Randomised Social Choice

Manuel Eberl

August 16, 2018

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
2.5 Preference profiles from lists ........................................ 35
2.6 Automatic evaluation of preference profiles .................... 39

3 Auxiliary facts about PMFs ........................................... 44
  3.1 Definition of von Neumann–Morgenstern utility functions .. 45

4 Stochastic Dominance .................................................. 48
  4.1 Definition of Stochastic Dominance ............................... 48
  4.2 Stochastic Dominance for preference profiles ................. 51
  4.3 SD efficient lotteries ................................................ 52
  4.4 Equivalence proof ................................................... 54
  4.5 Existence of SD-efficient lotteries ............................... 64

5 Social Decision Schemes .............................................. 67
  5.1 Basic Social Choice definitions ................................... 68
  5.2 Social Decision Schemes .......................................... 68
  5.3 Anonymity ........................................................... 68
  5.4 Neutrality ........................................................... 69
  5.5 Ex-post efficiency ................................................... 71
  5.6 SD efficiency ........................................................ 72
  5.7 Weak strategyproofness .......................................... 73
  5.8 Strong strategyproofness ............................................ 74

6 Lowering Social Decision Schemes ................................ 75

7 Random Dictatorship ................................................... 83
  7.1 Anonymity ........................................................... 85
  7.2 Neutrality ........................................................... 86
  7.3 Strong strategyproofness ............................................ 86

8 Random Serial Dictatorship ......................................... 87
  8.1 Auxiliary facts about RSD ......................................... 89
    8.1.1 Pareto-equivalence classes .................................. 89
    8.1.2 Facts about RSD winners ..................................... 90
  8.2 Proofs of properties ............................................... 95
    8.2.1 Well-definedness .............................................. 95
    8.2.2 RD extension .................................................. 95
    8.2.3 Anonymity ..................................................... 95
    8.2.4 Neutrality ..................................................... 96
    8.2.5 Ex-post efficiency ............................................ 96
    8.2.6 Strong strategy-proofness ................................. 97

9 Automatic definition of Preference Profiles ..................... 98
  9.1 Automatic definition of preference profiles from tables .... 100
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

type-synonym 'a relation = 'a ⇒ 'a ⇒ bool

definition map-relation :: ('a ⇒ 'b) ⇒ 'b relation ⇒ 'a relation where
  map-relation f R = (λx y. R (f x) (f y))
definition restrict-relation :: 'a set ⇒ 'a relation ⇒ 'a relation where
  restrict-relation A R = (λx y. x ∈ A ∧ y ∈ A ∧ R x y)

lemma restrict-relation-restrict-relation [simp]:
  restrict-relation A (restrict-relation B R) = restrict-relation (A ∩ B) R
  by (intro ext) (auto simp add: restrict-relation-def)

lemma restrict-relation-empty [simp]: restrict-relation {} R = (λ- -. False)
  by (simp add: restrict-relation-def)

lemma restrict-relation-UNIV [simp]: restrict-relation UNIV R = R
  by (simp add: restrict-relation-def)

1.2 Preorders

Preorders are reflexive and transitive binary relations.

locale preorder-on =
  fixes carrier :: 'a set
  fixes le :: 'a relation
  assumes not-outside: le x y ⇒ x ∈ carrier le x y ⇒ y ∈ carrier
  assumes refl: x ∈ carrier ⇒ le x x
  assumes trans: le x y ⇒ le y z ⇒ le x z
begin

lemma carrier-eq: carrier = {x. le x x}
  using not-outside refl by auto

lemma preorder-on-map:
  preorder-on (f − ' carrier) (map-relation f le)
  by unfold-locales (auto dest: not-outside simp: map-relation-def refl elim: trans)

4
lemma preorder-on-restrict:
preorder-on \((\text{carrier} \cap A)\) \((\text{restrict-relation} \ A \ \text{le})\)
by unfold-locales \((\text{auto simp: restrict-relation-def refl intro: trans not-outside})\)

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

lemma restrict-relation-carrier \([\text{simp}]\):
restrict-relation \(\text{carrier} \ \text{le} \ = \ \text{le}\)
using not-outside by \((\text{intro ext})\) \((\text{auto simp add: restrict-relation-def})\)

end

1.3 Total preorders

Total preorders are preorders where any two elements are comparable.

locale total-preorder-on = preorder-on +
assumes total: \(x \in \text{carrier} \implies y \in \text{carrier} \implies \text{le} \ x \ y \ \vee \ \text{le} \ y \ x\)
begin

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

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

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

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

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)

abbreviation (input) weakly-not-preferred (¬ ≽[−] - [51,10,51] 60) where
  a ≽[R] b ≡ b ≽[R] a
term a ≥[R] b ←→ b ≤[R] a

abbreviation (input) strongly-not-preferred (¬ ≫[−] - [51,10,51] 60) where
  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)
sows 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)} ≠ {}
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 \in A \land z \in A \implies le y z \implies 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 \( \neq \) \{\} \implies finite carrier \implies Max-wrt le \( \neq \) \{\}
  using Max-wrt-among-nonempty[of UNIV] by (simp add: Max-wrt-def)

lemma Max-wrt-among-map-relation-vimage:
  \( f \comp\) Max-wrt-among le A \subseteq Max-wrt-among (map-relation f le) (\( f \comp\) A)
  by (auto simp: Max-wrt-among-def map-relation-def)

lemma Max-wrt-map-relation-vimage:
  \( f \comp\) Max-wrt le \subseteq 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 \subseteq A
  shows f \comp B \subseteq the-inv-into A f \comp B
  using assms by (auto simp: the-inv-into-f-f)

lemma Max-wrt-among-map-relation-bij-subset:
  assumes bij f :: \(\) a \Rightarrow \(\) b
  shows f \comp Max-wrt-among le A \subseteq Max-wrt-among (map-relation (inv f) le) (f \comp A)
  using assms Max-wrt-among-map-relation-vimage[of inv f A]
  by (simp add: bij-imp-bij-inv inv-inv-eq bij-vimage-eq-inv-image)

lemma Max-wrt-among-map-relation-bij:
  assumes bij f
  shows f \comp Max-wrt-among le A = Max-wrt-among (map-relation (inv f) le) (f \comp A)
  proof (intro equalityI Max-wrt-among-map-relation-bij-subset assms)
    interpret R: preorder-on f \ comp carrier map-relation (inv f) le
      using preorder-on-map[of inv f] assms
    by (simp add: bij-imp-bij-inv inv-inv-eq inv-inv-eq)
    show Max-wrt-among (map-relation (inv f) le) (f \comp A) \subseteq f \ comp 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 inv-f-f image-Int [symmetric])
qed

lemma Max-wrt-map-relation-bij:
\[ \text{bij } f \implies f \circ \text{Max-wrt } \leq = \text{Max-wrt } (\text{map-relation } (\text{inv } f) \leq) \]
proof -
  assume bij: bij f
  interpret R: preorder-on f \circ carrier map-relation (\text{inv } f) \leq
  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:
\[ \leq x y \implies x \in \text{Max-wrt-among } \leq A \implies y \in A \implies y \in \text{Max-wrt-among } \leq A \]
using not-outside by (auto simp: Max-wrt-among-preorder intro: trans)

lemma Max-wrt-mono:
\[ \leq x y \implies x \in \text{Max-wrt } \leq \implies y \in \text{Max-wrt } \leq \]
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:
\[ \text{Max-wrt-among } \leq A = \{x \in \text{carrier } \cap A. \forall y \in \text{carrier } \cap A. \leq y x\} \]
unfolding Max-wrt-among-preorder using total by blast

lemma Max-wrt-total-preorder:
\[ \text{Max-wrt } \leq = \{x \in \text{carrier}. \forall y \in \text{carrier}. \leq y x\} \]
unfolding Max-wrt-preorder using total by blast

lemma decompose-Max:
  assumes A: A \subseteq\ carrier
  defines M \equiv \text{Max-wrt-among } \leq A
  shows restrict-relation A \leq = (\lambda x y. x \in A \land y \in M \land restrict-relation (A - M) \leq x y)
  using A by (intro ext) (auto simp: M-def Max-wrt-among-total-preorder
  restrict-relation-def Int-absorb1 intro: trans)
end

1.6 Weak rankings

inductive of-weak-ranking :: 'alt set list => 'alt relation where
\begin{align*}
i &\leq j \implies i < \text{length } zs \implies j < \text{length } zs \implies x \in xs \implies i \implies y \in xs \implies j \implies \\
x &\geq [\text{of-weak-ranking } xs] y
\end{align*}

**Lemma of-weak-ranking-Nil** [simp]: of-weak-ranking \(\square = \lambda x. \text{False}\)
by (intro ext) (simp add: of-weak-ranking.simps)

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

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

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

**Lemma of-weak-ranking-indifference**:
assumes \(A \in \text{set } xs\) \(x \in A\) \(y \in A\)
shows \(x \geq [\text{of-weak-ranking } xs] y\)
using assms by (induction \(xs\)) (auto simp: of-weak-ranking-Cons)

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

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

definition is-weak-ranking where
\( \text{is-weak-ranking } xs \leftrightarrow ((\{\} \notin \text{set } xs) \land \ (\forall i \ j. \ i < \text{length } xs \land \ j < \text{length } xs \land \ i \neq j \rightarrow \text{xs ! } i \cap \text{xs ! } j = \{\}) \)

definition is-finite-weak-ranking where
\( \text{is-finite-weak-ranking } xs \leftrightarrow \text{is-weak-ranking } xs \land \ (\forall x \in \text{set } xs. \ \text{finite } x) \)

definition weak-ranking :: 'alt relation \Rightarrow 'alt set list where
\( \text{weak-ranking } R = (\text{SOME } xs. \ \text{is-weak-ranking } xs \land R = \text{of-weak-ranking } xs) \)

lemma is-weak-rankingI [intro?]:
assumes \( \{\} \notin \text{set } xs \land \ (\forall i \ j. \ i < \text{length } xs \rightarrow j < \text{length } xs \rightarrow i \neq j \rightarrow \text{xs ! } i \cap \text{xs ! } j = \{\}) \)
shows \( \text{is-weak-ranking } xs \)
using assms by (auto simp add: is-weak-ranking-def)

lemma is-weak-ranking-nonempty: \( \text{is-weak-ranking } xs \rightarrow \{\} \notin \text{set } xs \)
by (simp add: is-weak-ranking-def)

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

lemma is-weak-ranking-iff:
\( \text{is-weak-ranking } xs \leftrightarrow \text{distinct } xs \land \text{disjoint} \ (\text{set } xs) \land \{\} \notin \text{set } xs \)
proof safe
assume wf: \( \text{is-weak-ranking } xs \)
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 < length xs j < length xs i ≠ j xs ! i = xs ! j 
  then have xs ! i ∩ xs ! j = {} by (intro is-weak-rankingD wf)
  with ij have xs ! i = {} by simp 
  with ij have {} ∈ set xs by (auto simp: set-conv-nth) 
  moreover from wf ij have {} ∉ set xs by (intro is-weak-ranking-nonempty wf)
  ultimately show False by contradiction 
qed 
next
assum A: distinct xs disjoint (set xs) {} ∉ 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) ←→ 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 (⋃ 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 ←→ 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 < length xs j < length xs x ∈ xs ! i y ∈ xs ! j i ≥ 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-Cons-empty [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]:
\(-\text{is-weak-ranking } (\varnothing \# xs)\) by (simp add: is-weak-ranking-def)

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

**lemma** is-weak-ranking-singleton [simp]:
is-weak-ranking \([x] \leftrightarrow x \neq \varnothing\)
by (auto simp add: is-weak-ranking-def)

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

**lemma** is-weak-ranking-append:
is-weak-ranking \((xs @ ys) \leftrightarrow\)
is-weak-ranking \(xs \land is-weak-ranking ys \land\)
\(((\text{set } xs \cap \text{set } ys) = \{\}) \land ((\bigcup \text{set } xs) \cap (\bigcup \text{set } ys) = \{\}))\)
by (simp only: is-weak-ranking-iff)
(auto dest: disjointD disjoint-unionD1 disjoint-unionD2 intro: disjoint-union)

**lemma** is-weak-ranking-Cons [code]:
is-weak-ranking \((x \# xs) \leftrightarrow\)
x \neq \varnothing \land is-weak-ranking \(xs \land x \cap \bigcup \text{set } xs = \{\}\)
using is-weak-ranking-append[of \([x] xs\)] by auto

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

primrec is-weak-ranking-aux where
is-weak-ranking-aux \(A \varnothing \leftrightarrow \text{True}\)
| is-weak-ranking-aux \(A (x \# xs) \leftrightarrow x \neq \varnothing \land\)
| \(A \cap x = \{\} \land is-weak-ranking-aux (A \cup x) xs\)

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

**lemma** is-weak-ranking-code [code]:
is-weak-ranking \(xs \leftrightarrow is-weak-ranking-aux \{\} xs\)
by (subst is-weak-ranking-aux) auto

