumes** finite: finite alts and alts ≠ {} and set agents' = agents

**shows** RSD-pareto-eqclass agents alts R (rsd-winners R alts agents')

**using** RSD-pareto-eqclass-rsd-winners-aux[of agents'] assms by simp

For the proof of strategy-proofness, we need to define indifference sets and lift preference relations to sets in a specific way.

**context**

**begin**

An indifference set for a given preference relation is a non-empty set of alternatives such that the agent is indifferent over all of them.

**private definition** indiff-set where

indiff-set S A ←→ A ≠ {} ∧ (∀x∈A. ∀y∈A. S x y)

**private lemma** indiff-set-mono: indiff-set S A → B ⊆ A → B ≠ {} → indiff-set S B

**unfolding** indiff-set-def by blast

Given an arbitrary set of alternatives A and an indifference set B, we say that B is set-preferred over A w.r.t. the preference relation R if all (or, equivalently, any) of the alternatives in B are preferred over all alternatives in A.

**private definition** RSD-set-rel where
\[ \text{RSD-set-rel } S \ A \ B \iff \text{indiff-set } S \ B \land (\forall x \in A . \forall y \in B . \ S \ x \ y) \]

The most-preferred alternatives (w.r.t. \( R \)) among any non-empty set of alternatives form an indifference set w.r.t. \( R \).

**Private lemma** indiff-set-Max-wrt-among:
- **Assumes** finite carrier \( A \subseteq \text{carrier } A \neq \{\} \) total-preorder-on carrier \( S \)
- **Shows** indiff-set \( S \) (Max-wrt-among \( S \) \( A \))
- **Unfolding** indiff-set-def

**Proof**
- From \( \text{assms (1 - 3)} \) interpret total-preorder-on carrier \( S \).
- From \( \text{assms (1 - 3)} \) show Max-wrt-among \( S \) \( A \) \( \neq \{\} \) by (intro Max-wrt-among-nonempty) auto
- From \( \text{assms (1 - 3)} \) show \( \forall x \in \text{Max-wrt-among } S \) \( A . \forall y \in \text{Max-wrt-among } S \) \( A . \ S \ x \ y \) by (auto simp: indiff-set-def Max-wrt-among-total-preorder)

**Qed**

We now consider the set of RSD winners in the setting of a preference profile \( R \) and a manipulated profile \( R(i := R_i') \). This theorem shows that the set of RSD winners in the outcome is either the same in both cases or the outcome for the truthful profile is an indifference set that is set-preferred over the outcome for the manipulated profile.

**Lemma** rsd-winners-manipulation-aux:
- **Assumes** wf: total-preorder-on alts \( R_i' \)
- and \( i : i \in \text{agents and set agents'} \subseteq \text{agents finite agents} \)
- and finite: finite alts and \( \text{alts } \neq \{\} \)
- **Defines** [simp]: \( w' \equiv \text{rsd-winners } (R(i := R_i')) \) alts and [simp]: \( w \equiv \text{rsd-winners } R \) alts
- **Shows** \( w' \) agents' = \( w \) agents' \( \lor \) RSD-set-rel \( R(i) \) \( (w' \) agents') \( (w \) agents')
- **Using** [set agents' \( \subseteq \) agents]
- **Proof** (induction agents')
  - **Case** \( (\text{Cons } j \text{ agents'}) \)
    - From \( \text{wf } i \) interpret \( R_i : \text{total-preorder-on alts } R i \) by simp
    - From \( \text{wf Cons.prems interpret } R_j : \text{total-preorder-on alts } R j \) by simp
    - From \( \text{wf } \) interpret \( R_i' : \text{total-preorder-on alts } R_i' \).
    - From \( \text{wf } \) assms Cons.prems
      - Have indiff-set: indiff-set \( R(i) \) (Max-wrt-among \( R(i) \) (rsd-winners \( R \) alts agents'))
        - By (intro indiff-set-Max-wrt-among[OF finite] rsd-winners-wf) simp-all
  - **Show** ?case
    - **Proof** (cases \( j = i \))
      - Assume \( j \) [simp]: \( j = i \)
        - From indiff-set Cons have RSD-set-rel \( R(i) \) \( (w' \) (j \# agents')) \( (w \) (j \# agents'))
          - Unfolding RSD-set-rel-def
            - By (auto simp: Ri.Max-wrt-among-total-preorder Ri'.Max-wrt-among-total-preorder)
            - Thus ?case ..
next
assume \( j \) \( \text{[simp]} \): \( j \neq i \)
from Cons have \( w'\text{-agents}' = w\text{-agents}' \lor \text{RSD-set-rel} (R i) (w'\text{-agents}') (w\text{-agents}') \) by simp
thus \( \text{fcase} \)
proof
assume rel: \( \text{RSD-set-rel} (R i) (w'\text{-agents}') (w\text{-agents}') \)
hence indiff-set: \( \text{indiff-set} (R i) (w\text{-agents}') \) by (simp add: \( \text{RSD-set-rel-def} \))
moreover from Cons.prems finite \( \langle \text{alts} \neq \{ \} \rangle \)
have \( w\text{-agents}' \subseteq \text{alts} \)
by (intro rsd-winners-af; simp)+
with finite have Max-wrt-among (R j) (w\text{-agents}') \( \neq \{ \} \)
by (intro \( \text{Rj.Max-wrt-among-nonempty} \) auto
ultimately have indiff-set (R i) (w\text{-agents}') (w\text{-agents}') unfolding \( \text{RSD-set-rel-def} \) ..
thus \( \text{case} \ .. \)
qed simp-all
qed

The following variant of the previous theorem is slightly easier to use. We eliminate the case where the two outcomes are the same by observing that the original outcome is then also set-preferred to the manipulated one. In essence, this means that no matter what manipulation is done, the original outcome is always set-preferred to the manipulated one.

**Lemma rsd-winners-manipulation:**
assumes \( \text{af: total-preorder-on alts Ri'} \)
and \( i: i \in \text{agents} \) and \( \text{set agents' = agents finite agents} \)
and finite: \( \text{finite alts and alts \neq \{ \}} \)
defines \( \text{[simp]} : w' \equiv \text{rsd-winners} (R(i := Ri')) \) alts and \( \text{[simp]} : w \equiv \text{rsd-winners} \)
\( R \) alts
shows \( \forall x \in w' \text{-agents}', \forall y \in w\text{-agents}' \). \( x \preceq[R i] y \)
proof –
have \( w'\text{-agents}' = w\text{-agents}' \lor \text{RSD-set-rel} (R i) (w'\text{-agents}') (w\text{-agents}') \)
using \( \text{rsd-winners-manipulation-aux}[\text{OF assms(1 - 2) - assms(4 - 6)] assms(3)} \)
by simp
thus \( \text{thesis} \)
proof
assume eq: \( w'\text{-agents}' = w\text{-agents}' \)
from assms have \( \text{RSD-pareto-eqclass (set agents') alts R (w agents')} \) unfolding \( w\text{-def} \)
by (intro \( \text{RSD-pareto-eqclass-rsd-winners-aux} \) simp-all
from \( \text{RSD-pareto-eqclass-indiff-set[OF this, of i'] i eq assms(3)} \) show \( \text{thesis} \)
by auto
93
The lottery that RSD yields is well-defined.

**Lemma random-serial-dictatorship-support:**

**Assumes** finite agents finite alts agents' ⊆ agents alts' ≠ {} alts' ⊆ alts

**Shows** set-pmf (random-serial-dictatorship agents' alts' R) ⊆ alts'

**Proof** –

- from assms have [simp]: finite agents' by (auto intro: finite-subset)
- have A: set-pmf (pmf-of-set (rsd-winners R alts' agents'')) ⊆ alts'
  - if agents'' ∈ permutations-of-set agents' for agents''
  - using that assms rsd-winners-wf[where alts' = alts' and agents' = agents'']
  - by (auto simp: permutations-of-set-def)
- from assms show ?thesis
  - by (auto dest!: A simp add: random-serial-dictatorship-altdef)

**QED**

Permutation of alternatives commutes with RSD winners.

**Lemma rsd-winners-permute-profile:**

**Assumes** perm: σ permutes alts and set agents' ⊆ agents

**Shows** rsd-winners (permute-profile σ R) alts agents' = σ' rsd-winners R alts agents'

**Using** (set agents' ⊆ agents)

**Proof** (induction agents')

- case Nil
  - from perm show ?case by (simp add: permutes-image)
- next
  - case (Cons i agents')
    - from wf Cons interpret total-preorder-on alts R i by simp
    - from perm Cons show ?case
      - by (simp add: permute-profile-map-relation Max-wrt-among-map-relation-bij permutes-bij)

**QED**

**Lemma random-serial-dictatorship-singleton:**

**Assumes** finite agents finite alts agents' ⊆ agents x ∈ alts

**Shows** random-serial-dictatorship agents' {x} R = return-pmf x (is ?d = -)

**Proof** –

- from assms have set-pmf ?d ⊆ {x}
  - by (intro random-serial-dictatorship-support) simp-all
- thus ?thesis by (simp add: set-pmf-subset-singleton)

**QED**

end
8.2 Proofs of properties

With all the facts that we have proven about the RSD winners, the hard
work is mostly done. We can now simply fix some arbitrary order of the
agents, apply the theorems about the RSD winners, and show the properties
we want to show without doing much reasoning about probabilities.

context election
begin

abbreviation RSD ≡ random-serial-dictatorship agents alts

8.2.1 Well-definedness

sublocale RSD: social-decision-scheme agents alts RSD
using pref-profile-wf, random-serial-dictatorship-support[of agents alts]
by unfold-locales (simp-all add: lotteries-on-def)

8.2.2 RD extension

lemma RSD-extends-RD:
assumes wf: is-pref-profile R and unique: has-unique-favorites R
shows RSD R = RD R
proof -
from wf interpret pref-profile-wf agents alts R .
from unique interpret pref-profile-unique-favorites by unfold-locales
have RSD R = pmf-of-set agents > >
(λi. random-serial-dictatorship (agents − {i}) (favorites R i) R)
by (simp add: random-serial-dictatorship-altdef Max-wrt-def)
also from assms have ... = pmf-of-set agents > > (λi. return-pmf (favorite R i))
by (intro bind-pmf-cong refl, subst random-serial-dictatorship-singleton [symmetric])
(auto simp: unique-favorites favorite-in-alts)
also from assms have ... = RD R
by (simp add: random-dictatorship-unique-favorites map-pmf-def)
finally show ?thesis .
qed

8.2.3 Anonymity

Anonymity is a direct consequence of the fact that we randomise over all
permutations in a uniform way.

sublocale RSD: anonymous-sds agents alts RSD
proof
fix π R assume perm: π permutes agents and wf: is-pref-profile R
let ′f = λagents'. pmf-of-set (rsd-winners R alts agents')
from perm wf have RSD (R ◦ π) = map-pmf (map π) (pmf-of-set (permutations-of-set
agents)) ≡ ′f
by (simp add: random-serial-dictatorship-altdef bind-map-pmf)
also from \texttt{perm} have \ldots = \texttt{RSD R}

by \texttt{(simp add: map-pmf-of-set-inj permutes-inj-on inj-on-mapI permutations-of-set-image-permutes random-serial-dictatorship-altdef)}

finally show \texttt{RSD (R \circ \pi) = RSD R}.

\texttt{qed}

### 8.2.4 Neutrality

Neutrality follows from the fact that the RSD winners of a permuted profile are simply the image of the original RSD winners under the permutation.

\texttt{sublocale RSD: neutral-sds agents alts RSD}

\texttt{proof}

\texttt{fix} \sigma \texttt{R assume perm: } \sigma \texttt{ permutes alts and wf: } is-pref-profile \texttt{R}

\texttt{from wf interpret pref-profile-wf agents alts R}.

\texttt{from perm show RSD (permute-profile } \sigma \texttt{ R) = map-pmf } \sigma \texttt{(RSD R)}


\texttt{qed}

### 8.2.5 Ex-post efficiency

Ex-post efficiency follows from the fact that the set of RSD winners is a Pareto-equivalence class.

\texttt{sublocale RSD: ex-post-efficient-sds agents alts RSD}

\texttt{proof}

\texttt{fix} \texttt{R assume wf: } is-pref-profile \texttt{R}

\texttt{then interpret pref-profile-wf agents alts R}.

\texttt{\{ fix x assume x: } x \texttt{ \in set-pmf (RSD R) x \in pareto-losers R}

\texttt{from x(2) obtain y where [simp]: y \in alts and pareto: y \succ [Pareto(R)] x}

\texttt{by (cases rule: pareto-losersE)}

\texttt{from x have [simp]: x \in alts using pareto-loser-in-alts by simp}

\texttt{from x(1) obtain agents’ where agents’: set agents’ = agents and}

\texttt{x \in set-pmf (pmf-of-set (rsd-winners R alts agents’))}

\texttt{by (auto simp: random-serial-dictatorship-altdef dest!: permutations-of-setD)}

\texttt{with wf have x’: x \in rsd-winners R alts agents’}

\texttt{using rsd-winners-uf [where alts’ = alts and agents’ = agents’]}

\texttt{by (subst (asm) set-pmf-of-set) (auto simp: permutations-of-setD)}

\texttt{from wf agents’}

\texttt{have RSD-pareto-eqclass agents alts R (rsd-winners R alts agents’)}

\texttt{by (intro RSD-pareto-eqclass-rsd-winners) simp-all}

\texttt{hence winner-iff: y \in rsd-winners R alts agents’ \longleftrightarrow (\forall i \in agents. x \preceq [R i] y)}

\texttt{if x \in rsd-winners R alts agents’ y \in alts for x y}

\texttt{using that unfolding RSD-pareto-eqclass-def by blast}

\texttt{from x’ pareto winner-iff[of x y] winner-iff[of y x] have False}
by (force simp: strongly-preferred-def Pareto-iff)
}

thus set-pmf (RSD R) ∩ pareto-losers R = {} by blast
qed

8.2.6 Strong strategy-proofness

Strong strategy-proofness is slightly more difficult to show. We have already shown that the set of RSD winners for the truthful profile is always set-preferred (by the manipulating agent) to the RSD winners for the manipulated profile. This can now be used to show strategy-proofness: We recall that the set of RSD winners is always an indifference class. Therefore, given any fixed alternative \( x \) and considering a fixed order of the agents, either all of the RSD winners in the original profile are at least as good as \( x \) or none of them are, and, since the original RSD winners are set-preferred to the manipulated ones, none of the RSD winners in the manipulated case are at least as good than \( x \) either in that case. This means that for a fixed order of agents, either the probability that the original outcome is at least as good as \( x \) is 1 or the probability that the manipulated outcome is at least as good as \( x \) is 0. Therefore, the original lottery is clearly SD-preferred to the manipulated one.

**sublocale** RSD: strongly-strategyproof-sds agents alts RSD

**proof** (unfold-locales, rule)

fix \( R \) \( i \) \( Ri' \) \( x \)
assume \( wf \): is-pref-profile \( R \) and \( i \) [simp]: \( i \in \) agents and \( x \in \) alts and
\( wf' \): total-preorder-on alts \( Ri' \)
interpret \( R \): pref-profile-uf agents alts \( R \) by fact
define \( R' \) where \( R' \) = \( R \) (\( i := Ri' \))
from \( wf \) \( wf' \) have is-pref-profile \( R' \) by (simp add: \( R' \)-def \( R \).wf-update)

then interpret \( R' \): pref-profile-uf agents alts \( R' \).

**note** \( wf = wf \) \( w f' \)

let \( ?A = \) preferred-alts \( (R \ i) \) \( x \)

from \( wf \) interpret \( Ri \): total-preorder-on alts \( R \) \( i \) by simp

\{ fix agents' assume agents': agents' ∈ permutations-of-set agents
from agents' have [simp]: set agents' = agents
by (simp add: permutations-of-set-def)

let \( ?W = \) rsd-winners \( R \) alts agents' and \( ?W' = \) rsd-winners \( R' \) alts agents'

\( \) \( \) have indiff-set: RSD-pareto-eqclass agents alts \( R \) \( ?W \)
\( \) \( \) by (rule R.RSD-pareto-eqclass-rsd-winners; simp add: \( w f' \)+)
from \( R \).rsd-winners-uf \( R' \).rsd-winners-uf
气息 winners: \( ?W \subseteq \) alts \( ?W \neq {} \) finite \( ?W \) \( ?W' \subseteq \) alts \( ?W' \neq {} \) finite
\( ?W' \)
by simp-all

97
from \(\{\not=\}\) obtain \(y\) where \(y : y \in \{\}\) by blast

with winners have [simp]: \(y : y \in \) alts by blast

from \(\text{wf}'\) i have mono: \(\forall x \in \{\}\). \(\forall y \in \{\}. R i x y\) unfolding \(R'\)-def by (intro \(R\).rsd-winners-manipulation) simp-all

have \(\text{lottery-prob} (\text{pmf-of-set} \{\}) \geq \) \(\text{lottery-prob} (\text{pmf-of-set} \{\}) \geq \) \(\text{lottery-prob} (\text{pmf-of-set} \{\}) \geq \)

proof (cases \(y \geq [R i] x\))

next

qed

hence \(\text{emeasure} (\text{measure-pmf} (\text{pmf-of-set} \{\})) \geq \) \(\text{emeasure} (\text{measure-pmf} (\text{pmf-of-set} \{\})) \geq \) \(\text{emeasure} (\text{measure-pmf} (\text{pmf-of-set} \{\})) \geq \)

by (simp add: measure-pmf.emeasure-eq-measure)

qed
imports
  Complex-Main
  ../Elections
keywords
  preference-profile :: thy-goal
begin
ML-file (preference-profiles.ML)

context election
begin

lemma preferred-alts-prefs-from-table:
  assumes prefs-from-table-wf agents alts xs i ∈ set (map fst xs)
  shows preferred-alts (prefs-from-table xs i) x =
    of-weak-ranking-Collect-ge (rev (the (map-of xs i))) x
proof −
  interpret pref-profile-wf agents alts prefs-from-table xs
  by (intro pref-profile-from-tableI assms)
  from assms have [simp]: i ∈ agents by (auto simp: prefs-from-table-wf-def)
  have of-weak-ranking-Collect-ge (rev (the (map-of xs i))) x =
    Collect (of-weak-ranking (the (map-of xs i)))
  by (rule eval-Collect-of-weak-ranking [symmetric])
  also from assms have the (map-of xs i) ∈ set (map snd xs)
  by (cases map-of xs i) (force simp: map-of-eq-None-iff dest: map-of-None)
  also from assms have of-weak-ranking (the (map-of xs i)) =
    prefs-from-table xs i
  by (subst prefs-from-table-map-of [OF assms(1)])
  (auto simp: prefs-from-table-wf-def)
finally show ?thesis by (simp add: of-weak-ranking-Collect-ge-def preferred-alts-altdef)
qed

lemma favorites-prefs-from-table:
  assumes wf: prefs-from-table-wf agents alts xs and i: i ∈ agents
  shows favorites (prefs-from-table xs) i = hd (the (map-of xs i))
proof (cases map-of xs i)
  case None
  with assms show ?thesis
  by (auto simp: map-of-eq-None-iff prefs-from-table-wf-def)
next
  case (Some y)
  with assms have is-finite-weak-ranking y y ≠ []
  by (auto simp: prefs-from-table-wf-def)
  with Some show ?thesis