**lemma** of-weak-ranking-altdef:
assumes is-weak-ranking \(xs \in \bigcup \text{set } xs \ y \in \bigcup \text{set } xs\)
shows of-weak-ranking \(xs \ y \leftrightarrow\)
\(\text{find-index } ((\varepsilon) \ x) xs \geq \text{find-index } ((\varepsilon) \ y) xs\)
proof –
from assms
  have A: find-index \( ((\in) x) \) \( xs < \) \( length xs \) find-index \((\in) y) \) \( xs < \) \( length xs \) by (simp-all add: find-index-less-size-conv)
from this[THEN nth-find-index]
  have B: \( x \in xs \) \( \land \) find-index \((\in) x) \) \( xs \) \( y \in xs \) \( \land \) find-index \((\in) y) \) \( xs \)
show ?thesis
proof
  assume of-weak-ranking \( xs \) \( x \) \( y \)
  then obtain \( i \) \( j \) where \( ij: j \leq i \) \( i < length xs \) \( j < length xs \) \( x \in xs \) \( \land \) \( i y \in xs \)
  by (cases rule: of-weak-ranking.cases) simp-all
  with A B have \( i = \) find-index \((\in) x) \) \( xs \) \( j = \) find-index \((\in) y) \) \( xs \)
   using assms(1) unfolding is-weak-ranking-def by blast+
  with \( ij \) show find-index \((\in) x) \) \( xs \) \( \geq \) \( find-index \((\in) y) \) \( xs \) by simp
next
  assume find-index \((\in) x) \) \( xs \) \( \geq \) \( find-index \((\in) y) \) \( xs \)
from this A(\(2,1\)) B(\(2,1\)) show of-weak-ranking \( xs \) \( x \) \( y \)
  by (rule of-weak-ranking.intros)
qed

lemma total-preorder-of-weak-ranking:
  assumes \( \bigsqcup \) set \( xs = A \)
  assumes is-weak-ranking \( xs \)
  shows total-preorder-on \( A \) (of-weak-ranking \( xs \))
proof
  fix \( x \) \( y \)
  assume x \( \sqsubseteq\) of-weak-ranking \( xs \) \( y \)
  with assms show \( x \in A \) \( y \in A \)
   by (auto simp: set-conv-nth)
next
  fix \( x \) \( y \)
  assume x \( \in \) \( A \) \( y \in A \)
  with assms(1) obtain \( i \) \( j \) where \( ij: i < length xs \) \( j < length xs \) \( x \in xs \) \( \land \) \( i y \in xs \)
   by (auto elim!: of-weak-ranking.cases)
  moreover from B obtain \( j \) \( k \)
next
  fix \( x \) \( y \) \( z \)
  assume A: \( x \leq\) of-weak-ranking \( xs \) \( y \) \( \land \) B: \( y \leq\) of-weak-ranking \( xs \) \( z \)
  from A obtain \( i \) \( j \)
    where \( ij: i \geq j \) \( i < length xs \) \( j < length xs \) \( x \in xs \) \( \land \) \( i y \in xs \)
     by (auto elim!: of-weak-ranking.cases)
  moreover from B obtain \( j \) \( k \)
next
where $j'k: j' \geq k \land j' < \text{length } xs \land k < \text{length } xs \land y \in xs \land j'z \in xs \land k$

by (auto elim! of-weak-ranking_cases)

moreover from $ijj'k$ is-weak-rankingD[OF assms(2), of $jj'k$]

have $j = j'$ by blast

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

qed

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

assumes is-weak-ranking $(A \# As)$

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

proof –

from 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: is-weak-ranking-Cons)

from assms show ?thesis using $R$.not-outside

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

is-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 $x :: 'alt set$ and $xs :: 'alt set \text{ list}$

assumes wf: 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: wf)

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

assumes of-weak-ranking $xs$ $a$ $b$

shows $a \in \bigcup \text{set } xs \land b \in \bigcup \text{set } xs$

using assms by (fastforce elim!: of-weak-ranking_cases)+

lemma of-weak-ranking-Cons':

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

shows of-weak-ranking $(x\#xs)$ $a$ $b \iff b \in x \vee (a \notin x \land \text{of-weak-ranking } xs$
proof
assume of-weak-ranking \((x \neq xs)\) a b
with wf of-weak-ranking-imp-in-set[of a b]
show \((b \in x \lor a \notin x \land of\text{-}weak-ranking xs a b)\)
by (auto simp: is-weak-ranking-Cons of-weak-ranking-Cons)

next
assume \(b \in x \lor a \notin x \land of\text{-}weak-ranking xs a b\)
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 \(\bigcup \text{set} \ xs \ of\text{-}weak-ranking \ xs\)
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 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 \(\bigcup \text{set} \ xs \ of\text{-}weak-ranking \ xs\)
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 \((x\#xs)\)) \(A\) =
(if \(x \cap A = \{\}\) 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 (relation Wellfounded.measure card)
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

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 thesis by auto

qed simp-all
lemma weak-ranking-aux-wf:
  \[ A \subseteq \text{carrier} \implies \text{is-weak-ranking} (\text{weak-ranking-aux} A) \]
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 \neq \{\}
    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 \neq \{\}
    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 \cap \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 \subseteq \text{carrier} \ x, y \in A
  shows of-weak-ranking (weak-ranking-aux A) x y \longleftrightarrow \text{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 \subseteq A unfolding M-def
  by (intro Max-wrt-among-subset)
  from nonempty.prems have in-MD: le x y if x \in A y \in 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 \in M if y \in M x \in 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 \notin M y \notin 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

lemma weak-ranking-aux-unique':
  assumes \( \bigcup \text{set} \text{As} \subseteq \text{carrier} \) is-weak-ranking \( \text{As} \)
  of-weak-ranking \( \text{As} = \) restrict-relation (\( \bigcup \text{set} \text{As} \)) le
shows \( \text{As} = \) weak-ranking-aux (\( \bigcup \text{set} \text{As} \))
using assms
proof (induction \( \text{As} \))
  case (Cons \( A \) \( \text{As} \))
  have restrict-relation (\( \bigcup \text{set} \text{As} \)) (of-weak-ranking (\( A \neq \text{As} \))) = of-weak-ranking \( \text{As} \)
by (intro restrict-relation-of-weak-ranking-Cons Cons,prems)
also have eq1: of-weak-ranking (A # As) = restrict-relation (\bigcup \text{set} \ (A \# As))
le by fact
finally have eq: of-weak-ranking As = restrict-relation (\bigcup \text{set} As) le
by (simp add: Int-absorb2)

from eq1 have
  Max-wrt-among le (\bigcup \text{set} \ (A \# As)) =
  Max-wrt-among (of-weak-ranking (A\#As)) (\bigcup \text{set} \ (A\#As))
by (intro Max-wrt-among-cong) simp-all
also from Cons.prems have ... = A
  by (subst Max-wrt-among-of-weak-ranking-Cons2)
    (simp-all add: is-weak-ranking-Cons)
finally have Max: Max-wrt-among le (\bigcup \text{set} \ (A \# As)) = A .

moreover from Cons.prems have A \neq \{\} by (simp add: is-weak-ranking-Cons)
ultimately have weak-ranking-aux (\bigcup \text{set} \ (A \# As)) = A \# weak-ranking-aux
(A \cup \bigcup \text{set} As - A)
using Cons.prems by simp
also from Cons.prems have A \cup \bigcup \text{set} As - A = \bigcup \text{set} As
  by (auto simp: is-weak-ranking-Cons)
also from eq2 have weak-ranking-aux ... = As .
finally show ?case ..
qed simp-all

lemma weak-ranking-aux-unique:
assumes is-weak-ranking As of-weak-ranking As = le
shows As = weak-ranking-aux carrier
proof –
interpret R: total-preorder-on \bigcup \text{set} As of-weak-ranking As
by (intro total-preorder-of-weak-ranking assms) simp-all
from assms have x \in (\bigcup \text{set} As) \iff x \in carrier for x
  using R.not-outside not-outside R.refl[of x] refl[of x]
by blast
hence eq: \bigcup \text{set} As = carrier by blast
from assms eq have As = weak-ranking-aux (\bigcup \text{set} As)
  by (intro weak-ranking-aux-unique') simp-all
with eq show ?thesis by simp
qed

lemma weak-ranking-total-preorder:
is-weak-ranking (weak-ranking le) of-weak-ranking (weak-ranking le) = le
proof –
from weak-ranking-aux-wf[of carrier] of-weak-ranking-weak-ranking-aux
have \exists x. is-weak-ranking x \land le = of-weak-ranking x by auto
hence is-weak-ranking (weak-ranking le) \land le = of-weak-ranking (weak-ranking le)

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

lemma weak-ranking-Union: (∪ set (weak-ranking le)) = carrier
  by (simp add: weak-ranking-altdef 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-altdef
  by (rule weak-ranking-aux-unique)

lemma weak-ranking-permute:
  assumes f permutes carrier
  shows weak-ranking (map-relation (inv f) le) = map (('f) 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) 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 x ∈ xs ! i x ∈ 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 ∈ xs ! i
  shows i = find-index ((∈) 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[of (∈) x]] blast+

lemma weak-ranking-eqclass1:
  assumes A ∈ set (weak-ranking le) x ∈ A y ∈ 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.intro[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
  define xs where xs = weak-ranking 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 xs-def)
  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 i'j': j' ≤ i' i' < length xs j' < length xs x ∈ xs ! j' y ∈ xs ! i'
    by (cases rule: of-weak-ranking.cases)
  from ij i'j' have eq: i = j' j = i'
    by (intro weak-ranking-index-unique[OF wf]; simp)+
  moreover from assms (2) weak-ranking-Union have weak-ranking le ≠ [] by auto
  ultimately have k = i using ij i'j' A
    by (auto simp: xs-def set-conv-nth)
  with ij i'j' k eq show ?thesis by (auto simp: xs-def)
qed

lemma hd-weak-ranking:
  assumes x: x ∈ hd (weak-ranking le) y ∈ carrier
  shows le y x
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.intro[of 0 i]) (auto simp: hd-conv-nth)
  thus ?thesis by (simp add: weak-ranking-total-preorder)
qed

lemma last-weak-ranking:
  assumes x: x ∈ last (weak-ranking le) y ∈ carrier
shows le x y
proof
  from weak-ranking-Union assms obtain i
  where 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) x y using assms(1)
  by (intro of-weak-ranking.intros[of i length (weak-ranking le) - 1])
  (auto simp: last-cone-nth)
thus ?thesis by (simp add: weak-ranking-total-preorder)
qed

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 weak-ranking-index :: 'a ⇒ nat where
  weak-ranking-index x = find-index (λA. x ∈ A) (weak-ranking le)

lemma nth-weak-ranking-index:
  assumes x ∈ carrier
  shows weak-ranking-index x < length (weak-ranking le)
    x ∈ weak-ranking le ! weak-ranking-index x
proof
  from assms weak-ranking-Union show weak-ranking-index x < length (weak-ranking le)
    unfolding weak-ranking-index-def by (auto simp add: find-index-less-size-conv)
thus x ∈ weak-ranking le ! weak-ranking-index x unfolding weak-ranking-index-def
  by (rule nth-find-index)
qed

lemma ranking-index-eqI:
  i < length (weak-ranking le) =⇒ x ∈ weak-ranking le ! i =⇒ weak-ranking-index
  x = i
  using weak-ranking-index-unique[of weak-ranking le i x]
  by (simp all add: weak-ranking-index-def weak-ranking-total-preorder)

lemma ranking-index-le-iff [simp]:
  assumes x ∈ carrier y ∈ carrier
  shows weak-ranking-index x ≥ weak-ranking-index y =⇒ le x y
proof
  have le x y =⇒ of-weak-ranking (weak-ranking le) x y
    by (simp add: weak-ranking-total-preorder)
  also have ... =⇒ weak-ranking-index x ≥ weak-ranking-index y
  proof
    assume weak-ranking-index x ≥ weak-ranking-index y
    thus of-weak-ranking (weak-ranking le) x y
      by (rule of-weak-ranking.intros) (simp all add: nth-weak-ranking-index assms)
  next
    assume of-weak-ranking (weak-ranking le) x y

23
then obtain $i \ j$ where

$i \leq j \ \ i < \ \ length \ (weak-ranking \ \ le) \ \ j < \ \ length \ (weak-ranking \ \ le)$

$x \in \ weak-ranking \ \ le \ \ ! \ j \ y \in \ weak-ranking \ \ le \ ! \ i$

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

with ranking-index-eqI[of $i$] ranking-index-eqI[of $j$]

show weak-ranking-index $x \geq weak-ranking-index \ y$ 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-locale 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-iiff:

finite-total-preorder-on $A$ $R$ $\iff$ total-preorder-on $A$ $R$ $\wedge$ finite $A$

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

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

assumes $\bigcup$ set $xs$ = $A$ 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-locale (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 $\bigcup$ set $xs$ 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  \( R_1 = R_2 \)
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 ⇒ '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 ≠ {}
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

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 = (λx. 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 split: if-splits)

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

lemma wf-update:
  assumes i ∈ agents total-preorder-on alts (R i')
  shows pref-profile-wf agents alts (R(i := R i'))
proof -
  interpret total-preorder-on alts R i' by fact
  from finite-alts have finite-total-preorder-on alts R i' 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 ∘ σ)
unfolding o-def using permutes-in-image[OF assms(1)]
by (intro pref-profile-wf.intro prefs-wf) simp-all

27
lemma (in −) pref-profile-eqI:
assumes pref-profile-wf agents alts R1 pref-profile-wf agents alts R2
assumes \(\forall x. x \in \text{agents} \Rightarrow R1 x = R2 x\)
shows \(R1 = R2\)
proof
interpret \(R1: \text{pref-profile-wf agents alts R1}\) by fact
interpret \(R2: \text{pref-profile-wf agents alts R2}\) by fact
fix \(x\) show \(R1 x = R2 x\)
  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 \(\text{permute-profile } f (\text{permute-profile } g \ R) = \text{permute-profile } (f \circ g) \ R\)
using assms by (simp add: permute-profile-def o-inv-distrib)

lemma (in pref-profile-wf) wf-permute-alts:
assumes \(\sigma\) \(\text{permutes}\) \(\text{alts}\)
shows \(\text{pref-profile-wf}\) \(\text{agents}\) \(\text{alts}\) (permute-profile \(\sigma\) \(R\))
proof (rule pref-profile-wf.intro)
fix \(i\) assume \(i \in \text{agents}\)
with assms interpret \(R: \text{finite-total-preorder-on}\) \(\text{alts}\) \(\text{R i}\) 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 \(\text{alts}\) (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 R.not-outside[of inv σ x inv σ y]
by (auto simp: permute-profile-def)

next
fix x y z assume permute-profile σ R i x y permute-profile σ R i y z
thus permute-profile σ R i x z
using R.trans[of inv σ x inv σ y inv σ z]
by (simp-all add: permute-profile-def)

This shows that the above definition is equivalent to that in the paper.

lemma permute-profile-iff [simp]:
fixes R :: ('agent, 'alt) pref-profile
assumes σ permutes alts x ∈ alts y ∈ alts
defines R' ≡ permute-profile σ R
shows σ x ⪯[R' i] σ y ←→ x ⪯[R i] y
using assms by (simp add: permute-profile-def permutes-inverses)