99

qed

lemma has-unique-favorites-prefs-from-table:
  assumes wf: prefs-from-table_wf agents alts xs
  shows has-unique-favorites (prefs-from-table xs) = list-all (λz. is-singleton (hd (snd z))) xs

proof
  interpret pref-profile-wf agents alts prefs-from-table xs
  by (intro pref-profile-from-tableI assms)
  from wf have agents = set (map fst xs) distinct (map fst xs)
  by (auto simp: prefs-from-table-wf-def)
  thus ?thesis unfolding has-unique-favorites-altdef using assms by (auto simp: favorites-prefs-from-table list-all-iff)
qed

end

9.1 Automatic definition of preference profiles from tables

function favorites-prefs-from-table where
  i = j ⇒ favorites-prefs-from-table ((j,x)#xs) i = hd x
| i ≠ j ⇒ favorites-prefs-from-table ((j,x)#xs) i = favorites-prefs-from-table xs i
| favorites-prefs-from-table [] i = {}
by (metis list.exhaust old.prod.exhaust)
termination by lexicographic-order

lemma (in election) eval-favorites-prefs-from-table:
  assumes prefs-from-table-wf agents alts xs
  shows favorites-prefs-from-table xs i = favorites (prefs-from-table xs) i
proof (cases i ∈ agents)
  assume i: i ∈ agents
  with assms have favorites (prefs-from-table xs) i = hd (the (map-of xs i))
  by (simp add: favorites-prefs-from-table)
  also from assms i have i ∈ set (map fst xs)
  by (auto simp: prefs-from-table-wf-def)
  hence hd (the (map-of xs i)) = favorites-prefs-from-table xs i
  by (induction xs i rule: favorites-prefs-from-table.induct) simp-all
finally show ?thesis ..
next
  assume i: i /∈ agents
  with assms have i': i /∈ set (map fst xs)
  by (simp add: prefs-from-table-wf-def)
  hence map-of xs i = None
begin

by (simp add: map-of-eq-None-iff)

hence prefs-from-table xs i = (λ _. False)
  by (intro ext) (auto simp: prefs-from-table-def)

hence favorites (prefs-from-table xs) i = {}
  by (simp add: favorites-def Max-wrt-altdef)

also from i' have \ldots = favorites-prefs-from-table xs i
  by (induction xs i rule: favorites-prefs-from-table.induct) simp-all

finally show \_thesis \_.

qed

function weak-ranking-prefs-from-table where
i ≠ j =⇒ weak-ranking-prefs-from-table ((i,x)#xs) j = weak-ranking-prefs-from-table xs j
| i = j =⇒ weak-ranking-prefs-from-table ((i,x)#xs) j = x
| weak-ranking-prefs-from-table {} j = {}
  by (metis list.exhaust old.prod.exhaust)

termination by lexicographic-order

lemma eval-weak-ranking-prefs-from-table:
  assumes prefs-from-table-wf agents alts xs
  shows weak-ranking-prefs-from-table xs i = weak-ranking (prefs-from-table xs i)
proof (cases i ∈ agents)
  assume i: i ∈ agents
  with assms have weak-ranking (prefs-from-table xs i) = the (map-of xs i)
    by (auto simp: prefs-from-table-def prefs-from-table-wf-def weak-ranking-of-weak-ranking
        split: option.splits)
  also from assms have i ∈ set (map fst xs)
    by (auto simp: prefs-from-table-wf-def)
  hence the (map-of xs i) = weak-ranking-prefs-from-table xs i
    by (induction xs i rule: weak-ranking-prefs-from-table.induct) simp-all

finally show \_thesis \_.

next
  assume i: i /∈ agents
  with assms have i': i /∈ set (map fst xs)
    by (simp add: prefs-from-table-wf-def)
  hence map-of xs i = None
    by (simp add: map-of-eq-None-iff)
  hence prefs-from-table xs i = (λ _. False)
    by (intro ext) (auto simp: prefs-from-table-def)
  hence weak-ranking (prefs-from-table xs i) = [] by simp
  also from i' have \ldots = weak-ranking-prefs-from-table xs i
    by (induction xs i rule: weak-ranking-prefs-from-table.induct) simp-all

finally show \_thesis \_.

qed

lemma eval-prefs-from-table-aux:
  assumes R ≡ prefs-from-table xs prefs-from-table-wf agents alts xs
  shows R i a b ⇔ prefs-from-table xs i a b

end
a ≺ [R i] b ←→ prefs-from-table xs i a b ∧ ¬ prefs-from-table xs i b a
anonymous-profile R = mset (map snd xs)
election agents alts = i ∈ set (map fst xs) =
preferred-alts (R i) x =
of-weak-ranking-Collect-ge (rev (the (map-of xs i))) x
election agents alts = i ∈ set (map fst xs) =
favorites R i = favorites-prefs-from-table xs i
election agents alts = i ∈ set (map fst xs) =
weak-ranking (R i) = weak-ranking-prefs-from-table xs i
election agents alts = i ∈ set (map fst xs) =
favorite R i = the-elem (favorites-prefs-from-table xs i)
election agents alts =
has-unique-favorites R = list-all (λz. is-singleton (hd (snd z))) xs

using assms prefs-from-table-wfD[OF assms(2)]
by (simp-all add: strongly-preferred-def favorite-def anonymise-prefs-from-table
election.preferred-alts-prefs-from-table election.eval-favorites-prefs-from-table
election.has-unique-favorites-prefs-from-table eval-weak-ranking-prefs-from-table)

lemma pref-profile-from-tableI':
assumes R1 ≡ prefs-from-table xss prefs-from-table-wf agents alts xss
shows prefs-profile-wf agents alts xss
using assms by (simp add: pref-profile-from-tableI)

ML ⟨
signature PREFERENCE-PROFILES-CMD =
sig
type info
val preference-profile :
  (term * term) * ((binding * (term * term list list) list) list) −> Proof.context
  −> Proof.state
val preference-profile-cmd :
  (string * string) * ((binding * (string * string list list) list) list) −> Proof.context
  −> Proof.state
val get-info : term −> Proof.context −> info
val add-info : term −> info −> Context.generic −> Context.generic
val transform-info : info −> morphism −> info
end
structure Preference-Profiles-Cmd : PREFERENCE-PROFILES-CMD =
struct
open Preference-Profiles

102
type info =
  raw : (term * term list list) list, eval-thms : thm list }

fun transform-info ({ term = t, binding, def-thm, wf-thm, wf-raw-thm, raw, eval-thms } : info) phi =
  let
    val thm = Morphism.thm phi
    val fact = Morphism.fact phi
    val term = Morphism.term phi
    val bdg = Morphism.binding phi
  in
    { term = term t, binding = bdg binding, def-thm = thm def-thm, wf-thm =
      thm wf-thm,
      wf-raw-thm = thm wf-raw-thm, raw = map (fn (a, bss) => (term a, map
        (map term) bss)) raw,
      eval-thms = fact eval-thms } in
end

structure Data = Generic-Data
(
  type T = (term * info) Item-Net.T
  val empty = Item-Net.init (op aconv o apply2 fst) (single o fst)
  val extend = I
  val merge = Item-Net.merge ) :

fun get-info term lthy =
  Item-Net.retrieve (Data.get (Context.Proof lthy)) term |> the-single |> snd

fun add-info term info lthy =
  Data.map (Item-Net.update (term, info)) lthy

fun add-infos infos lthy =
  Data.map (fold Item-Net.update infos) lthy

fun preference-profile-aux agents alts (binding, args) lthy =
  let
    val dest-Type' = Term.dest-Type #> snd #> hd
    val (agentT, altT) = apply2 (dest-Type' o fastype-of) (agents, alts)
    val alt-setT = HOLogic.mk-setT altT
    fun define t =
      Local-Theory.define (((binding, NoSyn),
        ((Binding.suffix-name -def binding, @{attributes [code]}), t)) lthy
    val ty = HOLogic.mk-prodT (agentT, HOLogic.listT (HOLogic.mk-setT altT))
    val args' =
      args |> map (fn x => x ||> map (HOLogic.mk-set altT)) ||> HOLogic.mk-list
alt-setT)
  val t-raw =
    args' |
    > map HOLogic.mk-prod
    > HOLogic.mk-list ty
val t = Const (@{const-name prefs-from-table},
  HOLogic.listT ty -->> pref-profileT agentT altT) $ t-raw
val ((prefs, prefs-def), lthy) = define t
val prefs-from-table-wf-const =
  Const (@{const-name prefs-from-table-wf},
     HOLogic.mk-setT agentT -->
     HOLogic.listT (HOLogic.mk-prodT (agentT, HOLogic.listT (HOLogic.mk-setT altT)))) -->>
     HOLogic.boolT
val wf-prop = (prefs-from-table-wf-const $ agents $ alts $ t-raw) |> HOLogic.mk-Trueprop
in
  ((prefs, wf-prop, prefs-def), lthy)
end

fun fold-accum f xs s =
let
  fun fold-accum-aux - [] s acc = (rev acc, s)
  | fold-accum-aux f (x::xs) s acc =
    case f x s of (y, s') => fold-accum-aux f xs s' (y::acc)
in
  fold-accum-aux f xs s []
end

fun preference-profile ((agents, alts), args) lthy =
let
  fun qualify pref suff = Binding.qualify true (Binding.name-of pref) (Binding.name suff)
val (results, lthy) = fold-accum (preference-profile-aux agents alts) args lthy
val prefs-terms = map #1 results
val wf-props = map #2 results
val defs = (map (snd o #3)) results
val raws = map snd args
val bindings = map fst args
fun tac lthy =
  let
    val lthy' = put-simpset HOL-ss lthy addsimps
    @{thms list.set Union-insert Un-insert-left insert-not-empty Int-empty-left
    Int-empty-right
    insert-commute Un-empty-left Un-empty-right insert-absorb2 Union-empty
    is-weak-ranking-Cons is-weak-ranking-Nil finite-insert finite.emptyI
    Set.singleton-iff Set.empty-iff Set.ball-simps}
  in
    tac lthy'
  end

in
  ((prefs, wf-prop, prefs-def), lthy)
end
fun after-qed [wf-thms-raw] lthy =
  let
  fun prep-thms attrs suffix (thms : thm list) binding =
    ((#qualify binding suffix, attrs), [(thms, [])])
  fun prep-thmss simp suffix thmss = map2 (prep-thms simp suffix) thmss

  val mk-infos =
    let
      fun aux acc
        (bdg :: bdgs) (t :: ts) (r :: raws) (def :: def-thms) (wf :: wf-thms)
        (wf-raw :: wf-raw-thms) (evals :: eval-thmss) =
         aux ((t, #binding = bdg, term = t, raw = r, def-thm = def, wf-thm =
        wf,  
        #wf-raw-thm = wf-raw, eval-thms = evals)) :: acc)
      bdgs ts raws def-thms wf-thms wf-raw-thms eval-thmss
      | aux acc [] - - - - - = (acc : (term * info) list)
      | aux - - - - - - - = raise Match
    in
      aux []
    end
  val infos = mk-infos bindings prefs-terms raws def-thms wf-thms-raw
  eval-thmss
  in
    lthy
    |> note wf-thms-raw wf-raw []
    |> note wf-thms wf @{attributes [simp]}
    |> notes eval-thmss eval []
Local-Theory.declaration { syntax = false, pervasive = false } (fn m => add-infos (map (fn (t,i) => (Morphism.term m t, transform-info i m)) infos))
end
| after-qed - - = raise Match

Proof theorem NONE after-qed [map (fn prop => (prop, [])) wf-props] lthy |
| Proof refine-singleton (Method.Basic (SIMPLE-METHOD o tac)) end

fun preference-profile-cmd ((agents, alts), argss) lthy =
let
val read = Syntax.read-term lthy
fun read′ ty t = Syntax.parse-term lthy t |> Type.constraint ty |> Syntax.check-term lthy
val agents′ = read agents
val alts′ = read alts
val agentT = agents′ |> fastype-of |> dest-Type |> snd |> hd
val altT = alts′ |> fastype-of |> dest-Type |> snd |> hd
fun read-pref-elem ts = map (read′ altT) ts
fun read-prefs prefs = map read-pref-elem prefs
fun prep (binding, args) =
  (binding, map (fn (agent, prefs) => (read′ agentT agent, read-prefs prefs)) args)
in
preference-profile ((agents′, alts′), map prep argss) lthy end

val parse-prefs =
let
val parse-pref-elem =
  (Args.bracks (Parse.list1 Parse.term)) ||
  Parse.term >> single
in
Parse.list1 parse-pref-elem
end

val parse-pref-profile =
  Parse.binding ++ | Args.$$ = -- Scan.repeat1 (Parse.term -- | Args.colon -- parse-prefs)
val - =
  Outer-Syntax.local-theory-to-proof @{command-keyword preference-profile}
  construct preference profiles from a table
  (Args.$$ agents |-- Args.colon |-- Parse.term --| Args.$$ alts --|
  Args.colon
  |-- Parse.term --| Args.$$ where --
  Parse.and-list1 parse-pref-profile >> preference-profile-cmd);
theory QSOpt-Exact
imports Complex-Main
begin

ML ⟨

signature RAT-UTILS = 
sig
  val rat-to-string : Rat.rat -> string
  val pretty-rat : Rat.rat -> string
  val string-to-rat : string -> Rat.rat option
  val mk-rat-number : typ -> Rat.rat -> term
  val dest-rat-number : term -> Rat.rat
end

structure Rat-Utils : RAT-UTILS =
struct

fun rat-to-string r =
  case Rat.dest r of
    (a, 1) => Int.toString a
    | (a, b) => (if a < 0 then ~ else ) ^ Int.toString (abs a) ^ / ^ Int.toString b

fun pretty-rat r =
  case Rat.dest r of
    (a, 1) => (if a < 0 then ~ else ) ^ Int.toString a
    | (a, b) => (if a < 0 then ~ else ) ^ Int.toString (abs a) ^ / ^ Int.toString b

fun string-to-rat s =
  let
    val (s1, s2') = s |> Substring.full |> Substring.splitl (fn x => x <> #)
    val (s1, s2) = (s1, s2') |> apsnd (Substring.triml 1) |> apply2 Substring.string in
      if Substring.isEmpty s2' then
        Option.map Rat.of-int (Int.fromString s1)
      else
        Option.mapPartial (fn x => Option.map (fn y => Rat.make (x, y))
         (Int.fromString s2)) (Int.fromString s1)
  end

fun dest-num x =
  case x of
end
⟩

end
end
Const (@{const-name Code-Numeral.int-of-integer}, -) $ x => dest-num x
| - => HOLogic.dest-number x

fun dest-rat-number t =
case t of
  (Const (@{const-name Rings.divide-class.divide},-)) $ a $ b
  => Rat.make (snd (dest-num a), snd (dest-num b))
| (Const (@{const-name Groups.uminus-class.uminus},-)) $ a
  => ~ (dest-rat-number a)
| (Const (@{const-name Rat.field-char-0-class.of-rat},-)) $ a => dest-rat-number a
| (Const (@{const-name Rat.Frct},-) $ (Const (@{const-name Product-Type.Pair},
-)) $ a $ b))
  => Rat.make (snd (dest-num a), snd (dest-num b))
| - => Rat.of-int (snd (dest-num t));

fun mk-rat-number ty r =
case Rat.dest r of
  (a, 1) => HOLogic.mk-number ty a
| (a, b) =>
  Const (@{const-name Rings.divide-class.divide}, ty --- ty --- ty) $
  HOLogic.mk-number ty a $ HOLogic.mk-number ty b
end

ML ⟨

signature LP-PARAMS =
sig
type T
val print : T => string
val read : string => T option
val compare : (T * T) => General.order
val negate : T => T
val from-int : int => T

end;

signature LINEAR-PROGRAM-COMMON =
sig
  exception QSOpt-Parse
  datatype 'a infty = Finite of 'a | Pos-Infty | Neg-Infty;
  datatype comparison = LEQ | EQ | GEQ
  datatype optimization-mode = MAXIMIZE | MINIMIZE
  datatype 'a result = Optimal of 'a * (string * 'a) list | Unbounded | Infeasible | Unknown

108
type var = string

val is-finite : 'a infty -> bool
val map-infty : ('a -> 'b) -> 'a infty -> 'b infty
val print-infty : (string -> 'a) -> 'a infty -> string
val print-comparison : comparison -> string
val print-optimization-mode : optimization-mode -> string
val gen-print-bound : (string -> 'a) -> 'a bound -> string
val gen-print-linterm : (string -> 'a linterm) -> 'a linterm -> string
val gen-print-constraint : (string -> 'a constraint) -> 'a constraint -> string
val gen-print-program : (string -> 'a prog) -> 'a prog -> string
val gen-read-result : (string -> 'a option) -> string -> 'a result

signature LINEAR-PROGRAM =
  sig
    include LINEAR-PROGRAM-COMMON
    type T
    val print-bound : T bound -> string
    val print-linterm : T linterm -> string
    val print-constraint : T constraint -> string
    val print-program : T prog -> string
    val save-program : string -> T prog -> unit
    val solve-program : T prog -> T result
    val read-result : string -> T result
    val read-result-file : string -> T result
  end;

structure Linear-Program-Common : LINEAR-PROGRAM-COMMON =
  struct
    exception QSOpt-Parse
  end;

109
datatype \( 'a \text{ infty} = \text{Finite of } 'a \mid \text{Pos-Infty} \mid \text{Neg-Infty} \)

datatype comparison = LEQ | EQ | GEQ

datatype optimization-mode = MAXIMIZE | MINIMIZE

datatype \( 'a \text{ result} = \text{Optimal of } 'a \ast (\text{string } \ast 'a) \text{ list } \mid \text{Unbounded} \mid \text{Infeasible} \mid \text{Unknown} \)

type var = string

type \( 'a \text{ bound} = 'a \text{ infty } \ast \text{var } \ast 'a \text{ infty} \)

type \( 'a \text{ linterm} = ('a \ast \text{var}) \text{ list} \)

type \( 'a \text{ constraint} = 'a \text{ linterm } \ast \text{comparison } \ast 'a \)

type \( 'a \text{ prog} = \text{optimization-mode } \ast 'a \text{ linterm } \ast 'a \text{ constraint list } \ast 'a \text{ bound list} \)

fun is-finite (Finite -) = true
| is-finite - = false

fun map-infty f (Finite x) = Finite (f x)
| map-infty - Pos-Infty = Pos-Infty
| map-infty - Neg-Infty = Neg-Infty

fun print-infty - Neg-Infty = - INF
| print-infty - Pos-Infty = INF
| print-infty f (Finite x) = f x

fun print-comparison LEQ = <
| print-comparison EQ = =
| print-comparison GEQ = >=

fun print-optimization-mode MINIMIZE = MINIMIZE
| print-optimization-mode MAXIMIZE = MAXIMIZE

fun gen-print-bound - (Neg-Infty, v, Pos-Infty) = v ˆ free
| gen-print-bound f (Neg-Infty, v, u) = v ˆ <= ˆ print-infty f u
| gen-print-bound f (l, v, Pos-Infty) = print-infty f l ˆ <= ˆ v
| gen-print-bound f (l, v, u) = print-infty f l ˆ <= ˆ v ˆ <= ˆ print-infty f u

fun gen-print-summand (cmp, from-int, print, negate) first c v = let
val neg = (cmp (c, from-int 0) = LESS)
fun eq x = (cmp (c, x) = EQUAL)
val one = eq (from-int 1)
val mone = eq (from-int (~1))
val c' = if first andalso one then
    else if first andalso mone then −
    else if first then print c ˆ
    else if mone then −
    else if one then +
    else if neg then − ˆ print (negate c) ˆ
    else + ˆ print c ˆ
fun gen-print-linterm ops t =
  let
    val n = length t
    val print-summand = gen-print-summand ops
    fun go (c, v) (i, acc) = (i+1, print-summand (i = n) c v ^ acc)
  in
    snd (fold go (rev t) (1, ))
  end

fun gen-print-constraint (ops as (-, -, print, -)) (lhs, cmp, rhs) =
  gen-print-linterm ops lhs ^ print-comparison cmp ^ print rhs

fun gen-print-program (ops as (-, -, print, -)) (mode, obj, constrs, bnds) =
  let
    val padding = replicate-string 4
    fun mk-block s f xs = (s :: map (prefix padding o f) xs)
    fun mk-block' s f xs = if null xs then [] else mk-block s f xs
    val lines =
      mk-block (print-optimization-mode mode) (gen-print-linterm ops) [obj] @
      mk-block' ST (gen-print-constraint ops) constrs @
      mk-block' BOUNDS (gen-print-bound print) bnds @ [END, ]
  in
    cat-lines lines
  end