2.1 Pareto dominance

definition Pareto :: ('agent ⇒ 'alt relation) ⇒ 'alt relation where
x ⪯[Pareto R] y ←→ (∃j. x ⪯[R j] x) ∧ (∀i. x ⪯[R i] x → x ⪯[R i] y)

A Pareto loser is an alternative that is Pareto-dominated by some other alternative.

definition pareto-losers :: ('agent, 'alt) pref-profile ⇒ 'alt set where
pareto-losers R = {x. ∃y. y >[Pareto R] x}

lemma pareto-losersI [intro?, simp]: y >[Pareto R] x → x ∈ pareto-losers R
by (auto simp: pareto-losers-def)

context preorder-family
begin

lemma Pareto-iff:
  x ⪯[Pareto R] y ←→ (∀i∈dom. x ⪯[R i] y)
proof
  assume A: x ⪯[Pareto R] y
  then obtain j where j: x ⪯[R j] x by (auto simp: Pareto-def)
  hence j*: j ∈ dom by (cases j ∈ dom) auto
  then interpret preorder-on carrier R j by simp
  from j have x ∈ carrier by (auto simp: carrier-eq)
  with A preorder-on.refl[OF in-dom]
  show (∀i∈dom. x ⪯[R i] y) by (auto simp: Pareo-def)

next
  assume A: (∀i∈dom. x ⪯[R i] y)
  from nonempty-dom obtain j where j: j ∈ dom by blast
  then interpret preorder-on carrier R j by simp
  from j A have x ⪯[R j] y 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)
qd

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

lemma Pareto-strictI’:
assumes $(\forall i. \ i \in \text{dom} \Longrightarrow x \preceq [R i] y) \ i \in \text{dom} \neg x \preceq [R i] y$
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 ?thesis by (auto simp: Pareto-strict-iff)
qd

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 $A = \text{preorder-on.not-outside}[\text{OF this}(1)] \ \text{preorder-on.refl}[\text{OF this}(1)]$
preorder-on.trans[\text{OF this}(1)]
from nonempty-dom obtain $i \ where$ $i \ i \in \text{dom}$
show preorder-on carrier $(\text{Pareto} \ R)$
proof
fix $x y$ assume $x \preceq [\text{Pareto}(R)] y$
with $A(1,2)[OF \ i]$ i show $x \in \text{carrier} \ y \in \text{carrier}$ by (auto simp: Pareto-iff)
qd

lemma pareto-loser-in-alt:
assumes $x \in \text{pareto-losers} \ R$
shows $x \in \text{carrier}$
proof
from assms obtain $y \ i \ where$ $i \ i \in \text{dom}$
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
qd

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

30
obtains y where y ∈ carrier y ≻ [Pareto(R)] x

proof –
from assms obtain y where y: y ≻ [Pareto(R)] x unfolding pareto-losers-def
by blast
  with Pareto,not-outside[of x y] have y ∈ 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 ⊆ alts (is ?A)
and finite-preferred-alts [simp,intro!]: finite (preferred-alts (R i) x) (is ?B)
proof –
  have ?A ∧ ?B
proof (cases i ∈ agents)
  assume i ∈ agents
  then interpret total-preorder-on alts R i by simp
  have preferred-alts (R i) x ⊆ alts using not-outside
    by (auto simp: preferred-alts-def)
  thus ?thesis by (auto dest: finite-subset)
qed (auto simp: preferred-alts-def)
thus ?A ∧ ?B by blast+
qed

lemma preferred-alts-altdef:
i ∈ agents =⇒ preferred-alts (R i) x = {y ∈ alts. y ≥ [R i] x}
by (simp add: preorder-on.preferred-alts-altdef)
end

2.3 Favourite alternatives

definition favorites :: ('agent, 'alt) pref-profile ⇒ 'agent ⇒ 'alt set where
  favorites R i = Max-wrt (R i)

definition favorite :: ('agent, 'alt) pref-profile ⇒ 'agent ⇒ 'alt where
  favorite R i = the-elem (favorites R i)

definition has-unique-favorites :: ('agent, 'alt) pref-profile ⇒ bool where
  has-unique-favorites R = (∀ i. favorites R i = {}) ∨ is-singleton (favorites R i))

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

proof (cases \(i \in \text{agents}\))
  assume \(i \in \text{agents}\)
  then interpret total-preorder-on \(R \ i\) by simp
  show \(?\text{thesis}\)
  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]: \(i \notin \text{agents} \implies \text{favorites } R \ i = \{\}\)
  by (auto simp: favorites-def Max-wrt-def Max-wrt-among-def)

lemma favorites-altdef':
  \(\text{favorites } R \ i = \{x \in \text{alts}. \ \forall y \in \text{alts}. \ x \succeq [R \ i] y\}\)

proof (cases \(i \in \text{agents}\))
  assume \(i \in \text{agents}\)
  then interpret finite-total-preorder-on \(R \ i\) by simp
  by (auto simp: favorites-altdef Max-wrt-among-total-preorder)
qed simp-all

lemma favorites-subset-alts: \(\text{favorites } R \ i \subseteq \text{alts}\)
  by (auto simp: favorites-altdef')

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

lemma favorites-nonempty: \(i \in \text{agents} \implies \text{favorites } R \ i \neq \{\}\)
  proof –
  assume \(i \in \text{agents}\)
  then interpret finite-total-preorder-on \(R \ i\) by simp
  show \(?\text{thesis}\) unfolding favorites-def by (intro Max-wrt-nonempty) simp-all
qed

lemma favorites-permute:
  assumes \(i: i \in \text{agents} \text{ and perm: } \sigma \text{ permutes alts}\)
  shows \(\text{favorites } (\text{permute-profile } \sigma \ R) \ i = \sigma \cdot \text{favorites } R \ i\)
  proof –
  from \(i\) interpret finite-total-preorder-on \(R \ i\) by simp
  from perm show \(?\text{thesis}\)
  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:
  \(\text{has-unique-favorites } R \longleftrightarrow (\forall i \in \text{agents}. \text{is-singleton } (\text{favorites } R \ i))\)
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

type-synonym ('agent, 'alt) apref-profile = 'alt set list multiset

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
qd
lemma (in pref-profile-wf) anonymous-profile-def:
anonymous-profile R = image-mset (weak-ranking o R) (mset-set agents)
by (simp only: agents-eq anonymous-profile-auxdef)

lemma (in pref-profile-wf) anonymous-profile-permute:
assumes σ permutes alts finite agents
shows anonymous-profile (permute-profile σ R) =
    image-mset (map ((′) σ)) (anonymous-profile R)
proof –
from assms(1) interpret R': pref-profile-wf agents alts permute-profile σ R
    by (rule wf-permute-alts)
have anonymous-profile (permute-profile σ R) =
    {#weak-ranking (map-relation (inv σ) (R x)). x ∈# mset-set agents#}
    unfolding R'.anonymous-profile-def
    by (simp add: multiset.map-comp permute-profile-map-relation o-def)
also from assms have ... = {#map ((′) σ) (weak-ranking (R x)). x ∈# mset-set agents#}
    by (intro image-mset-cong)
    (simp add: finite-total-preorder-on.weak-ranking-permute[of alts])
also have ... = image-mset (map ((′) σ)) (anonymous-profile R)
    by (simp add: anonymous-profile-def multiset.map-comp o-def)
finally show ?thesis .
qed

lemma (in pref-profile-wf) anonymous-profile-update:
assumes i: i ∈ agents and fin [simp]: finite agents and total-preorder-on alts Ri'
shows anonymous-profile (R(i := Ri')) =
    anonymous-profile R - {#weak-ranking (R i)#} + {#weak-ranking Ri'#}
proof –
from assms interpret R': pref-profile-wf agents alts R(i := Ri')
    by (simp add: finite-total-preorder-on-iff wf-update)
have anonymous-profile (R(i := Ri')) =
    {#weak-ranking (if x = i then Ri' else R x). x ∈# mset-set agents#}
    by (simp add: R'.anonymous-profile-def o-def)
also have ... = {#if x = i then weak-ranking Ri' else weak-ranking (R x). x ∈# mset-set agents#}
    by (intro image-mset-cong) simp-all
also have ... = {#weak-ranking Ri', x ∈# mset-set {x ∈ agents. x = i}#} +
    {#weak-ranking (R x). x ∈# mset-set {x ∈ agents. x ≠ i}#}
    by (subst image-mset-If) ((subst filter-mset-mset-set., simp)+, rule refl)
also from i have {x ∈ agents. x = i} = {i} by auto
also have {x ∈ agents. x ≠ i} = agents - {i} by auto
also have {#weak-ranking Ri'. x ∈# mset-set {i}#} = {#weak-ranking Ri'#}
by simp
also from i have mset-set (agents - {i}) = mset-set agents - {#i#}
    by (simp add: mset-set-Diff)
also from i
have \( \{ \# \text{weak-ranking} (R x) \mid x \in \# \} = \{ \# \text{weak-ranking} (R x) \mid x \in \# \text{mset set agents#} \} - \{ \# \text{weak-ranking} (R i)\} \)
by (subst image-mset-Diff) (simp-all add: in-multiset-in-set mset-subset-eq-single)
also have \( \{ \# \text{weak-ranking} Ri'\} + \ldots = \text{anonymous-profile} R - \{ \# \text{weak-ranking} (R i)\} + \{ \# \text{weak-ranking} \}
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 × 'alt set list) list ⇒ ('agent, 'alt) pref-profile
where
\text{prefs-from-table} xss = (\lambda i. \text{case-option (} (\lambda- -. False) \text{ of-weak-ranking (} \text{map-of xss i) })
definition \text{prefs-from-table-wf} where
\text{prefs-from-table-wf} agents alts xss ←→ agents ≠ {} ∧ alts ≠ {} ∧ distinct (map fst xss) ∧
set (map fst xss) = agents ∧ (∀ xs ∈ set (map snd xss). (∪ set xs) = alts ∧ is-finite-weak-ranking xs)

lemma \text{prefs-from-table-wfI}:
assumes agents ≠ {} alts ≠ {} distinct (map fst xss)
assumes set (map fst xss) = agents
assumes (∀ xs. xs ∈ set (map snd xss) ⇒ (∪ set xs) = alts)
assumes (∀ xs. xs ∈ set (map snd xss) ⇒ is-finite-weak-ranking xs
shows \text{prefs-from-table-wf} agents alts xss
using 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 ≠ {} alts ≠ {} distinct (map fst xss)
and set (map fst xss) = agents
and (∀ xs. xs ∈ set (map snd xss) ⇒ (∪ set xs) = alts)
and (∀ xs. xs ∈ set (map snd xss) ⇒ is-finite-weak-ranking xs
using assms unfolding \text{prefs-from-table-wf-def} by auto

lemma \text{pref-profile-from-tableI}:
\text{prefs-from-table-wf} agents alts xss ⇒ \text{pref-profile-wf} agents alts (\text{prefs-from-table} xss)
proof (intro \text{pref-profile-wf.intro})
assume wf: \text{prefs-from-table-wf} agents alts xss
fix i assume i ∈ agents
with wf have i ∈ set (map fst xss) by (simp add: \text{prefs-from-table-wf-def})
then obtain xs where xs: xs ∈ set (map 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 \notin agents
  with wf have i \notin 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 \negprefs-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 \notin agents
  shows prefs-from-table xss i = (\lambda -. False)
proof --
  from assms have i \notin 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 \in 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 \in set (map fst xs)
  defines xs' \equiv map (\lambda(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 \mid k \notin set (map fst xs) \mid k \neq i \in 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: \( xs'\)-def split: if-splits)
ultimately show \( \text{thesis using } \text{assms}(1) \) \( k \) \( xs' \)
by (auto simp add: \( \text{prefs-from-table-def map-of-eq-None-iff dest!} \) \( \text{map-of-SoD split: option.splits} \))

next
assume \( k : k \notin \text{set } (\text{map fst } xs) \)
with \( \text{assms}(1) \) have \( k' : k \neq i \) by auto
with \( k \) \( xs' \) have \( \text{map-of } xs \) \( k = \text{None map-of } xs' \) \( k = \text{None} \)
by (simp-all add: 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 \( k(1) \) have \( \text{map-of } xs \) \( k = \text{map-of } xs' \) \( k \) unfolding \( xs'\)-def
by (induction \( xs \)) fastforce+
with \( k \) show \( \text{thesis by } (\text{simp add: } \text{prefs-from-table-def}) \)

qed

lemma \( \text{prefs-from-table-swap}: \)
\( x \neq y \implies \text{prefs-from-table } ((x,x')##(y,y')##xs) = \text{prefs-from-table } ((y,y')##(x,x')##xs) \)
by (intro ext) (auto simp: \( \text{prefs-from-table-def} \))

lemma \( \text{permute-prefs-from-table}: \)
assumes \( \sigma \) permutes \( \text{fst } \) set \( xs \)
sows \( \text{prefs-from-table } xs \circ \sigma = \text{prefs-from-table } (\text{map } (\lambda (x,y). (\text{inv } x, y))) \) \( xs \)
proof
fix \( i \)
have \( (\text{prefs-from-table xs } \circ \sigma) \) \( i = \)
\( \) \begin{itemize}
\item \( \text{case map-of } xs \) \( (\sigma \) \( i \) \( ) \) of
  \item \( \text{None } \Rightarrow \lambda - . \) \text{False}
  \item \( \text{Some } x \Rightarrow \text{of-weak-ranking } x \)
\end{itemize}
by (simp add: \( \text{prefs-from-table-def o-def} \))
also have \( \text{map-of } xs \) \( (\sigma \) \( i \) \( ) = \text{map-of} \) \( (\text{map } (\lambda (x,y). (\text{inv } x, y))) \) \( xs \) \( i \)
using \( \text{map-of-permute}[OF assms] \) by (simp add: o-def fun-eq-iff)
finally show \( (\text{prefs-from-table xs } \circ \sigma) \) \( i = \text{prefs-from-table } (\text{map } (\lambda (x,y). (\text{inv } x, y))) \) \( \) \begin{itemize}
\item \( \text{xs } i \) \( ) \)
\end{itemize}
by (simp only: \( \text{prefs-from-table-def} \))

qed

lemma \( \text{permute-profile-from-table}: \)
assumes \( \text{wf: } \text{prefs-from-table-wf agents alts xss} \)
assumes \( \text{perm: } \sigma \) permutes \( \text{alts} \)
sows \( \text{permute-profile } \sigma (\text{prefs-from-table xss}) = \)
\( \text{prefs-from-table } (\text{map } (\lambda (x,y). (x, \text{map } (((\ ) \sigma \)) \( y))) \) \( xss \)) \( (\) \begin{itemize}
\item \( \text{is } f = \) \( g \) \end{itemize}
proof
fix \( i \)
have $\text{wf}' : \text{prefs-from-table-wf agents alts} (\map{\lambda(x, y). (x, \map{(\cdot)^\sigma} y)} \text{xs})$

proof (intro \text{prefs-from-table-wfI}, goal-cases)

  case (5 \text{xs})
  then obtain $y$ where $y \in \text{set} \text{xs} \text{xs} = \map{(\cdot)^\sigma} (\text{snd} y)$
  by (auto simp add: o-def case-prod-unfold)
  with \text{assms} show \text{?case}:
  by (simp add: image-Union \text{[symmetric]} \text{prefs-from-table-wf-def} \text{permutes-image} o-def case-prod-unfold)

next

  case (6 \text{xs})
  then obtain $y$ where $y \in \text{set} \text{xs} \text{xs} = \map{(\cdot)^\sigma} (\text{snd} y)$
  by (auto simp add: o-def case-prod-unfold)
  with \text{assms} show \text{?case}:
  by (auto simp add: is-finite-weak-ranking-def is-weak-ranking-iff \text{prefs-from-table-wf-def} distinct-map \text{permutes-inj-on} \text{inj-on-image} intro \text{!}: disjoint-image)