exception QSOpt-Parse

fun read-status x =
  if String.isPrefix status x andalso not (String.isPrefix status = x) then
    let
      val statuses = [OPTIMAL, INFEASIBLE, UNBOUNDED]
    in
      case find-first (fn s => String.isPrefix (status ^ s) x) statuses of
        NONE => SOME UNKNOWN |
        SOME y => SOME y
    end
  else
    NONE

fun apply - - [] = NONE
  | apply abort f (x :: xs) =
      if abort x then
        NONE
else case f x of
  NONE => apply abort f xs
  | SOME y => SOME (y, xs)

fun apply-repeat abort (f : string -> 'a option) : string list -> 'a list * string list
  = let
    fun go acc xs =
      case apply abort f xs of
        NONE => (rev acc, xs)
      | SOME (y, xs) => go (y :: acc) xs
    in go [] end

fun the-apply f xs =
  case apply (K false) f xs of
    NONE => raise QSOpt-Parse
  | SOME y => y

fun apply-unit p xs =
  case apply (not o p) (K (SOME ())) xs of
    NONE => raise QSOpt-Parse
  | SOME (_, xs) => xs

fun gen-read-value read x =
  let
    val x = unprefix Value = x
  in
    read x
  end
  handle Fail => NONE

val trim =
  let
    fun chop [] = []
    | chop (l as (x::xs)) = if Char.isSpace x then chop xs else l
  in
    String.implode o chop o rev o chop o rev o String.explode
  end

fun gen-read-assignment read x : (string * 'a) option =
  x |> try (Substring.full
    |> Substring.split (fn x => x <> #=)
    |> apply2 Substring.string
    |> apsnd (unprefix =)
    |> apply2 trim
    |> Option.mapPartial (fn (x,y) => Option.map (fn y => (x,y)) (read y)))
fun gen-read-result read s = 
  let
    val s = s |> split-lines |> map trim
    val (status, s) = the-apply read-status s
    val (result, _) =
      if status = OPTIMAL then
        let
          val (value, s) = the-apply (gen-read-value read) s
          val s = apply-unit (fn x => x = VARS:) s
          val (vars, s) = apply-repeat (String.isSuffix :) (gen-read-assignment read)
        in
          (Optimal (value, vars), s)
        end
      else if status = INFEASIBLE then
        (Infeasible, s)
      else if status = UNBOUNDED then
        (Unbounded, s)
      else
        (Unknown, s)
    in
    result
  end
end;

functor Linear-Program(LP-Params : LP-PARAMS) : LINEAR-PROGRAM =
struct
  open Linear-Program-Common;

  local
    open LP-Params;
    val ops = (compare, from-int, print, negate)
    in
      type T = T
      val print-bound = gen-print-bound print
      val print-linterm = gen-print-linterm ops
      val print-constraint = gen-print-constraint ops
      val print-program = gen-print-program ops
    end

  fun save-program filename prog =
    let
      val output = print-program prog
      val f = TextIO.openOut filename
      val _ = TextIO.output (f, output)
    in
      ...
    end
end
val _ = TextIO.closeOut f

fun wrap s = \\
  s

val read-result = gen-read-result LP-Params.read

fun read-result-file filename = let
  val f = TextIO.openIn filename
  val s = TextIO.input f
  val _ = TextIO.closeIn f
  in
  read-result s
  end

fun solve-program prog = let
  val name = string-of-int (Time.toMicroseconds (Time.now ()))
  val lpname = Path.implode (Path.expand (Isabelle-System.create-tmp-path name .lp))
  val resultname = Path.implode (Path.expand (Isabelle-System.create-tmp-path name .sol))
  val _ = save-program lpname prog
  val esolver-path = getenv QSOPT-EXACT-PATH
  val esolver = if esolver-path = then esolver else esolver-path
  val command = wrap esolver "-O" wrap resultname "-" wrap lpname
  val {err = err, rc = rc, ...} = Bash.process command
  in
  if rc <> 0 then
    raise Fail (QSopt-exact returned with an error (return code " Int.toString rc ");\n    n " err)
  else
    let
      val result = read-result-file resultname
      val _ = OS.FileSys.remove lpname
      val _ = OS.FileSys.remove resultname
      in
      result
      end
  end
end

structure Rat-Linear-Program = Linear-Program( struct

114
type $T = \text{Rat.rat}$

val print = Rat-Utils.rat-to-string
val read = Rat-Utils.string-to-rat
val compare = Rat.ord
val from-int = Rat.of-int
val negate = Rat.neg

end)

end

10 Automatic Fact Gathering for Social Decision Schemes

theory SDS-Automation
  imports
    Preference-Profile-Cmd
    QSOpt-Exact
    ../Social-Decision-Schemes
keywords
derive-orbit-equations
derive-support-conditions
derive-ex-post-conditions
find-inefficient-supports
prove-inefficient-supports
derive-strategyproofness-conditions :: thy-goal
begin

We now provide the following commands to automatically derive restrictions on the results of Social Decision Schemes satisfying Anonymity, Neutrality, Efficiency, or Strategy-Proofness:

derive-orbit-equations to derive equalities arising from automorphisms of the given profiles due to Anonymity and Neutrality

derive-ex-post-conditions to find all Pareto losers and the given profiles and derive the facts that they must be assigned probability 0 by any ex-post-efficient SDS

find-inefficient-supports to use Linear Programming to find all minimal SD-inefficient (but not ex-post-inefficient) supports in the given profiles and output a corresponding witness lottery for each of them
prove-inefficient-supports to prove a specified set of support conditions arising from \textit{ex-post-} or SD-Efficiency. For conditions arising from SD-Efficiency, a witness lottery must be specified (e.g. as computed by \texttt{derive-orbit-equations}).

derive-support-conditions to automatically find and prove all support conditions arising from \textit{ex-post-} and SD-Efficiency.

derive-strategyproofness-conditions to automatically derive all conditions arising from weak Strategy-Proofness and any manipulations between the given preference profiles. An optional maximum manipulation size can be specified.

All commands except \texttt{find-inefficient-supports} open a proof state and leave behind proof obligations for the user to discharge. This should always be possible using the Simplifier, possibly with a few additional rules, depending on the context.

\textbf{lemma} \texttt{disj-False-right}: \(P \lor \text{False} \iff P\) by simp

\textbf{lemmas} \texttt{multiset-add-ac} = \texttt{add-ac[where ?a = 'a multiset]}

\textbf{lemma} \texttt{less-or-eq-real}: 
\( (x::\text{real}) < y \lor x = y \iff x \leq y \lor y = x \iff x \leq y \) by \texttt{linarith+}

\textbf{lemma} \texttt{multiset-Diff-single-normalize}:
\begin{align*}
\text{fixes} & \; a \; c \\
\text{assumes} & \; a \neq c \\
\text{shows} & \; ((\#a\#) + B) - (\#c\#) = (\#a\#) + (B - (\#c\#)) \\
\text{using} & \; \texttt{assms by auto}
\end{align*}

\textbf{lemma} \texttt{ex-post-efficient-aux}:
\begin{align*}
\text{assumes} & \; \texttt{prefs-from-table-wf agents alts xss R} \equiv \texttt{prefs-from-table xss} \\
\text{assumes} & \; i \in \texttt{agents} \lor i \in \texttt{agents}. \; y \geq \texttt{[prefs-from-table xss i]} \implies \texttt{x \sim y} \leq \texttt{[prefs-from-table xss i]} \\
\text{shows} & \; \texttt{ex-post-efficient-sds agents alts sds} \rightarrow \texttt{pmf (sds R) x = 0} \\
\text{proof} & \; \\
\text{assume} & \; \texttt{ex-post: ex-post-efficient-sds agents alts sds} \\
\text{from} & \; \texttt{assms(1,2) have wf: pref-profile-wf agents alts R} \\
\text{by} & \; \texttt{(simp \ add: pref-profile-from-tableI')} \\
\text{from} & \; \texttt{ex-post interpret ex-post-efficient-sds agents alts sds} \\
\text{from} & \; \texttt{assms(2--) show pmf (sds R) x = 0} \\
\text{by} & \; \texttt{(intro \ ex-post-efficient''[OF \ \texttt{wf}, \ \texttt{of \ i \ y}]) \ simp-all}
\end{align*}
\texttt{qed}

\textbf{lemma} \texttt{SD-inefficient-support-aux}:
\begin{align*}
\text{assumes} & \; R: \texttt{prefs-from-table-wf agents alts xss R} \equiv \texttt{prefs-from-table xss} \\
\text{assumes} & \; \texttt{as: as \neq [] set as \subseteq alts \ distinct as A = set as} \\
\text{assumes} & \; \texttt{ys: \forall x \in \texttt{set (map snd ys)}. \; 0 \leq x \sim \text{sum-list (map snd ys)} = 1 \texttt{ set (map \ \texttt{fst \ y} \ys) \subseteq alts}}
\end{align*}
assumes \( i : i \in \text{agents} \)
assumes \( SD1: \forall i\in\text{agents}. \forall x\in\text{alts}. \)
\[
\sum_{y} (\text{map snd} (\text{filter} (\lambda y. \text{prefs-from-table} \ xss \ i \ x \ (\text{fst} \ y)) \ ys)) \geq \text{real} (\text{length} (\text{filter} (\text{prefs-from-table} \ xss \ i \ x) \ as)) / \text{real} (\text{length} as)
\]
assumes \( SD2: \exists x\in\text{alts}. \sum_{y} (\text{map snd} (\text{filter} (\lambda y. \text{prefs-from-table} \ xss \ i \ x \ (\text{fst} \ y)) \ ys)) > \text{real} (\text{length} (\text{filter} (\text{prefs-from-table} \ xss \ i \ x) \ as)) / \text{real} (\text{length} as)
\)
shows \( \text{sd-efficient-sds} \text{ agents alts sds} \rightarrow (\exists x\in A. \text{pmf} (\text{sds} R) \ x = 0) \)

proof
assume \( \text{sd-efficient-sds} \text{ agents alts sds} \)
from \( R \) have \( uf: \text{pref-profile-uf} \text{ agents alts} R \)
by \( (\text{simp add: pref-profile-from-tableI}) \)
then interpret \( \text{pref-profile-uf} \text{ agents alts} R \).
interpret \( \text{sd-efficient-sds} \text{ agents alts sds} \) by fact
from \( ys \) have \( ys': \text{pmf-of-list-uf} \ ys \) by \( (\text{intro pmf-of-list-ufI}) \) auto