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

show \text{?f i} = \text{?g i}
proof (cases \text{i} \in \text{agents})

assume \text{i} \notin \text{agents}
with \text{assms} \text{wf}' show \text{?thesis}:
by (simp add: \text{permute-profile-def} \text{prefs-from-table-undef})

next

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

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

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

have \text{xs-in-xss} : \text{xs} \in \text{snd set} \text{xss}
  using \text{xs} by (force dest!: map-of-SomeD)

with \text{wf} have \text{set-xss} : \bigcup \text{set} \text{xs} = \text{alts}
  by (simp add: \text{prefs-from-table-wfD})

from \text{i} have \text{prefs-from-table} (\map{\lambda(x,y). (x, \map{(\cdot)^\sigma} y)} \text{xs}) \text{i} = \text{of-weak-ranking} (\text{the} (\map{\lambda(x,y). (x, \map{(\cdot)^\sigma} y)} \text{xss}) \text{i})
  using \text{wf}' by (intro \text{prefs-from-table-map-of}) simp-all

also have \ldots = \text{of-weak-ranking} (\text{map} (\cdot)^\sigma \text{xs})
  by (subst \text{map-of-map}) (simp add: \text{xss})

also have \ldots = (\lambda a b. \text{of-weak-ranking} \text{xss} (\text{inv} \sigma a) (\text{inv} \sigma b))
  by (intro \text{ext}) (simp add: \text{of-weak-ranking-permute map-relation-def set-xss} \text{perm})

also have \ldots = \text{permute-profile} \sigma (\text{prefs-from-table \text{xss}}) \text{i}
  by (simp add: \text{prefs-from-table-def} \text{xs} \text{permute-profile-def})

finally show \text{?thesis} ..

qed
2.6 Automatic evaluation of preference profiles

lemma eval-prefs-from-table [simp]:
  prefs-from-table [] i = (λ- -. False)
 prefs-from-table ((i, y) ≠ xs) i = of-weak-ranking y
 i ≠ j ⇒ prefs-from-table ((j, y) ≠ xs) i = prefs-from-table xs i
 by (simp-all add: prefs-from-table-def)

lemma eval-of-weak-ranking [simp]:
a /∈ ⋃ set xs ⇒ ¬ of-weak-ranking xs a b
b /∈ x = ⇒ a /∈ ⋃ set (x # xs) = of-weak-ranking (x # xs) a b
b /∈ x = ⇒ of-weak-ranking (x # xs) a b ↔ of-weak-ranking xs a b
by (induction xs) (simp-all add: of-weak-ranking-Cons)

lemma prefs-from-table-cong [cong]:
assumes prefs-from-table xs = prefs-from-table ys
shows prefs-from-table (x # xs) = prefs-from-table (x # ys)
proof
fix i
show 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 xs x = ⋃ set (x # xs)
lemma eval-Collect-of-weak-ranking:
  Collect (of-weak-ranking xs x) = of-weak-ranking-Collect-ge (rev xs) x
by (simp add: of-weak-ranking-Collect-ge-def)

lemma of-weak-ranking-Collect-ge-empty [simp]:
of-weak-ranking-Collect-ge [] x = {}
by (simp add: of-weak-ranking-Collect-ge-def)

lemma of-weak-ranking-Collect-ge-Cons [simp]:
y /∈ x = ⇒ of-weak-ranking-Collect-ge (x # xs) y = (⋃ set (x # xs))
y /∈ x = ⇒ of-weak-ranking-Collect-ge (x # xs) y = 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':
of-weak-ranking-Collect-ge (x # xs) = (λy. (if y ∈ x then (⋃ set (x # xs)) else 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 prefs-from-table-wf agents alts xs
shows anonymous-profile (prefs-from-table xs) = mset (map snd xs)
proof
from assms interpret pref-profile-wf agents alts prefs-from-table xs
by \(\text{simp add: pref-profile-from-tableI}\)

from assms have agents: \(\text{agents} = \text{fst set} \text{ xs}\)
  by \(\text{simp add: prefs-from-table-wf-def}\)

hence \([\text{simp}]\): finite agents by auto

have anonymous-profile \((\text{prefs-from-table xs}) = \{
  \text{\#weak-ranking} \text{ (prefs-from-table xs i)}. \text{x} \in \# \text{ mset-set agents}\}\)
  by \(\text{simp add: a-def anonymous-profile-def}\)

also from assms have \(\ldots = \{\text{\#(map-of xs i)}. \text{i} \in \# \text{ mset-set agents}\}\)

proof (intro image-mset-cong)

fix \(i\)
assume \(i: \text{i} \in \# \text{ mset-set agents}\)

from \(i\) assms have weak-ranking \((\text{prefs-from-table xs i})\) = weak-ranking \((\text{of-weak-ranking} \text{ (the (map-of xs i))})\)
  by \(\text{simp add: prefs-from-table-map-of}\)

also from assms \(i\) have \(\ldots = \text{the (map-of xs i)}\)
  by (intro weak-ranking-of-weak-ranking)

(auto simp: prefs-from-table-wf-def)

finally show weak-ranking \((\text{prefs-from-table xs i})\) = \(\text{the (map-of xs i)}\).

qed

also from agents have mset-set agents = mset \((\text{set (map fst xs)})\)
  by simp

also from assms have \(\ldots = \text{mset (map fst xs)}\)
  by (intro mset-set-set simp: prefs-from-table-wf-def)

also from assms have \(\{\text{\#the (map-of xs i)}. \text{i} \in \# \text{ mset (map fst xs)}\}\) = mset \((\text{map snd xs})\)
  by (intro image-mset-map-of simp: prefs-from-table-wf-def)

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

qed

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

proof –

from \(\text{wf (1)}\) have agents: \(\text{agents} = \text{set (map fst xs)}\)
  by (simp-all add: prefs-from-table-wf-def)

from \(\text{wf (2)}\) have agents': \(\text{agents} = \text{set (map fst ys)}\)
  by (simp-all add: prefs-from-table-wf-def)

from agents agents' \(\text{wf (1)}\) \(\text{wf (2)}\) have \(\text{mset (map fst xs) = mset (map fst ys)}\)
  by (subst set-eq-iff-mset-eq-distinct [symmetric]) (simp-all add: prefs-from-table-wfD)

hence \(\text{same-length: length xs = length ys}\) by (auto dest: mset-eq-length simp del: mset-map)

from \(\text{mset (map fst xs) = mset (map fst ys)}\)
  obtain \(g\) where \(g\) permutes \(\ldots<\text{length ys}\) \(\text{permute-list g (map fst ys)} = \text{map fst xs}\)
    by (auto elim: mset-eq-permutation simp: same-length simp del: mset-map)

from mset-eq \(g\)
have mset (map snd ys) = mset (permute-list g (map snd ys)) by simp
with mset-eq obtain f
  where f: f permutes {..<length xs}
  permute-list f (permute-list g (map snd ys)) = map snd 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 < length xs \iff x < length xs
  f x < length ys \iff x < length ys for x by (simp-all add: same-length)

define idx unidx where idx = index (map fst xs) and unidx i = map fst xs ! i for i
from wf(1) have bij-betw idx agents {0..<length xs} unfolding idx-def
  by (intro bij-betw-index) (simp-all add: prefs-from-table-wf-def)
hence bij-betw-idx: bij-betw idx agents {..<length xs} by (simp add: atLeast0LessThan)
have [simp]: idx x < length xs if x \in agents for x
  using that by (simp add: idx-def agents)
have [simp]: unidx i \in agents if i < length xs for i
  using that by (simp add: agents unidx-def)

have unidx-idx: unidx (idx x) = x if x: x \in agents for x
  using x unfolding idx-def unidx-def using nth-index[of x map fst xs]
have idx-unidx: idx (unidx i) = i if i < length xs for i
  unfolding idx-def unidx-def using wf(1) index-nth-id[of map fst xs i] i
by (simp add: prefs-from-table-wfD(3))

define \pi where \pi x = (if x \in agents then (unidx o f o idx) x else x) for x
define \pi' where \pi' x = (if x \in agents then (unidx o inv f o idx) x else x) for x
have bij-betw (unidx o f o idx) agents agents (is ?P) unfolding unidx-def
  by (rule bij-betw-trans bij-betw-idx permutes-imp-bij f g bij-betw-nth)+
  (insert wf(1) g, simp-all add: prefs-from-table-wfD same-length)
also have ?P \iff bij-betw \pi agents agents
  by (intro bij-betw-conq) (simp add: \pi-def)
finally have perm: \pi permutes agents
  by (intro bij-imp-permutes) (simp-all add: \pi-def)

define h where h = g o f
from f g have h: h permutes {..<length ys} unfolding h-def
  by (intro permutes-compose) (simp-all add: same-length)

have inv-\pi: inv \pi = \pi'
proof (rule permutes-inv1[OF perm])
  fix x assume x: x \in agents
  with f(1) show \pi' (\pi x) = x
    by (simp add: \pi-def \pi'-def idx-unidx unidx-idx inv-f-f permutes-inj)
qed (simp add: \pi-def \pi'-def)
with perm have inv-\pi': inv \pi' = \pi by (auto simp: inv-inv-eq permutes-bij)

from wf h have prefs-from-table ys = prefs-from-table (permute-list h ys)
by (intro prefs-from-table-eql)  
(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 ... = π (unidx i) by (simp add: π-def idx-unidx)  
also from i have ... = map π (map fst xs) ! i by (simp add: unidx-def)  
finally have permute-list f (map fst xs) ! i = map π (map fst xs) ! i.  
}  
hence permute-list f (map fst xs) = map π (map fst xs)  
by (intro nth-equalityI) simp-all  
also have zip (map π (map fst xs)) (map snd xs) = map (λ(x,y). (inv π' x, y)) xs  
by (induction xs) (simp-all add: case-prod-unfold inv-π')  
also from permutes-inv[OF perm] inv-π have prefs-from-table ... = prefs-from-table xs ∘ π'  
by (intro permute-prefs-from-table [symmetric]) (simp-all add: agents)  
finally have prefs-from-table xs ∘ π' = prefs-from-table ys ..  
with that[of π'] permutes-inv[OF perm] inv-π show ?thesis by auto  
qed

lemma permute-list-distinct:  
assumes f' ![..<length xs] ⊆ {..<length xs} distinct xs  
shows permute-list f xs = map (λx. xs ![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 ∈ # mset-set A#] = ![#g x. x ∈ # mset-set A#] finite A  
obtains π where π permutes A ![x. x ∈ A ⇒ g (π 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 π where  
π: π permutes ![0..<length xs] permute-list π (map g xs) = map f xs  
by (auto simp: atLeast0LessThan)  
define π' where π' x = (if x ∈ A then (!xs o π o index xs) x else x) for x  
have bij-betw (!xs o π o index xs) A A (is ?P)  
by (rule bij-betw-trans bij-betw-index xs refl permutes-imp-bij π bij-betw-nth)+  
(simp-all add: atLeast0LessThan)

42
also have \( \pi' \longleftarrow \pi \ A \ A \)

by (intro bij-betw-cong) (simp-all add: \( \pi'\)-def)

finally have \( \pi' \) permutes \( A \)

by (rule bij-imp-permutes) (simp-all add: \( \pi'\)-def)

moreover from \( \pi \) \( xs(1) \)[symmetric] \( xs(2) \) have \( g(\pi' x) = f x \) if \( x \in A \) for \( x \)

by (simp add: permute-list-map permute-list-distinct permutes-image \( \pi'\)-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 \( \pi \) where \( \pi \) permutes agents \( R2 \circ \pi = R1 \)

proof –

interpret \( R1 \): pref-profile-wf agents alts \( R1 \) by fact

interpret \( R2 \): pref-profile-wf agents alts \( R2 \) by fact

from eq have \( \{\#weak-ranking (R1 x), x \in \# \ mset-set agents\#\} = \{\#weak-ranking (R2 x), x \in \# \ mset-set agents\#\} \)

by (simp add: \( R1\).anonymous-profile-def \( R2\).anonymous-profile-def o-def)

from image-mset-eq-permutation[OF this fin] guess \( \pi \) . note \( \pi = \) this

from \( \pi \) have \( \text{wf'}: \) pref-profile-wf agents alts \( (R2 \circ \pi) \)

by (intro \( R2\).wf-permute-agents)

then interpret \( R2': \) pref-profile-wf agents alts \( R2 \circ \pi \).

have \( R2 \circ \pi = R1 \)

proof (intro pref-profile-eqI[OF \( \text{wf'} \) \( \text{wf}(1) \)])

fix \( x \) assume \( x: x \in \) agents

with \( \pi \) have weak-ranking \( ((R2 \circ \pi) x) = \) weak-ranking \( (R1 x) \) by simp

with \( \text{wf'} \) \( \text{wf}(1) \) \( x \) show \( (R2 \circ \pi) x = R1 x \)

by (intro weak-ranking-eqD[of alts] \( R2'\).prefs-wf) simp-all

qed

from \( \pi(1) \) 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 \( \neq \) {}

assumes nonempty-alts [simp]: alts \( \neq \) {}
begin

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

lemma finite-total-preorder-on-iff' [simp]:
finite-total-preorder-on alts R ≫总额 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-pres-finite: finite (Max-wrt R)
unfolding R.Max-wrt-preorder by simp

lemma Max-wrt-pres-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

\section{Auxiliary facts about PMFs}

theory Lotteries
  imports Complex/Main HOL−Probability.Probability

44
The type of lotteries (a probability mass function)

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

**definition** lotteries-on :: 'a set ⇒ 'a lottery set

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]:

x ∈ A ⇒ return-pmf x ∈ lotteries-on A

by (simp add: lotteries-on-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** \(
\mathbb{E}(p) = \sum_{A \in \text{weak-ranking le}} \mathbb{E}(\text{SOME } x. x \in A) \times \text{measure-pmf.prob } p A
\)

**Proof** –

- **From** Assms **Have** \( \mathbb{E}(p) = \sum_{a \in \text{carrier}} u(a) \times \text{pmf } p a \)
  - By \( \text{subst integral-measure-pmf[of finite-carrier]} \)
    - (auto simp: lotteries-on-def ac-simps)
  - Also have \( \text{carrier} = \bigcup \{ A \in \text{set (weak-ranking le)} : A \in \text{set (weak-ranking le)} \} \)
    - By \( \text{simp add: weak-ranking-Union} \)
  - Also from \( \text{carrier} \) **Have** \( \text{finite} \)
    - If \( A \in \text{set (weak-ranking le)} \)
      - Using \( \text{by (rule someI-ex)} \)
        - (auto simp: weak-ranking-total-preorder)
    - Thus \( \sum_{a \in \bigcup \{ A \in \text{set (weak-ranking le)} \}} u(a) \times \text{pmf } p a = \sum_{A \in \text{weak-ranking le}} \sum_{a \in A} u(a) \times \text{pmf } p a \)
      - Using \( \text{is = sum-list ?xs} \)
        - (auto simp: is-weak-ranking-iff disjoint-def sum.distinct-set-conv-list)
    - Also have \( ?\text{xs} = \text{map } (\lambda A. \sum_{a \in A} u(a) \times \text{pmf } p a) \) (weak-ranking le)
      - Proof (intro map-cong HOL.refl sum.cong)
        - Fix \( x A \) **Assume** \( x \in A \) and \( A \in \text{set (weak-ranking le)} \)
        - From Assms \( \text{of } x A \)
          - By \( \text{rule someI-ex} \)
            - (auto simp: weak-ranking-total-preorder)
    - Thus \( u x \times \text{pmf } p x = \sum_{A \in \text{set (weak-ranking le)} \} \sum_{a \in A} u(a) \times \text{pmf } p a \times \text{pmf } p x \)
      - Qed
    - Also have \( \ldots = \text{map } (\lambda A. \sum_{a \in A} u(a) \times \text{pmf } p a) \) (weak-ranking le)
      - Using \( \text{finite by (intro map-cong HOL.refl)} \)
        - (auto simp: sum-distrib-left measure-measure-pmf-finite)
      - Finally show \( \text{?thesis} \)
      - Qed

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

**By** unfold-locals (insert utility-le-iff, auto)

**Lemma add-right:**

**Assumes** \( \forall x y. \text{le } x y \implies f x \leq f y \)

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

**Proof** –

- Fix \( x y \) **Assume** \( xy \) \( x \in \text{carrier } y \in \text{carrier} \)
  - From Assms[of x y] utility-le-iff[of xy] Assms[of x y] utility-le-iff[of xy(2,1)]
    - Show \( (u x + f x \leq u y + f y) = \text{le } x y \) **By** auto
    - Qed

**Lemma add-left:**

46
\[(\forall x \ y. \ le \ x \ y \implies f \ x \leq f \ y) \implies \text{vnm-utility carrier le} \ (\lambda x. \ f \ x + u \ x)\]

by (subst add.commute) (rule add-right)

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 \(A: \forall x \ y. \ le \ x \ y \implies le \ y \ x \implies f \ x = f \ y\)

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

**Proof:**

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

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

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

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

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

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

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: \(u \ x + \varepsilon \ast f \ x < u \ y + \varepsilon \ast f \ y\) if \(xy: x \prec[le] \ y\) 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 \(eq: u \ x + \varepsilon \ast f \ x = u \ y + \varepsilon \ast f \ y\) if \(xy: x \preceq[le] \ y \ y \preceq[le] \ x\) for \(x \ y\)

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

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

**Proof:**

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

show \((u \ x + \varepsilon \ast f \ x \leq u \ y + \varepsilon \ast f \ y) \iff le \ x \ y\)

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

by (cases \(le \ x \ y\); cases \(le \ y \ x\)) (auto simp: strongly-preferred-def)

**Qed**

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

47
lemma \( \text{diff-epsilon} \):
assumes \( \forall x, y. \text{le } x, y \Rightarrow \text{le } y, x \Rightarrow f x = f y \)
shows \( \exists \varepsilon > 0. \text{vnm-utility carrier } \text{le} (\lambda x. u x - \varepsilon \ast f x) \)
proof
from assms have \( \exists \varepsilon > 0. \text{vnm-utility carrier } \text{le} (\lambda x. u x + \varepsilon \ast -f x) \)
by (intro add-epsilon) (subst neg-equal-iff-equal)
thus ?thesis by simp
qed

4 Stochastic Dominance

theory Stochastic-Dominance
imports
  Complex-Main
  HOL-Probability.Probability
  Lotteries
  Preference-Profiles
  Utility-Functions
begin

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.

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

lemma \( SD\text{-empty} \): \( \text{SD } (\lambda- \cdot, \cdot) = (\lambda- \cdot, \cdot) \)
by (auto simp: fun-eq_iff SD-def lotteries-on_def set-pmf-not-empty)

Stochastic dominance over any relation is a preorder.

lemma \( SD\text{-refl} \): \( p \leq_{[SD(R)]} p \iff p \in \text{lotteries-on } \{ x. R x x \} \)
by (simp add: SD-def)

lemma \( SD\text{-trans} \): \( p \leq_{[SD(R)]} q \Rightarrow q \leq_{[SD(R)]} r \Rightarrow p \leq_{[SD(R)]} r \)
unfolding SD-def by (auto intro: order.trans)

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

begin

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

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

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

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

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

lemma SD-singleton-left:
assumes \( x \in \text{carrier} \land q \in \text{lotteries-on carrier} \)
shows \( \text{return-pmf} x \preceq [SD(le)] q \iff (\forall y \in \text{set-pmf} q. x \preceq [le] y) \)

proof
assume SD: \( \text{return-pmf} x \preceq [SD(le)] q \)
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 have \( \text{measure-pmf.prob} (\text{return-pmf} x) (\text{preferred-alts le} x) = 1 \)
by (simp add: indicator-def)
finally have \( AE y \in q. y \geq [le] x \)
by (simp add: measure-pmf.measure-ge-1-iff measure-pmf.prob-eq-1 preferred-alts-def)
thus \( \forall y \in \text{set-pmf} q. y \geq [le] x \)
by (simp add: AE.measure-pmf-iiff)
next
assume A: \( \forall y \in \text{set-pmf} q. x \preceq [le] y \)
show \( \text{return-pmf} x \preceq [SD(le)] q \)
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
proof (simp-all add: preferred-alts-def indicator-def measure-nonneg)
qed (insert assms, simp-all add: lotteries-on-def)

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[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
  qed (insert assms, simp-all add: lotteries-on-def)

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 $q \nsucc_{[SD(\leq)]} \ x \iff (\forall y \in \text{set-pmf} \ q. \ y \preceq_{\leq} x) \land (\exists y \in \text{set-pmf} \ q. \ y \prec_{[\leq]} x))$
using assms by (auto simp add: strongly-preferred-def SD-singleton-left SD-singleton-right)

lemma SD-singleton [simp]:
$x \in \text{carrier} \implies y \in \text{carrier} \implies \text{return-pmf} \ x \preceq_{[SD(\leq)]} \ y 
\leq_{[\leq]} x
by (\text{subst SD-singleton-left}) (\text{simp-all add: lotteries-on-def})

lemma SD-strict-singleton [simp]:
$x \in \text{carrier} \implies y \in \text{carrier} \implies \text{return-pmf} \ x \prec_{[SD(\leq)]} \ y 
\leq_{[\leq]} x
by (\text{simp add: strongly-preferred-def})

end

context \text{pref-profile-wf}
begin
context
fixes $i$ assumes $i : i \in \text{agents}$
begin
interpretation $R_i : \text{preorder-on alts} \ R \ i$ by (simp add: $i$)

lemmas SD-singleton-left = $R_i$.SD-singleton-left
lemmas SD-singleton-right = $R_i$.SD-singleton-right
lemmas SD-strict-singleton-left = $R_i$.SD-strict-singleton-left
lemmas SD-strict-singleton-right = $R_i$.SD-strict-singleton-right
lemmas SD-singleton = $R_i$.SD-singleton
lemmas SD-strict-singleton = $R_i$.SD-strict-singleton

end
end

lemmas (in \text{pref-profile-wf}) [simp] = SD-singleton SD-strict-singleton

4.2 Stochastic Dominance for preference profiles

context \text{pref-profile-wf}
begin

lemma SD-pref-profile:
assumes $i \in \text{agents}$
shows $p \nsucc_{[SD(R \ i)]} q \iff p \in \text{lotteries-on alts} \land q \in \text{lotteries-on alts} \land
(\forall x \in \text{alts}. \ \text{measure-pmf}. \ \text{prob} \ p \ (\text{preferred-alts} \ (R \ i) \ x) \geq
\text{measure-pmf}. \ \text{prob} \ q \ (\text{preferred-alts} \ (R \ i) \ x))$
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 notoutside
    by (auto simp: preferred-alts-def)
  thus \(\text{thesis by (simp add: SD-preorder preferred-alts-def)}\)
qed

lemma SD-pref-profileI [intro?]:
  assumes i ∈ agents p ∈ lotteries-on alts q ∈ lotteries-on alts
  assumes \(\bigwedge x. x \in \text{alts} \implies \text{measure-pmf.prob p (preferred-alts (R i) x) ≥ measure-pmf.prob q (preferred-alts (R i) x)}\)
  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 \(\bigwedge x. x \in \text{alts} \implies \text{measure-pmf.prob p (preferred-alts (R i) x) ≥ measure-pmf.prob q (preferred-alts (R i) x)}\)
  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 {x. ∃ i. R i x x}. q >\([\text{Pareto (SD \circ R)}] p\))

context pref-profile-wf
begin

sublocale SD: preorder-family agents lotteries-on alts SD \circ 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 >\([\text{Pareto (SD \circ R)}] p\))
proof
  have SD-efficient R p ←→ ¬(∃ q ∈ lotteries-on {x. ∃ i. R i x x}. q >\([\text{Pareto (SD \circ R)}] p\))
    unfolding SD-efficient-auxdef ..
  also from nonempty-agents obtain i where i: i ∈ agents by blast
with preorder-on.ryff[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 \succeq [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} q \succ [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 \succ [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} q \succ [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 \land 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
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 \( \text{thesis by blast} \)
qed

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 \( \equiv \) 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 \( (\text{real} (\text{length} (\text{weak-ranking } le) - \text{weak-ranking-index } x)) \leq \text{real} (\text{length} (\text{weak-ranking } le) - \text{weak-ranking-index } y) \) \( \longleftrightarrow \) \( le \ x \ y \)
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 \( p \preceq [\text{SD}(le)] \ q \longleftrightarrow \\
(\forall u. \text{is-vnm-utility } u \longrightarrow \text{measure-pmf.expectation } p \ u \leq \text{measure-pmf.expectation } q \ u) \)
proof safe
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: \( \text{is-weak-ranking } xs \leq \text{af-weak-ranking } xs \)
by (simp-all add: xs-def weak-ranking-total-preorder)
let \( ?\text{pref} = \lambda p x. \text{measure-pmf.prob } p \{ y. \ x \preceq[le] \ y \} \) and
\( ?\text{pref}' = \lambda p x. \text{measure-pmf.prob } p \{ y. \ x \prec[le] \ y \} \)
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 some1-ex[\( \lambda x. \ x \in \text{xs} ! i \)] (auto simp: set-cone-nth)
have \( f' \colon i \in \text{carrier} \) if \( i < \text{length} \text{xs} \) for \( i \)
  using that f weak-ranking-Union unfolding xs-def by (auto simp: set-cone-nth)
define \( n \) where \( n = \text{length} \text{xs} - 1 \)
from assms weak-ranking-Union have carrier-nonempty: carrier \( \neq \{\} \) and \( \text{xs} \neq [] \)
  by (auto simp: xs-def lotteries-on-def set-pmf-not-empty)
hence \( n \colon \text{length} \text{xs} = \text{Suc} n \) and \( \text{xs}-\text{nonempty} : \text{xs} \neq [] \) by (auto simp add: n-def)
have SD': \( \text{pref} p (f i) \leq \text{pref} q (f i) \) if \( i < \text{length} \text{xs} \) for \( i \)
  using f-le: le \( (f i) (f j) \) \( \rightarrow i \geq j \) if \( i < \text{length} \text{xs} j < \text{length} \text{xs} \) for \( i \)
  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!: of-weak-ranking.intros)