\[
\{ \ 
\begin{align*}
\text{fix} & \ i \ x \ \text{assume} \ x \in \text{alts} \ i \in \text{agents} \\
\text{with} \ & ys' \ \text{have} \ \text{lottery-prob} \ (\text{pmf-of-set} \ ys) \ (\text{preferred-alts} \ (R \ i) \ x) = \\
& \sum_{y} (\text{map snd} (\text{filter} (\lambda y. \text{prefs-from-table} \ xss \ i \ x \ (\text{fst} \ y)) \ ys)) \\
& \text{by} \ (\text{subst measure-pmf-of-list}) \ (\text{simp-all add: preferred-alts-def} R) \\
\} \ \\
\text{note} & A = \text{this} \\
\{ \ 
\begin{align*}
\text{fix} & \ i \ x \ \text{assume} \ x \in \text{alts} \ i \in \text{agents} \\
\text{with} \ & \text{as} \ \text{have} \ \text{lottery-prob} \ (\text{pmf-of-set} \ (\text{set} \ as)) \ (\text{preferred-alts} \ (R \ i) \ x) = \\
& \text{real} (\text{card} (\text{set} \ as \cap \text{preferred-alts} \ (R \ i) \ x)) / \text{real} (\text{card} (\text{set} \ as)) \\
& \text{by} \ (\text{subst measure-pmf-of-set}) \ \text{simp-all} \\
\text{also} & \ \text{have} \ \text{set} \ as \cap \text{preferred-alts} \ (R \ i) \ x = \text{set} \ (\text{filter} (\lambda y. \ R \ i \ x \ y) \ as) \\
& \text{by} \ (\text{auto simp add: preferred-alts-def}) \\
\text{also} & \ \text{have} \ \text{card} \ldots = \text{length} \ (\text{filter} (\lambda y. \ R \ i \ x \ y) \ as) \\
& \text{by} \ (\text{intro distinct-card distinct-filter assms}) \\
\text{also} & \ \text{have} \ \text{card} \ (\text{set} \ as) = \text{length} \ as \ \text{by} \ (\text{intro distinct-card assms}) \\
\text{finally} & \ \text{have} \ \text{lottery-prob} \ (\text{pmf-of-set} \ (\text{set} \ as)) \ (\text{preferred-alts} \ (R \ i) \ x) = \\
& \text{real} (\text{length} (\text{filter} (\text{prefs-from-table} \ xss \ i \ x) \ as)) / \text{real} (\text{length} as) \\
& \text{by} \ (\text{simp add: R}) \\
\} \ \\
\text{note} & B = \text{this} \\
\from \ uf \ \text{show} \exists x\in A. \text{pmf} (\text{sds} R) \ x = 0 \\
\text{proof} \ (\text{rule SD-inefficient-support}') \\
\from \ ys \ ys' \ \text{show} \ \text{lottery1: pmf-of-list} \ ys \in \text{lotteries} \ \text{by} \ (\text{intro pmf-of-list-lottery}) \\
\text{show} \ i: i \in \text{agents by fact} \\
\from \ \text{as} \ \text{have} \ \text{lottery2: pmf-of-set} \ (\text{set} \ as) \in \text{lotteries} \\
\text{by} \ (\text{intro pmf-of-set-lottery}) \ \text{simp-all} \\
\from \ i \ \text{as} \ SD2 \ \text{lottery1} \ \text{lottery2} \ \text{show} \ \neg SD (R \ i) \ (\text{pmf-of-list} \ ys) \ (\text{pmf-of-set} A) \\
\text{by} \ (\text{subst preorder-on_.SD-preorder[alts]}) \ (\text{auto simp: A B not-le}) \\
\from \ \text{as} \ SD1 \ \text{lottery1} \ \text{lottery2} \\
\text{show} \ \forall i\in\text{agents}. SD (R \ i) \ (\text{pmf-of-set} A) \ (\text{pmf-of-list} ys)
### Definition

**Definition pref-classes**

\[
\text{pref-classes alts le} = \text{preferred-alts le ' alts} - \{\text{alts}\}
\]

**Primrec pref-classes-lists**

```
| \text{pref-classes-lists} [[\]] = \{} \\
| \text{pref-classes-lists} (xs\#xss) = \text{insert} (\bigcup (\text{set} (xs\#xss))) (\text{pref-classes-lists} xss)
```

**Function pref-classes-lists-aux**

```
| \text{pref-classes-lists-aux} [[\]] = \{} \\
| \text{pref-classes-lists-aux} \text{acc} (xs\#xss) = \text{insert} \text{acc} (\text{pref-classes-lists-aux} (\text{acc} \cup \text{xss}) xss)
```

### Lemma pref-classes-lists-append

**Lemma pref-classes-lists-append**

\[
\text{pref-classes-lists} (xs \@ ys) = (\bigcup (\text{set} \text{ys})) \ ' \text{pref-classes-lists} \text{xs} \cup \text{pref-classes-lists} \text{ys}
\]

**By** (induction \text{xs}) auto

### Lemma pref-classes-lists-aux

**Lemma pref-classes-lists-aux**

\[
\text{assumes} \text{is-weak-ranking} xss \text{acc} \cap (\bigcup (\text{set} xss)) = \{}
\]

**Show** \text{pref-classes-lists-aux} \text{acc} (xs\#xss) = \text{insert} \text{acc} (\text{pref-classes-lists-aux} (\text{acc} \cup \text{xss}) xss)

**Using** assms

**Proof** (induction acc xss rule: pref-classes-lists-aux.induct [case-names Nil Cons])

**Case (Cons acc xs xss)**

**From** Cons.prems have \text{A: acc} \cap (xs \cup (\text{set} xss)) = \{}

**By** (simp-all add: is-weak-ranking-Cons)

**From** Cons.prems have \text{pref-classes-lists-aux} \text{acc} (xs \# xss) = \text{insert} (\text{acc} \cup \text{xs}) ((\lambda A. A \cup \text{acc}) \ ' \text{pref-classes-lists} \text{rev} xss)) - \{\text{acc} \cup (\text{set} \text{xss})\}

**By** (intro Cons.IH) (auto simp: is-weak-ranking-Cons)

**With** Cons.prems have \text{pref-classes-lists-aux} \text{acc} (xs\# xss) = \text{insert} \text{acc} (\text{insert} (\text{acc} \cup \text{xs}) ((\lambda A. A \cup \text{acc} \cup \text{xs}) \ ' \text{pref-classes-lists} \text{rev} xss)) - \{\text{acc} \cup (\text{xs} \cup (\text{set} \text{xss}))\}

**By** (simp-all add: is-weak-ranking-Cons pref-classes-lists-append image-image Un-ac)

**Also from** \text{A have} ... = \text{insert} \text{acc} (\text{insert} (\text{acc} \cup \text{xs}) ((\lambda x. x \cup \text{acc} \cup \text{xs}) \ ' \text{pref-classes-lists} \text{rev} xss)) - \{\text{acc} \cup (\text{xs} \cup (\text{set} \text{xss}))\}

**By** blast

**Finally show** \text{?case}
by (simp-all add: pref-classes-lists-append image-image Un-ac)

qed simp-all

lemma pref-classes-list-aux-hd-tl:
  assumes is-weak-ranking xss xss \not= []
  shows pref-classes-lists-aux (hd xss) (tl xss) = pref-classes-lists (rev xss) − {\bigcup (set xss)}

proof –
  from assms have A: xss = hd xss \not= tl xss by simp
  from assms have hd xss \cap \bigcup (set (tl xss)) = {} \land is-weak-ranking (tl xss)
    by (subst (asm) A, subst (asm) is-weak-ranking-Cons) simp-all
  hence pref-classes-lists-aux (hd xss) (tl xss) =
    insert (hd xss) ((\lambda A. A \cup hd xss) \setminus pref-classes-lists (rev (tl xss))) −
    {hd xss \cup \bigcup (set (tl xss))} by (intro pref-classes-lists-aux) simp-all
  also have hd xss \cup \bigcup (set (tl xss)) = \bigcup (set xss) by (subst (3) A, subst set-simps) simp-all
  also have insert (hd xss) ((\lambda A. A \cup hd xss) \setminus pref-classes-lists (rev (tl xss))) =
    pref-classes-lists (rev (tl xss) @ [hd xss])
    by (subst pref-classes-lists-append) auto
  also have rev (tl xss) @ [hd xss] = rev xss by (subst (3) A) (simp only: rev.simps)
  finally show ?thesis .

qed

lemma pref-classes-of-weak-ranking-aux:
  assumes is-weak-ranking xss
  shows of-weak-ranking-Collect-ge xss \setminus (\bigcup (set xss)) = pref-classes-lists xss

proof safe
  fix X x assume x \in X \in set xss
  with assms show of-weak-ranking-Collect-ge xss x \in pref-classes-lists xss
    by (induction xss) (auto simp: is-weak-ranking-Cons of-weak-ranking-Collect-ge-Cons')

next
  fix x assume x \in pref-classes-lists xss
  with assms show x \in of-weak-ranking-Collect-ge xss \setminus (\bigcup (set xss))
  proof (induction xss)
    case (Cons xs xss)
    from Cons.prems consider x = xs \cup (set xss) \mid x \in pref-classes-lists xss by auto
    thus ?case
    proof cases
      assume x = xs \cup (set xss)
      with Cons.prems show ?thesis
        by (auto simp: is-weak-ranking-Cons of-weak-ranking-Collect-ge-Cons')
    next
      assume x: x \in pref-classes-lists xss
      from Cons.prems x have x \in of-weak-ranking-Collect-ge xss \setminus (\bigcup (set xss))
        by (intro Cons.IH) (simp-all add: is-weak-ranking-Cons)
      moreover from Cons.prems have xs \cap (\bigcup (set xss)) = {}
        by (simp add: is-weak-ranking-Cons)
      ultimately have x \in of-weak-ranking-Collect-ge xss \setminus (\bigcup (set xss))