have measure-pmf.expectation p u =
  \( \sum_{i<n.} (u (f i) - u (f (\text{Suc} i))) * \text{pref} p (f i) + u (f n) \)
if \( p \colon p \in \text{lotteries-on} \text{carrier} \) for \( p \)
proof
  from \( p \) have measure-pmf.expectation p u =
    \( \sum_{i<\text{length} \text{xs}.} u (f i) * \text{measure-pmf.prob} p (\text{xs} (! i)) \)
    by (simp add: f-def expected-utility-weak-ranking xs-def sum-list-sum-nth
      atLeast0LessThan)
  also have \( \ldots = (\sum_{i<\text{length} \text{xs}.} u (f i) * (\text{pref} p (f i) - ?\text{pref}' p (f i))) \)
  proof (intro sum.cong HOL.refl)
    fix \( i \) assume \( i \colon i \in \{..<\text{length} \text{xs}\} \)
    have \( ?\text{pref} p (f i) - ?\text{pref}' p (f i) = \)
      measure-pmf.prob p \( \{y. \ f i \leq [\le] \ y\} - \{y. \ f i \prec [\le] \ y\} \)
      by (auto simp: strongly-preferred-def)
    also have \( \{y. \ f i \leq [\le] \ y\} - \{y. f i \prec [\le] \ y\} = \)
      \{y. \ f i \leq [\le] \ y \land y \leq [\le] f i\} (is - = ?A)
      by (auto simp: strongly-preferred-def)
    also have \( \ldots = \text{xs} ! i \)
    proof safe
      fix \( x \) assume \( le: le \ (f i) x \) le \( f i \)
      from \( i \) \( f \) show \( x \in \text{xs} ! i \)
        by (intro weak-ranking-eqclass2[OF - - le]) (auto simp: xs-def)
    next
      fix \( x \) assume \( x \in \text{xs} ! i \)
      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 \( u (f i) * \text{measure-pmf.prob} p (\text{xs} ! i) = \)
      \( u (f i) * (\text{pref} p (f i) - ?\text{pref}' p (f i)) \) by simp
    qed
  also have \( \ldots = (\sum_{i<\text{length} \text{xs}.} u (f i) * (\text{pref} p (f i)) - \)
    \( (\sum_{i<\text{length} \text{xs}.} u (f i) * ?\text{pref}' p (f i)) \)
\begin{verbatim}
by (simp add: sum-subtractf ring-distrib)
also have \((\sum i<\text{length }xs. u\ (f\ i) \ast \ ?\text{pref}\ p\ (f\ i)) = \)
\((\sum i<n. u\ (f\ i) \ast \ ?\text{pref}\ p\ (f\ i)) + u\ (f\ n) \ast \ ?\text{pref}\ p\ (f\ n)\) \(\text{(is - =}
\\text{?sum})\)
by (simp add: n)
also have \((\sum i<\text{length }xs. u\ (f\ i) \ast \ ?\text{pref}'\ p\ (f\ i)) = \)
\((\sum i<n. u\ (f\ (\text{Suc}\ i)) \ast \ ?\text{pref}'\ p\ (f\ (\text{Suc}\ i))) + u\ (f\ 0) \ast \ ?\text{pref}'\ p\ (f\ 0)\)
unfolding \(n\ \text{sum-lessThan-Suc-shift}\) by simp
also have \((\sum i<n. u\ (f\ (\text{Suc}\ i)) \ast \ ?\text{pref}'\ p\ (f\ (\text{Suc}\ i))) = \)
\((\sum i<n. u\ (f\ (\text{Suc}\ i)) \ast \ ?\text{pref}\ p\ (f\ (\text{Suc}\ i))))\)
proof (intro sum.cong HOL.refl)
fix \(i\) assume \(i: i \in \{..<n\}\)
have \((\text{Suc}\ i) \prec\lceil[le]\ y \longleftrightarrow f\ i \preceq\lceil[le]\ y\) for \(y\)
proof (cases \(y \in\) carrier)
assume \(y \in\) carrier
with \(\text{weak-ranking-Union}\) obtain \(j\) where \(j: j < \text{length }xs\ y \in xs\ !\ j\)
by (auto simp: set-conv-nth xs-def)
with \(\text{weak-ranking-eqclass1}[OF - f\ j(2)]\) weak-ranking-eqclass1[\(\text{OF - j(2)}\) \(f\)]
have \(\text{iff: le y z} \longleftrightarrow le\ (f\ j)\ z\) \(\le\ z\ y \longleftrightarrow le\ z\ (f\ j)\) \text{for}\ \(z\)
by (auto intro: trans simp: xs-def)
thus \(\text{thesis using}\ i\ j\ unfolding\ n-def\)
by (auto simp: iff-\(\prec\) strongly-preferred-def)
qed (insert not-outside, auto simp: strongly-preferred-def)
thus \(u\ (f\ (\text{Suc}\ i)) \ast \ ?\text{pref}'\ p\ (f\ (\text{Suc}\ i)) = u\ (f\ (\text{Suc}\ i)) \ast \ ?\text{pref}\ p\ (f\ i)\) by simp
qed
also have \(\text{?sum - (.. + u}\ (f\ 0) \ast \ ?\text{pref}'\ p\ (f\ 0)) = \)
\((\sum i<n. (u\ (f\ i) - u\ (f\ (\text{Suc}\ i))) \ast ?\text{pref}\ p\ (f\ i)) - \)
\(u\ (f\ 0) \ast \ ?\text{pref}'\ p\ (f\ 0) + u\ (f\ n) \ast \ ?\text{pref}\ p\ (f\ n)\)
by (simp add: ring-distrib)
also have \(\{y. f\ 0 \prec\lceil[le]\ y\} = \{}\)
using \(\text{hd-weak-ranking[of } f\ 0\]\) \(\text{xs-nonempty } f\) \(\text{not-outside}\)
by (auto simp: hd-conv-nth xs-def strongly-preferred-def)
also have \(\{y. \le\ (f\ n)\ y\} = \) \text{carrier}\nusing \(\text{last-weak-ranking[of } f\ n\]\) \(\text{xs-nonempty } f\) \(\text{not-outside}\)
by (auto simp: last-conv-nth xs-def n-def)
also from \(p\) have \(\text{measure-pmf.prob } p\ \text{carrier} = 1\)
by (subst measure-pmf.prob-eq-1)
(auto simp: \(\text{AE-measure-pmf-iff}\) \(\text{lotteries-on-def}\))
finally show \(\text{thesis by simp}\)
qed
from \(\text{assms}[\text{THEN this}]\) show \(\text{measure-pmf.expectation } p\ u \le\ \text{measure-pmf.expectation } q\ u\)
unfolding \(\text{SD-preorder } n\-def\) using \(f'\)
by (auto intro!: \(\text{sum-mono}\) \(\text{mult-left-mono}\) \(\text{SD'}\) simp: utility-le-iff f-le)
next
\end{verbatim}
\textbf{assume} \( \forall u. \text{is-vnm-utility } u \longrightarrow \text{measure-pmf.expectation } p \ u \leq \text{measure-pmf.expectation } q \ u \)

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

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

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

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

\textbf{by} \( \left( \text{simp-all add: } xs\text{-def weak-ranking-total-total-preorder} \right) \)

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

\textbf{by} \( \left( \text{simp add: } xs\text{-def weak-ranking-Union} \right) \)

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

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

\{ 

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

\textbf{let} \( \lambda idx = \lambda y. \text{length } xs - \text{weak-ranking-index } y \)

\textbf{have} \( \text{preferred-subset-carrier: } \{ y. \ le \ x \ y \} \subseteq \text{carrier} \)

\textbf{using} \( \text{not-outside } x \ \text{by auto} \)

\textbf{have} \( \text{measure-pmf.prob } p \ \{ y. \ le \ x \ y \} / \text{real (length } xs) \leq \text{measure-pmf.prob } q \ \{ y. \ le \ x \ y \} / \text{real (length } xs) \)

\textbf{proof} \( \left( \text{rule field-le-epsilon} \right) \)

\textbf{fix} \( \epsilon :: \text{real assume } \epsilon: \epsilon > 0 \)

\textbf{define} \( u \ \text{where } u = \text{indicator } \{ y. \ y \geq [le] x \} \ y + \epsilon * \lambda idx y \ \text{for } y \)

\textbf{have} \( \text{is-utility: is-vnm-utility } u \ \text{unfolding } u\text{-def } xs\text{-def} \)

\textbf{proof} \( \left( \text{intro vnm-utility.add-left vnm-utility.scaled utility-weak-ranking-index} \right) \)

\textbf{fix} \( y \ z \ \text{assume } le \ y \ z \)

\textbf{thus} \( \text{indicator } \{ y. \ y \geq [le] x \} \ y \leq (\text{indicator } \{ y. \ y \geq [le] x \} \ z :: \text{real}) \)

\textbf{by} \( \left( \text{auto intro: } \text{trans simp: } \text{indicator-def split: if-splits} \right) \)

\textbf{qed} \( \text{fact+} \)

\textbf{have} \( (\sum y\{le \ x \ y. \ \text{pmf } p \ y\} \leq \ (\sum y\{le \ x \ y. \ \text{pmf } p \ y\} + \epsilon * (\sum y\in \text{carrier}. \ ?idx y * \text{pmf } p \ y) \)

\textbf{using} \( \epsilon \ \text{by} \ \left( \text{auto intro: } \text{multi-nonneg-nonneg sum-nonneg pmf\text{-nonneg}} \right) \)

\textbf{also from} \( \text{expected-utility-le is-utility have} \)

\textbf{measure-pmf.expectation } p \ u \leq \text{measure-pmf.expectation } q \ u . \)

\textbf{with} \( \text{assms} \)

\textbf{have} \( (\sum a\in \text{carrier}. \ a \ast \text{pmf } p \ a) \leq (\sum a\in \text{carrier}. \ a \ast \text{pmf } q \ a) \)

\textbf{by} \( \left( \text{subst } (\text{asm}) \ (1 2) \text{ integral-measure-pmf[OF finite-carrier]} \right) \)

\textbf{(auto simp: } \text{lotteries-on-def set-pmf-eq ac-simps) \)

\textbf{hence} \( (\sum y\{le \ x \ y. \ \text{pmf } p \ y\} + \epsilon * (\sum y\in \text{carrier}. \ ?idx y * \text{pmf } p \ y) \leq \ (\sum y\{le \ x \ y. \ \text{pmf } q \ y\} + \epsilon * (\sum y\in \text{carrier}. \ ?idx y * \text{pmf } q \ y) \)

\textbf{using} \( \text{x preferred-subset-carrier unfolding } u\text{-def} \)

\textbf{by} \( \left( \text{simp add: } \text{sum.distrib finite-carrier algebra-simps Int-absorb1 sum-distrib-left} \right) \)

\textbf{also have} \( (\sum y\in \text{carrier}. \ ?idx y * \text{pmf } q \ y) \leq (\sum y\in \text{carrier}. \ \text{length } xs * \text{pmf } q \ y) \)

\textbf{by} \( \left( \text{intro sum-mono multi-right-mono} \right) (\text{simp-all add: pmf\text{-nonneg}}) \)

\textbf{also have} \( \ldots \ = \text{measure-pmf.expectation } q \ (\lambda -. \ \text{length } xs) \)

\textbf{using} \( \text{assms by} \ \left( \text{subst integral-measure-pmf[OF finite-carrier]} \right) \)

\textbf{(auto simp: } \text{lotteries-on-def set-pmf-eq ac-simps) \)

57
also have \( \ldots = \text{length } xs \) by simp
also have \( (\sum y \mid \leq x y. \text{pmf } p y) = \text{measure-pmf.prob } p \{ y. \leq x y \} \)
using finite-subset[OF preferred-subset-carrier finite-carrier]
by (simp add: measure-measure-pmf-finite)
also have \( (\sum y \mid \leq x y. \text{pmf } q y) = \text{measure-pmf.prob } q \{ y. \leq x y \} \)
using finite-subset[OF preferred-subset-carrier finite-carrier]
by (simp add: measure-measure-pmf-finite)
finally show \( \text{measure-pmf.prob } p \{ y. \leq x y \} / \text{length } xs \leq \text{measure-pmf.prob } q \{ y. \leq x y \} / \text{length } xs + \varepsilon \)
using \(\varepsilon\) by (simp add: divide-simps)
qed
moreover from carrier-nonempty carrier have \( xs \neq [] \) by auto
ultimately have \( \text{measure-pmf.prob } p \{ y. \leq x y \} \leq \text{measure-pmf.prob } q \{ y. \leq x y \} \)
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 \( \neg(p \prec_{[SD(\leq)]} q) \iff (\exists u. \text{is-vnm-utility } u \land \text{measure-pmf.expectation } u \leq \text{measure-pmf.expectation } q u) \)
proof
let \( ?E = \text{measure-pmf.expectation} :: 'a \text{pmf} \Rightarrow - \Rightarrow \text{real} \)
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 \( u: \text{vnm-utility carrier } le u \)
show \( \neg p \prec_{[SD \leq]} q \)
proof
assume less: \( p \prec_{[SD \leq]} q \)
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-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 \( u': \text{vnm-utility carrier } le u' \)
have \( \exists \varepsilon > 0. \text{is-vnm-utility } (\lambda x. u x - \varepsilon \ast u' x) \)
by (intro u.diff-epsilon antisym u'.utility-le)
then guess \( \varepsilon \) by (elim exE conjE) note \( \varepsilon = \) this
define \( u'' \) where \( u'' x = u x - \varepsilon \ast u' x \) for \( x \)
interpret \( u'': \text{vnm-utility carrier } le u'' \) unfolding u''-def by fact
have exp-u'': \( ?E p u'' = ?E p u - \varepsilon \ast ?E p u' \) if \( p \in \text{lotteries-on carrier} \) for \( p \)
using that
by (subst (1 2 3) integral-measure-pmf[of carrier])

58
(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 pq[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 (\( p \prec_{SD (le)} q \)) ←→ (\( \forall u.\ is-vnm-utility u \longrightarrow measure-pmf.expectation p u < measure-pmf.expectation q u \))
  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: \( \neg SD-efficient R p' \)
  assumes support: set-pmf p' ⊆ set-pmf p
  assumes lotteries: p ∈ lotteries-on alts
  shows \( \neg SD-efficient R p \)

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
    \( \bigwedge \ i. \ i \in \ agents \ \Rightarrow \ q' \succeq_{SD(R i)} p' \) \( q' \succ_{SD(R i)} p' \)
    unfolding SD-efficient-def' by blast

  have subset: \( \{x. pmf p' x > pmf q' x\} \subseteq \set-pmf p' \) by (auto simp: set-pmf-eq)
  also have \( \ldots \subseteq \set-pmf p \) by fact
  also have \( \ldots \subseteq \alts \) 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 > pmf q' x\})

end

59
define $\text{supp}$ where $\text{supp} = \text{set-pmf} \ p \cup \text{set-pmf} \ q$
from lotteries finite-alts $q(1)$ have $\text{finite-supp} \ : \ \text{finite supp}$
  by (auto simp: lotteries-on-def supp-def dest: finite-subset)
from support have [simp]: $\text{pmf} \ p \ x = 0 \ \text{pmf} \ p' \ x = 0 \ \text{pmf} \ q' \ x = 0$ if $x \notin \text{supp}$
for $x$
  using that by (auto simp: supp-def set-pmf-eq)

from finite support subset have $\varepsilon: \varepsilon > 0$ unfolding $\varepsilon$-def
  by (auto simp: field-simps set-pmf-eq)
have nonneg: $\text{pmf} \ p \ x + \varepsilon \cdot (\text{pmf} \ q' \ x - \text{pmf} \ p' \ x) \geq 0$ for $x$
proof (cases $\text{pmf} \ p' \ x > \text{pmf} \ q' \ x$
  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 $\forall \text{thesis by} \ (\text{simp add: field-simps})$
next
  case False
  with pmf-nonneg[of $p \ x$] $\varepsilon$ show $\forall \text{thesis by} \ \text{simp}$
qed

define $q$ where $q = \text{embed-pmf} \ (\lambda x. \ \text{pmf} \ p \ x + \varepsilon \cdot (\text{pmf} \ q' \ x - \text{pmf} \ p' \ x))$

proof (subst $nn$-integral-count-space)
  have $(\sum x \in \text{supp}. \ \text{ennreal} \ (\text{pmf} \ p \ x + \varepsilon \cdot (\text{pmf} \ q' \ x - \text{pmf} \ p' \ x))) = \text{ennreal} ((\sum x \in \text{supp}. \ \text{pmf} \ p \ x) + \varepsilon \cdot ((\sum x \in \text{supp}. \ \text{pmf} \ q' \ x) - (\sum x \in \text{supp}. \ \text{pmf} \ p' \ x)))$
    by (subst $\text{sum-ennreal}[(\text{OF nonneg}], \ \text{rule ennreal-cong})$
      (auto simp: $\text{sum-subtractf ring-distributes sum-distrib}$)
  also from finite-supp support have $\ldots = 1$
    by (subst $(1 \ 2 \ 3)$ sum-pmf-eq-1) (auto simp: supp-def)
  finally show $(\sum x \in \text{supp}. \ \text{ennreal} \ (\text{pmf} \ p \ x + \varepsilon \cdot (\text{pmf} \ q' \ x - \text{pmf} \ p' \ x))) = 1.$
qed (insert nonneg finite-supp, simp-all)

with nonneg have pmf-$q$: $\text{pmf} \ q \ x = \text{pmf} \ p \ x + \varepsilon \cdot (\text{pmf} \ q' \ x - \text{pmf} \ p' \ x)$ 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-mf: $q \in \text{lotteries-on alts}$
  by (auto simp add: lotteries-on-def supp-def)

from support-$q$ support have expected-utility:
  measure-pmf.expectation $q \ u = \text{measure-pmf.expectation} \ p \ u + \varepsilon \cdot \text{measure-pmf.expectation} \ q' \ u - \text{measure-pmf.expectation} \ p' \ u$ 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 $q \geq [\text{SD}(R \ i)]$ $p$ if $i: i \in \text{agents for} \ i$
proof
  from i interpret finite-total-preorder-on alts R i by simp
  from i lotteries q'(1) q'(3) [OF i] q-wf p'\text{-}wf \in show \ ?thesis
  by (fastforce simp: SD-iff-expected-utilities-len expected-utility)
qed
moreover from \langle i \in \text{agents} \rangle interpret finite-total-preorder-on alts R i by simp
  from lotteries q'(1,4) q-wf p'\text{-}wf \in have q \succ [SD(R i)] p
  by (force simp: SD-iff-expected-utilities-len expected-utility not-le strongly-preferred-def)
ultimately show \ ?thesis using q-wf \langle i \in \text{agents} \rangle unfolding SD-efficient-def'
by blast
qed

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

definition SD-efficient-same-support:
  assumes set-pmf p = set-pmf p' p \in lotteries-on alts
  shows SD-efficient R p \longleftrightarrow SD-efficient R p'
using SD-inefficient-support-subset[OF p p'] SD-inefficient-support-subset[OF p p]
assms
by (auto simp: lotteries-on-def)

definition SD-efficient-iff:
  assumes p \in lotteries-on alts
  shows SD-efficient R p \longleftrightarrow SD-efficient R (pmf-of-set (set-pmf p))
using assms finite-alts
by (intro SD-efficient-same-support)

definition SD-efficient-no-pareto-loser:
  assumes efficient: SD-efficient R p and p-wf: p \in lotteries-on alts
  shows set-pmf p \cap pareto-losers R = {}
proof
  have x \notin pareto-losers R if x: x \in set-pmf p for x
  proof
    from x have set-pmf (return-pmf x) \subseteq set-pmf p by auto
    from efficient this p-wf have SD-efficient R (return-pmf x)
    by (rule SD-efficient-support-subset)
    moreover from assms x have x \in alts by (auto simp: lotteries-on-def)
    ultimately show x \notin pareto-losers R by (simp add: SD-efficient-singleton-iff)
  qed
  thus \ ?thesis 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 $p \in \text{lottteries-on alts} \ q \in \text{lottteries-on alts} \ q \succ [\text{Pareto}(SD \circ R)] \ p$

set-$\text{pmf} \ p = \text{set-pmf} \ q$

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

**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 \ldots \subseteq \text{alts} using \text{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 \text{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 $\varepsilon = \text{Min} \{ \text{pmf} \ p \ x / (\text{pmf} \ q \ x - \text{pmf} \ p \ x) | x. \ \text{pmf} \ p \ x > \text{pmf} \ q \ x \}$

define $\text{supp}$ where $\text{supp} = \text{set-pmf} \ p$

from \text{assms finite-alts} have finite-$\text{supp}$: finite $\text{supp}$

by (auto simp: lotteries-on-def $\text{supp-def}$ dest: finite-subset)

from \text{assms} have [simp]: $\text{pmf} \ p \ x = 0 \ \text{pmf} \ q \ x = 0$ if $x \notin \text{supp}$ for $x$

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

proof (cases $\text{pmf} \ p \ x > \text{pmf} \ q \ x$)

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

have $(\int x. \text{ennreal} (\text{pmf} \ p \ x + \varepsilon * (\text{pmf} \ q \ x - \text{pmf} \ p \ x)) \ 0 \ 0)$

= 1

proof (subst nn-integral-count-space)

  have $(\sum x \in \text{supp. ennreal} (\text{pmf} \ p \ x + \varepsilon * (\text{pmf} \ q \ x - \text{pmf} \ p \ x)))$

  = ennreal $(\sum x \in \text{supp. pmf} \ p \ x + \varepsilon * ((\sum x \in \text{supp. pmf} \ q \ x) - (\sum x \in \text{supp. pmf} \ p \ x)))$

  by (subst sum ennreal[OF nonneg], rule ennreal-cong)

  (auto simp: sum-subtractf ring-distribs sum.distrib sum-distrib-left)

also from finite-$\text{supp}$ have \ldots = 1

by (subst (1 2 3) sum-pmf-eq-1) (auto simp: supp-def \text{assms})
finally show \((\sum_{x \in \text{supp. ennreal}} (\text{pmf } p \, x + \varepsilon \times (\text{pmf } q \, x - \text{pmf } p \, x))) = 1\).

qed (insert nonneg finite-suppm, simp-all)

with nonneg have pmf-r: \(\text{pmf } r \, x = \text{pmf } p \, x + \varepsilon \times (\text{pmf } q \, x - \text{pmf } p \, x)\) for \(x\)

unfolding \text{r-def} by (intro pmf-embed-pmf) simp-all

with assms have set-pmf r \(\subseteq\) supp

unfolding \text{supp-def} by (auto simp: set-pmf-eq)

from finite \text{ex-less} have \(\varepsilon \in \{\text{pmf } p \, x / \text{pmf } p \, x - \text{pmf } q \, x\} \mid x. \text{pmf } p \, x > \text{pmf } q \, x\)

unfolding \(\varepsilon\)-def by (intro Min-in) auto

then obtain \(x\) where \(\varepsilon = \text{pmf } p \, x / (\text{pmf } p \, x - \text{pmf } q \, x)\) \(\text{pmf } p \, x > \text{pmf } q \, x\)

by blast

hence \(\text{pmf } r \, x = 0\) by (simp add: pmf-r field-simps)

moreover from \(\text{pmf } p \, x > \text{pmf } q \, x\)

have \(\text{pmf } p \, x > 0\) by linarith

ultimately have \(x \in \text{set-pmf } p - \text{set-pmf } r\) by (auto simp: set-pmf-iff)

with \(\text{set-pmf } r \subseteq \text{supp}\) have support-r: \(\text{set-pmf } r \subset \text{set-pmf } p\)

unfolding \text{supp-def} by blast

from this assms have \(\text{r-wf}: r \in \text{lotteries-on alts}\) by (simp add: lotteries-on-def)

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

proof

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

then interpret \(\text{finite-total-preorder-on alts } R \, i\) by simp

show \(r \geq [SD(R \, i)] q\)

proof (subst SD-iff-expected-utilities-le; safe?)

fix \(u\) assume \(u: \text{is-vnm-utility } u\)

from support-r have expected-utility-r:

\[
\text{measure-pmf}.\text{expectation } r \, u = \text{measure-pmf}.\text{expectation } p \, u + \varepsilon \times (\text{measure-pmf}.\text{expectation } q \, u - \text{measure-pmf}.\text{expectation } p \, u)
\]

by (subst (1 2 3 4) integral-measure-pmf[\text{OF finite-suppm}])

(auto simp: supp-def assms pmf-r sum-distrib sum-distrib-left
sum-distrib-right sum-subtractf algebra-simps)

from assms \(i\) have \(q \geq [SD(R \, i)] p\) by (simp add: \text{SD-Pareto-strict iff})

with assms \(u\) have \(\text{measure-pmf}.\text{expectation } q \, u \geq \text{measure-pmf}.\text{expectation } p \, u\)

by (simp add: \text{SD-iff-expected-utilities-le r-wf})

hence \((\varepsilon - 1) \times \text{measure-pmf}.\text{expectation } p \, u \leq (\varepsilon - 1) \times \text{measure-pmf}.\text{expectation } q \, u\)

using \(\varepsilon\) by (intro mult-left-mono) simp-all

thus \(\text{measure-pmf}.\text{expectation } q \, u \leq \text{measure-pmf}.\text{expectation } r \, u\)

by (simp add: algebra-simps expected-utility-r)

qed fact+

qed

from that[\text{OF r-wf this support-r}] show \(\text{?thesis}\).

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′. 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-aldef: 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-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 finiteE-min[OF wf nonempty']) 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-aldef]
  show r ∈ lotteries-on alts r ≻ [Pareto(SD ◦ R)] q
  (∀ r′. 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
private lemma sd-chain-None-propagate:
  \( m \geq n \rightarrow \text{sd-chain } n = \text{None} \rightarrow \text{sd-chain } m = \text{None} \)
  by (induction rule: inc-induct) simp-all

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

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

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

private lemma sd-chain-Suc:
  assumes \( \text{sd-chain } m = \text{Some } q \)
  assumes \( \text{sd-chain} (\text{Suc } m) = \text{Some } r \)
  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:
  assumes \( m < n \)
  assumes \( \text{sd-chain } m = \text{Some } q \)
  assumes \( \text{sd-chain } n = \text{Some } s \)
  shows \( q \preceq [\text{Pareto}(\text{SD} \circ R)] s \)
  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 \( \text{Suc } k \leq n \) by simp
  from sd-chain-Some-propagate[of this, of s] step.prems obtain r
  where r: \( \text{sd-chain } (\text{Suc } k) = \text{Some } r \)
  by (auto simp del: sd-chain.simps)
  moreover from r step.prems have \( r \prec [\text{Pareto}(\text{SD} \circ R)] s \)
  by (intro step.IH)
  ultimately show ?case by (rule SD.Pareto.strict-trans)
  qed

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

65
case True
from sd-chain-strictly-preferred[OF this assms\{2,3\}] show \( ?\text{thesis} \)
by (simp add: strongly-preferred-def)

next

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

qed

lemma SD-efficient-lottery-exists:
obtains \( q \) where \( q \in \text{lotteries-on alts} \) \( q \succeq \text{Pareto}(SD\circ R) \) \( p \) SD-efficient \( R \) \( q \)

proof –

case 1

fix \( n \) \( k \) show \( ?\text{thesis} \)
by (auto)

next

case 2

show \( ?\text{thesis} \)
by (simp_all)

qed
\[-(k > m)}) \neq \{}\]
  by (intro notI) simp-all
then obtain \(n\) where \(mn: n > m\) 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: \(p \prec [Pareto(SEDoR)] q\) by (rule sd-chain-strictly-preferred)
from \((m < n)\) have \(n > 0\) by simp
with \(sd-chain n = Some q\); sd-chain.simps(2)[of \(n - 1\)]
obtain \(r\) where \(r: \sim SD-efficient R q = improve-lottery r\)
by (auto simp del: sd-chain.simps split: if-splits option.splits)
from \(pq\) have \(p \in lotteries-on\) \(alts\) \(q \in lotteries-on\) \(alts\)
by (simp-all add: sd-chain-lottery)
from improve-lottery-support-subset[OF this less supp-eq] guess \(s\). note \(s =\) this
from improve-lottery(2)[of \(r\)] \(r s\) have \(s > [Pareto(SEDoR)] r\)
by (auto intro: SD.Pareto.strict-weak-trans)
from improve-lottery(3)[OF \(r(1)\) \(s(1)\) this] supp-eq \(r\)
  have \(\sim set-pmf s \subset set-pmf p\) by simp
with \(s(3)\) show \(?\)thesis by contradiction
qed
qed

end

lemma
assumes \(p \in lotteries-on\) \(alts\)
shows \(\exists q \in \) lotteries-on\(\) \(alts,\) \(q \geq [Pareto(SEDoR)]\) \(p \land SD-efficient R q\)
using SD-efficient-lottery-exists[OF assms] by blast

end

end

5 Social Decision Schemes

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

\[
\text{lotteries} \equiv \text{lotteries-on alts}
\]