  119
\[(\{x : x \notin xs\} \cup \bigcup \text{set} \ xss) \cap \{x : x \notin \text{set} \ xss\} \] by blast

\textbf{thus} thesis \ by \ (simp \ add: \ of-weak-ranking-Collect-ge-Cons')

\textbf{qed}

\textbf{qed simp-all}

\textbf{qed}

\textbf{lemma} eval-pref-classes-of-weak-ranking:
\textbf{assumes} \ \bigcup \ (\text{set} \ xss) = \text{alts} \ is-weak-ranking \ xss \ alts \neq \{} 
\textbf{shows} \ \text{pref-classes} \ \text{alts} \ (\text{of-weak-ranking} \ xss) = \text{pref-classes-lists-aux} \ (hd \ xss) \ (tl \ xss)
\textbf{proof} – 
\textbf{have} \ \text{pref-classes} \ \text{alts} \ (\text{of-weak-ranking} \ xss) = \text{preferred-alts} \ (\text{of-weak-ranking} \ xss)' (\bigcup \ (\text{set} \ (\text{rev} \ xss))) \ - \ {\bigcup \ (\text{set} \ xss)}
\textbf{by} \ (simp \ add: \ \text{pref-classes-def} \ \text{assms})
\textbf{also} \ 
\textbf{have} \ \text{of-weak-ranking-Collect-ge} \ (\text{rev} \ xss)' (\bigcup \ (\text{set} \ (\text{rev} \ xss))) = \text{pref-classes-lists} \ (\text{rev} \ xss)
\textbf{using} \ \text{assms} \ \textbf{by} \ \text{(intro \ \text{pref-classes-of-weak-ranking-aux} \ \text{simp-all})}
\textbf{also \ have} \ \text{of-weak-ranking-Collect-ge} \ (\text{rev} \ xss) = \text{preferred-alts} \ (\text{of-weak-ranking} \ xss)
\textbf{by} \ \text{(intro \ \text{ext})} \ \text{(simp-all \ add: \ \text{of-weak-ranking-Collect-ge-def} \ \text{preferred-alts-def})}
\textbf{finally \ have} \ \text{preferred-alts} \ (\text{of-weak-ranking} \ xss)' (\bigcup \ (\text{set} \ (\text{rev} \ xss))) = \text{pref-classes-lists} \ (\text{rev} \ xss) \ .

\textbf{also \ from} \ \text{assms} \ \textbf{have} \ \text{pref-classes-lists} \ (\text{rev} \ xss) \ - \ {\bigcup \ (\text{set} \ xss)} = \text{pref-classes-lists-aux} \ (hd \ xss) \ (tl \ xss)
\textbf{by} \ \text{(intro \ \text{pref-classes-list-aux-hd-tl} \ [\text{symmetric}] \ \text{auto})}
\textbf{finally \ show} \ \text{thesis} \ \textbf{by} \ \text{simp}

\textbf{qed}

\textbf{context} preorder-on
\textbf{begin}

\textbf{lemma} SD-iff-pref-classes:
\textbf{assumes} \ p \in \text{lotteries-on} \ \text{carrier} \ q \in \text{lotteries-on} \ \text{carrier}
\textbf{shows} \ p \preceq [\text{SD}(\text{le})] q \iff \ (\forall A \in \text{pref-classes} \ \text{carrier} \ \text{le}. \ \text{measure-pmf.prob} \ p \ A \leq \text{measure-pmf.prob} \ q \ A)
\textbf{proof} \ \text{safe}
\textbf{fix} \ A \ \textbf{assume} \ p \preceq [\text{SD}(\text{le})] q \ A \in \text{pref-classes} \ \text{carrier} \ \text{le}
\textbf{thus} \ \text{measure-pmf.prob} \ p \ A \leq \text{measure-pmf.prob} \ q \ A
\textbf{by} \ \text{(auto \ simp: \ \text{SD-preorder \ \text{pref-classes-def})}
\textbf{next}
\textbf{assume} \ A: \forall A \in \text{pref-classes} \ \text{carrier} \ \text{le}. \ \text{measure-pmf.prob} \ p \ A \leq \text{measure-pmf.prob} \ q \ A
\textbf{show} \ p \preceq [\text{SD}(\text{le})] q
\textbf{proof} \ \text{(rule \ \text{SD-preorder1})}

\textbf{120}
fix \( x \) assume \( x \in \text{carrier} \)
show \( \text{measure-pmf.prob p (preferred-alts le x)} \leq \text{measure-pmf.prob q (preferred-alts le x)} \)
proof (cases preferred-alts le x = carrier)
case False
with \( x \) have preferred-alts le x \( \in \) pref-classes carrier le
unfolding pref-classes-def by (intro DiffI imageI) simp-all
with A show \(?thesis by simp\)
next
case True
from assms have measure-pmf.prob p carrier = 1 measure-pmf.prob q carrier
= 1
by (auto simp: measure-pmf.prob-eq-1 lotteries-on-def AE-measure-pmf-iff)
with True show \(?thesis by simp\)
qed
qed (insert assms, simp-all)
qed

lemma \( \text{in } \text{strategyproof-an-sds} \) strategyproof':
assumes \( \text{wf}: \text{is-pref-profile R total-preorder-on alts Ri'} \) and \( i: i \in \text{agents} \)
shows \((\exists A \in \text{pref-classes alts (R i)}. \text{lottery-prob (sds (R(i := Ri'))}) A < \text{lottery-prob (sds R) A}) \lor \\
(\forall A \in \text{pref-classes alts (R i)}. \text{lottery-prob (sds (R(i := Ri'))}) A = \text{lottery-prob (sds R) A}) \)
proof –
from \( \text{wf(1)} \) interpret R: pref-profile-wf agents alts R .
from i interpret total-preorder-on alts R i by simp
from assms have \( \neg \) manipulable-profile R i Ri' by (intro strategyproof)
moreover from \( \text{wf i} \) have sds R \( \in \) lotteries sds (R(i := Ri')) \( \in \) lotteries
by (simp-all add: sds-wf)
ultimately show \(?thesis \)
by (fastforce simp: manipulable-profile-def strongly-preferred-def SD-iff-pref-classes not-le not-less)
qed

lemma pref-classes-lists-aux-finite:
\( A \in \text{pref-classes-lists-aux acc xss} \implies \text{finite acc} \implies (\forall A. A \in \text{set xss} \implies \text{finite A}) \)
\( \implies \text{finite A} \)
by (induction acc xss rule: pref-classes-lists-aux.induct) auto

lemma strategyproof-aux:
assumes \( \text{wf}: \text{prefs-from-table-wf agents alts xss1 R1 = prefs-from-table xss1} \)
prefs-from-table-wf agents alts xss2 R2 = prefs-from-table xss2
assumes sds: strategyproof-an-sds agents alts sds and \( i: i \in \text{agents} \) and \( j: j \in \text{agents} \)
assumes eq: \( R1(i := R2 j) = R2 \) the (map-of xss1 i) = xs
pref-classes-lists-aux (hd xs) (tl xs) = ps

shows \( (\exists A \in \text{ps}. (\sum x \in A. \text{pmf} (\text{sd}s \ R2) \ x) < (\sum x \in A. \text{pmf} (\text{sd}s \ R1) \ x)) \lor (\forall A \in \text{ps}. (\sum x \in A. \text{pmf} (\text{sd}s \ R2) \ x) = (\sum x \in A. \text{pmf} (\text{sd}s \ R1) \ x)) \)

proof –

from \( \text{sds interpret} \) strategyproof-an-sds agents alts \( \text{sds} \).

let \( ?R_i' = R_2 \ j \)

from \( \text{wf} \ j \) have \( \text{wf}' \) : is-pref-profile \( \text{R}_1 \) total-preorder-on alts \( ?R_i' \)

by (auto intro: pref-profile-from-tableI pref-profile-wf.prefs-wf'(1))

from \( \text{wf}(1) \ i \) have \( i \in \text{set} (\text{map} \ \text{fst} \ \text{zss1}) \) by (simp add: prefs-from-table-wf-def)

with \( \text{prefs-from-table-wfD}(3)[\text{OF} \ \text{wf}(1)] \) eq

note \( \text{xs} = \text{prefs-from-table-wfD}(2)[\text{OF} \ \text{wf}(1)] \) prefs-from-table-wfD(5,6)[\text{OF} \ \text{wf}(1)]

this

{ 
  fix \( p \ A \) assume \( A: A \in \text{pref-classes-lists-aux} (\text{hd} \ \text{xs}) (\text{tl} \ \text{xs}) \)
  from \( \text{xs} \) have \( \text{xs} \neq [] \) by auto
  with \( \text{xs} \) have \( \text{finite} \ A \)
    by (intro pref-classes-lists-aux-finite[\text{OF} \ \text{A}])
  (auto simp: is-finite-weak-ranking-def list.set-sel)
  hence \( \text{lottery-prob} p \ A = (\sum x \in A. \text{pmf} \ p \ x) \)
    by (rule measure-measure-pmf-finite)
}

note \( \text{A} = \text{this} \)

from strategyproof"[\text{OF} \ \text{wf}' \ i] \) eq have

(\( \exists A \in \text{pref-classes alts} (\text{R}_1 \ i). \text{lottery-prob} (\text{sd}s \ R2) \ A < \text{lottery-prob} \ (\text{sd}s \ R1) \ A \)
\( \lor \)
(\( \forall A \in \text{pref-classes alts} (\text{R}_1 \ i). \text{lottery-prob} (\text{sd}s \ R2) \ A = \text{lottery-prob} \ (\text{sd}s \ R1) \ A \)

by simp

also from \( \text{wf eq i} \) have \( \text{R}_1 \ i = \text{of-weak-ranking} \ \text{xs} \)

by (simp add: prefs-from-table-map-of)

also from \( \text{xs} \) have \( \text{pref-classes alts} (\text{of-weak-ranking} \ \text{xs}) = \text{pref-classes-lists-aux} (\text{hd} \ \text{xs}) (\text{tl} \ \text{xs}) \)

unfolding is-finite-weak-ranking-def by (intro eval-pref-classes-of-weak-ranking)

simp-all

finally show \( ? \text{thesis} \) by (simp add: \( A \) eq)

qed

lemma strategyproof-aux':

assumes \( \text{wf}: \text{prefs-from-table-wf agents alts xss1 \text{R}_1 \equiv \text{prefs-from-table xss1} \text{R}_2} \)

prefs-from-table-wf agents alts \( \text{xss2} \) \( \equiv \text{prefs-from-table xss2} \)

assumes \( \text{sds}: \text{strategyproof-an-sds agents alts sds \ and} \ i: i \in \text{agents} \) and \( j: j \in \text{agents} \)

assumes \( \text{perm}: \text{list-permutates} \ \text{ys} \ \text{alts} \)

defines \( \sigma \equiv \text{permutation-of-list} \ \text{ys} \) and \( \sigma' \equiv \text{inverse-permutation-of-list} \ \text{ys} \)

defines \( \text{xs} \equiv \text{the} \ (\text{map-of} \ \text{xss1} \ i) \)

defines \( \text{xs'}: \text{xs}' \equiv \text{map} (((') \ \sigma) \ (\text{the} \ (\text{map-of} \ \text{xss2} \ j)) \)
defines $R_i' \equiv \text{of-weak-ranking} \; x_i'$
assumes distinct-ps: $\forall A \in ps. \; \text{distinct} \; A$
assumes eq: $\text{mset}(\text{map} \; \text{snd} \; xss1) - \{\#(\text{map} \; \text{i} \; xss1)\} + \{\#x_i'\} =$
$\text{mset}(\text{map} \; ((\cdot)^{\prime}) \; \text{snd} \; xss2)$
$\text{pref-classes-lists-aux} \; (\text{hd} \; x) \; (\text{tl} \; x) = \text{set} \; i \; ps$
s shows
\[
\text{list-permutes} \; ys \; \text{alts} \land
(\exists A \in ps. \; (\sum x \leftarrow A. \; \text{pmf} \; (sds \; R2) \; (\sigma' \; x)) < (\sum x \leftarrow A. \; \text{pmf} \; (sds \; R1) \; x)) \lor
(\forall A \in ps. \; (\sum x \leftarrow A. \; \text{pmf} \; (sds \; R2) \; (\sigma' \; x)) = (\sum x \leftarrow A. \; \text{pmf} \; (sds \; R1) \; x))
\]
\[
(\text{is} \; - \; \text{th})
\]
proof
from perm have perm': $\sigma \; \text{permutes} \; \text{alts}$ by (simp add: $\sigma$-def)
from sds interpret strategy-proof-an-sds agents alts sds .
from wf(3) j have j \in set (map \; \text{fst} \; xss2) by (simp add: \text{prefs-from-table-wf-def})
with \text{prefs-from-table-wfD}(3)[OF \; wf(3)]
have $x_i'^{\prime}$-aux: \; (the \; (map \; \text{of} \; xss2 \; j) \; \in \; \text{set} \; (\text{map} \; \text{snd} \; xss2) \; \text{by force}
with wf(3) have $x_i'^{\prime}$-aux': \; \text{is-finite-weak-ranking} \; \text{the} \; (\text{map-of} \; xss2 \; j)
by (auto simp: \text{prefs-from-table-wf-def})
hence \; \text{is-weak-ranking} \; \text{of} \; x_i'$ unfolding $x_i'$
by (intro is-weak-ranking-map-inj permutes-inj-on[OF perm'])
(auto simp add: \text{is-finite-weak-ranking-def})
moreover from \; $x_i'^{\prime}$-aux' have \; \text{is-finite-weak-ranking} \; x_i'
by (auto simp: \; $x_i'$-is-finite-weak-ranking-def)
moreover from \; \text{prefs-from-table-wfD}(5)[OF \; wf(3)] \; \text{x_i'-aux}
have \; \bigcup (\text{set} \; x_i') = \text{alts unfolding} \; x_i'
by (simp add: image-Union [symmetric] \; \text{permutes-image}[OF \; perm'])
ultimately have \; wf-$x_i'$: \; \text{is-weak-ranking} \; x_i'$ \; \text{is-finite-weak-ranking} \; x_i'$ \bigcup (\text{set} \; x_i')
= \; \text{alts}
by (simp-all add: \; \text{is-finite-weak-ranking-def})
from \; \text{this} \; \text{wf} \; j \; \text{have} \; \text{wf}_i': \; \text{is-pref-profile} \; R1 \; \text{total-preorder-on} \; \text{alts} \; R_i'
\text{is-pref-profile} \; R2 \; \text{finite-total-preorder-on} \; \text{alts} \; R_i'
unfolding \; R_i'-def by (auto intro: \; \text{pref-profile-from-table1} \; \text{pref-profile-wf} \; \text{prefs-wf'}(1)
\text{total-preorder-of-weak-ranking})
interpret \; R1: \; \text{pref-profile-wf} \; \text{agents} \; \text{alts} \; R1 \; \text{by fact}
interpret \; R2: \; \text{pref-profile-wf} \; \text{agents} \; \text{alts} \; R2 \; \text{by fact}

from \; \text{wf}(1) \; i \; \text{have} \; i \in \text{set} \; (\text{map} \; \text{fst} \; xss1) \; \text{by (simp add: \; \text{prefs-from-table-wf-def})}
with \; \text{prefs-from-table-wfD}(3)[OF \; \text{wf}(1)] \; \text{eq}(2)
have $x_i \in \text{set} \; (\text{map} \; \text{snd} \; xss1) \; \text{unfolding} \; x_i$-def \; \text{by force}
note $x_i = \text{prefs-from-table-wfD}(2)[OF \; \text{wf}(1)] \; \text{prefs-from-table-wfD}(5,6)[OF \; \text{wf}(1)]
\; \text{this}$

from \; \text{wf} \; i \; \text{wf}' \; \text{wf}$-x_i'$ \; \text{x \; eq}
\; \text{have} \; \text{eq}_i': \; \text{anonymous-profile} \; (R1(i := R_i')) = \text{image-mset} \; (\text{map} \; ((\cdot)^{\prime}) \; \sigma)
(\text{anonymous-profile} \; R2)
by (subst \; \text{R1.anonymous-profile-update})
(simp-all add: \; \text{R_i'-def weak-ranking-of-weak-ranking} \; \text{mset-map} \; \text{multiset.map-comp}
\[
\{ \\
\text{fix } p \ A \text{ assume } A : A \in \text{pref-classes-lists-aux (hd } xs) \ (tl \ xs) \\
\text{from } xs \text{ have } xs \neq [] \text{ by auto} \\
\text{with } xs \text{ have } \text{finite } A \\
\quad \text{by (intro pref-classes-lists-aux-finite[of } A])} \\
\quad \quad \text{(auto simp: is-finite-weak-ranking-def list.set.sel)} \\
\text{hence } \text{lottery-prob } p \ A = (\sum x \in A. \text{pmf } p \ x) \\
\quad \text{by (rule measure-measure-pmf-finite)} \\
\} \\
\text{note } A = \text{this} \\
\text{from } \text{strategyproof'[OF } wf'(1,2) \ i \ eq' have} \\
(\exists A \in \text{pref-classes alts } (R1 \ i)) \ \text{lottery-prob } \text{(sds } (R1(i := R'i))) A < \text{lottery-prob } \text{(sds } R1) A \\
(\forall A \in \text{pref-classes alts } (R1 \ i)) \ \text{lottery-prob } \text{(sds } (R1(i := R'i))) A = \text{lottery-prob } \text{(sds } R1) A \\
\quad \text{by simp} \\
\text{also from } eq' \ i \text{ have } \text{sds } (R1(i := R'i)) = \text{map-pmf } \sigma \ \text{(sds } R2) \\
\quad \text{unfolding } \sigma\text{-def by (intro sds-anonymous-neutral permutation-of-list-permutates perm } wf' \\
\quad \text{pref-profile-wf.wf-update eq)} \\
\text{also from } wf \ eq \ i \text{ have } R1 \ i = \text{of-weak-ranking } xs \\
\quad \text{by (simp add: pref-classes-lists-aux (hd } xs) \ (tl \ xs)} \\
\text{also from } xs \text{ have } \text{pref-classes alts } (\text{of-weak-ranking } xs) = \text{pref-classes-lists-aux } (\text{hd } xs) \ (\text{tl } xs) \\
\quad \text{unfolding } \text{is-finite-weak-ranking-def by (intro eval-pref-classes-of-weak-ranking)} \\
\quad \quad \text{simp-all} \\
\text{finally have } (\exists A \in ps. (\sum x \leftarrow A. \text{pmf } (\text{map-pmf } \sigma \ \text{(sds } R2)) \ x) < (\sum x \leftarrow A. \text{pmf } (\text{sds } R1) \ x)) \\
\quad \quad (\forall A \in ps. (\sum x \leftarrow A. \text{pmf } (\text{map-pmf } \sigma \ \text{(sds } R2)) \ x) = (\sum x \leftarrow A. \text{pmf } (\text{sds } R1) \ x)) \\
\quad \text{using } \text{distinct-ps} \\
\quad \text{by (simp add: A eq sum.distinct-set-cone-list del: measure-map-pmf)} \\
\text{also from } \text{perm'} \text{ have } \text{pmf } (\text{map-pmf } \sigma \ \text{(sds } R2)) = (\lambda x. \text{pmf } (\text{sds } R2) (\text{inv } \sigma \ x)) \\
\quad \text{using } \text{pmf-map-inj'[of } \sigma - \text{ inv } \sigma \ x \text{ for } x} \\
\quad \text{by (simp add: fun-eq-iff permutes-inj permuates-inverses)} \\
\text{also from } \text{perm} \ \text{have } \text{inv } \sigma = \sigma' \text{ unfolding } \sigma\text{-def } \sigma'\text{-def} \\
\quad \text{by (rule inverse-permutation-of-list-correct [symmetric])} \\
\text{finally show } ?th . \\
\text{qed fact+} \\
\text{ML-file } \langle \text{randomised-social-choice.ML} \rangle \\
\text{ML-file } \langle \text{sds-automation.ML} \rangle \\
\text{end}
References