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

abbreviation **lottery-prob** :: 'alt lottery \(\Rightarrow\) 'alt set \(\Rightarrow\) real where

\[
\text{lottery-prob} \equiv \text{measure-pmf}.\text{prob}
\]

lemma **lottery-prob-alts-superset**:  
assumes \(p \in \text{lotteries alts} \subseteq A\)  
shows \(\text{lottery-prob} p A = 1\)  
using assms by (subst measure-pmf.prob-eq-1) (auto simp: AE-measure-pmf-iff lotteries-on-def)

lemma **lottery-prob-alts**: \(p \in \text{lotteries} \implies \text{lottery-prob} p \text{ alts} = 1\)  
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 \(\Rightarrow\) 'alt lottery  
assumes sds-wf: is-pref-profile R \(\Rightarrow\) sds R \(\in\) 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: \(\pi\) permutes agents \(\Rightarrow\) is-pref-profile R \(\Rightarrow\) sds \((R \circ \pi) = sds R\)
lemma anonymity-prefs-from-table:
  assumes prefs-from-table-wf agents alts xs prefs-from-table-wf agents alts ys
  assumes mset (map snd xs) = mset (map snd ys)
  shows sds (prefs-from-table xs) = sds (prefs-from-table ys)
proof
  with anonymous[of π, 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 R1 = prefs-from-table xs prefs-from-table-wf agents alts xs
  assumes R2 = prefs-from-table ys prefs-from-table-wf agents alts ys
  assumes mset (map snd xs) = mset (map snd ys)
  shows sds R1 = sds R2 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: σ permutes alts ⇒ is-pref-profile R ⇒
    sds (permute-profile σ R) = map-pmf σ (sds R)
begin
Alternative formulation of neutrality that shows that our definition is equivalent to that in the paper.

lemma neutral’:
  assumes σ permutes alts
  assumes is-pref-profile R
  assumes a ∈ alts
  shows pmf (sds (permute-profile σ R)) (σ a) = pmf (sds R) a
proof
  from assms have A: set-pmf (sds R) ⊆ alts using sds-uf
    by (simp add: lotteries-on-def)
  from assms(1,2) have pmf (sds (permute-profile σ R)) (σ a) = pmf (map-pmf σ (sds R)) (σ a)
    by (subst neutral) simp-all
  also from assms have ... = pmf (sds R) a
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)
  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 perm: σ permutes alts and wf: is-pref-profile R
assumes eq: image-mset (map ((‘) σ)) (anonymous-profile R) = anonymous-profile R
shows map-pmf σ (sds R) = sds R
using sds-anonymous-neutral[OF perm wf wf eq [symmetric]] ..

end

lemma an-sds-automorphism-aux:
assumes wf: prefs-from-table-wf agents alts yss R ≡ prefs-from-table yss
assumes an: an-sds agents alts sds
assumes eq: mset (map ((‘)) (permutation-of-list xs)) ○ snd) yss = mset (map snd yss)
assumes perm: set (map fst xs) ⊆ alts set (map snd xs) = set (map fst xs)
and x: x ∈ alts y = permutation-of-list xs x
shows pmf (sds R) x = pmf (map-pmf ?σ (sds R)) x
proof –

note perm = list-permutesI [OF perm]
let ?σ = permutation-of-list xs
note perm' = permutation-of-list-permutes [OF perm]
from wf have wf': pref-profile-wf agents alts R by (simp add: pref-profile-from-tableI)
then interpret R: pref-profile-wf agents alts R .
from perm' interpret R': pref-profile-wf agents alts permute-profile ?σ R
by (simp add: R.wf-permute-alts)
from an interpret an-sds agents alts sds .

from eq wf have eq': image-mset (map ((‘) ?σ)) (anonymous-profile R) = anonymous-profile R
by (simp add: anonymise-prefs-from-table mset-map multiset.map-comp)
from perm' x have pmf (sds R) x = pmf (map-pmf ?σ (sds R)) (‘σ x)
by (simp add: pmf-map-inj' permutes-inj)
also from eq' x wf' perm' have map-pmf ?σ (sds R) = sds R
by (intro sds-automorphism)
(simp-all add: R.anonymous-profile-permute pref-profile-from-tableI)
finally show ?thesis using x by simp
qed

5.5 Ex-post efficiency
locale ex-post-efficient-sds = social-decision-scheme agents alts sds
for agents :: 'agent set and alts :: 'alt set and sds +
assumes ex-post-efficient:
is-pref-profile R ⇒ set-pmf (sds R) ∩ pareto-losers R = {}
begin
lemma ex-post-efficient':
assumes is-pref-profile R y ⇒[Pareto(R)] x
shows pmf (sds R) x = 0
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

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. 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 ≠ {} A ⊆ alts and inefficient: ¬SD-efficient R (pmf-of-set A)

assumes_wf: is-pref-profile R

shows ∃ x ∈ A. pmf (sds R) x = 0

**proof (rule ccontr)**

interpret pref-profile-wf agents alts R by fact

with A have set-pmf (pmf-of-set A) ⊆ set-pmf (sds R)

by (subst set-pmf-of-set) (auto simp: set-pmf-eq intro: finite-subset[OF -finite-alts])

from inefficient and this have ¬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 ≠ {} A ⊆ alts and

wit: p ∈ lotteries ∀ i ∈ agents. p ≥ [SD(R i)] pmf-of-set A i ∈ agents

shows ∃ x ∈ A. pmf (sds R) x = 0

**proof (rule SD-inefficient-support)**

from wf interpret pref-profile-wf agents alts R

from wit show ¬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 ⇒ 'agent ⇒ 'alt relation ⇒ bool where

manipulable-profile R i Ri' ←→ sds (R(i := Ri')) > [SD (R i)] sds R

end
An SDS is weakly strategyproof (or just strategyproof) if it is not manipulable 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:

\[
\text{is-pref-profile } R \implies i \in \text{agents} \implies \text{total-preorder-on alts } Ri' \implies \\
\neg \text{manipulable-profile } R i Ri'
\]

### 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 \(R'_{i}\) if the lottery for obtained for \(R\) is at least as good for \(i\) as the lottery obtained when \(i\) misrepresents her preferences as \(R'_{i}\).

**definition** strongly-strategyproof-profile

\[
:: ('agent, 'alt) \text{ pref-profile} \Rightarrow 'agent \Rightarrow 'alt \text{ relation} \Rightarrow \text{bool where}
\]

\[
\text{strongly-strategyproof-profile } R i Ri' \iff \text{sds } R \succeq [SD (R i)] \text{ sds } (R(i := R'i))
\]

**lemma** strongly-strategyproof-profileI [intro]:

**assumes** is-pref-profile \(R\) total-preorder-on alts \(R'_{i}\) \(i \in \text{agents}\)

**assumes** \(\forall x. x \in \text{alts} \implies \text{lottery-prob } (\text{sds } (R(i := R'i))) (\text{preferred-alts } (R i) x)

\[
\leq \text{lottery-prob } (\text{sds } R) (\text{preferred-alts } (R i) x)
\]

**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 \(R i Ri'\)

**shows** \(\neg \text{manipulable-profile } R i Ri'\)

**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 alt-profile Ri' alt-profile R i alt-profile R i'
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
for agents :: 'agent set and alts :: 'alt set and 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 = (\lambda 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
  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 $R' = (\lambda x. \text{False})$ 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$'$ $\equiv$ lift-pref-profile agents alts agents$'$ $R$ $\circ$ $\pi$
  using assms permutes-subset[of assms]
  by (auto simp add: lift-pref-profile-def o-def permutes-in-image)

lemma lift-pref-profile-permute-alts:
  assumes $\sigma$ permutes alts alts $\subseteq$ alts$'$
  shows lift-pref-profile agents alts agents$'$ $(\text{permute-profile } \sigma) R$ $\equiv$ $(\text{permute-profile } \sigma) (\text{lift-pref-profile agents alts agents$'$ R})$
proof
  from assms have inv: $\sigma$ permutes alts by (intro permutes-inv)
  from this assms have inv $\sigma$ permutes alts$'$ by (rule permutes-subset)
  with inv show ?thesis using assms permutes-inj[of $\sigma$ 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 lottery-on-subset AE-measure-pmf-iff)

context
  fixes agents alts R agents$'$ alts$'$ $R'$
  assumes $R$-uf: $\text{pref-profile-uf agents alts } R$
  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 $R$: $\text{pref-profile-uf agents alts } R$ by fact
interpretation $R'$: $\text{pref-profile-uf agents$'$ alts$'$ } R'$
  using election $R$-uf by (simp add: $R'$-def lift-pref-profile-uf)
lemma lift-pref-profile-strict-iff:
\[ x \prec [\text{lift-pref-profile agents alts agents'} alts' R \ i] y \iff i \in \text{agents} \land ((y \in \text{alts} \land x \in \text{alts'}) \lor x \prec [R \ i] y) \]
proof (cases \(i \in \text{agents}\))
case True
then interpret total-preorder-on alts R i by simp
show \(?thesis\) using 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 \(i: i \in \text{agents'}\) and \(x: x \in \text{alts'}\)
shows \(\text{preferred-alts (R'} i) x = (if i \in \text{agents} \land x \in \text{alts} then \text{preferred-alts (R i)} x \ else \text{alts'})\)
proof (cases \(i \in \text{agents}\))
assume \(i: i \in \text{agents}\)
then interpret Ri: total-preorder-on alts R i by simp
show \(?thesis\) using i x election Ri.
not-outside
by (auto simp: preferred-alts-def R'-def lift-pref-profile-def Ri.
refl)
qed (auto simp: preferred-alts-def R'-def lift-pref-profile-def i x)

lemma lift-pref-profile-Pareto-iff:
\[ x \preceq [\text{Pareto (R'} i)] y \iff x \in \text{alts'} \land y \in \text{alts'} \land (x \notin \text{alts} \lor x \preceq [\text{Pareto (R)}] y) \]
proof
from R.nonempty-agents obtain \(i: i \in \text{agents}\) by blast
then interpret Ri: finite-total-preorder-on alts R i by simp
show \(?thesis\) unfolding R'.Pareto-iff R.Pareto-iff unfolding R'-def lift-pref-profile-def
using election i by (auto simp: preorder-on.refl[OF R.in-dom]
simp del: R.nonempty-alts R.nonempty-agents intro: Ri.not-outside)
qed

lemma lift-pref-profile-Pareto-strict-iff:
\[ x \prec [\text{Pareto (R'} i)] y \iff x \in \text{alts'} \land y \in \text{alts'} \land (x \notin \text{alts} \land y \in \text{alts} \lor x \prec [\text{Pareto (R)}] y) \]
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 \(A: x \in \text{alts} y \in \text{alts} if x \prec [\text{Pareto (R)}] y for x y\)
using that R.Pareto.not-outside unfolding strongly-preferred-def by auto
have \(B: x \in \text{alts'} if x \in \text{alts} for x using election that by blast\)
from R.nonempty-alts obtain \(x: x \in \text{alts}\) by blast
thus \(?thesis\) unfolding pareto-losers-def lift-pref-profile-Pareto-strict-iff [abs-def]
by (auto dest: A B)
qed

class context
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  \( Ri : \text{preorder-on alts} \) \( R i \) by simp
  from  
\( i \) election  
\( i' : i \in \text{agents'} \) by blast
then  
interpret  \( R' i : \text{preorder-on alts} \) \( R' i \) by simp
from  
assms election  
\( p \in \text{lotteries-on alts'} \) \( q \in \text{lotteries-on alts'} \)
  by (auto intro: lotteries-on-subset)
with  
assms election \( i' \) show  \(?\)thesis
  by (auto simp: Ri.SD-preorder R'.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  
\( i' : i \in \text{agents'} \) by blast
then  
interpret  \( R' i : \text{preorder-on alts} \) \( R' i \) by simp
from  
assms election  
\( p \in \text{lotteries-on alts'} \) \( q \in \text{lotteries-on alts'} \)
  by (auto intro: lotteries-on-subset)
with  
assms election \( i' \) show  \(?\)thesis
  by (auto simp: R'.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':
\( p \in \text{lotteries-on alts} \implies q \in \text{lotteries-on alts} \implies i \in \text{agents'} \implies \)
\( p \preceq [SD(R' i)] q \iff i \notin \text{agents} \vee p \preceq [SD(R i)] q \)
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  
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  
\( p \preceq [Pareto(\circ SD R') q \iff p \preceq [Pareto(\circ SD R)] q \)
using  
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  
\( p \prec [Pareto(\circ SD R') q \iff p \prec [Pareto(\circ SD R)] q \)

78
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-pmf (lowered R)
    hence x: x ∈ set-pmf (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

80
qed
Qed

sublocale ex-post-efficient-sds agents' alts' lowered
proof
  fix R assume R-wf: lowered.is-pref-profile R
  hence is-pref-profile (lift R) by simp
  have set-pmf (lowered R) ∩ pareto-losers (lift R) = {}
    unfolding lowered-def o-def by (intro ex-post-efficient lift-wf R-wf)
  also have pareto-losers (lift R) = pareto-losers R ∪ (alts − alts')
    by (intro pareto-losers-lift R-wf)
  finally show set-pmf (lowered R) ∩ pareto-losers R = {} by blast
qed

lemma lowered-in-lotteries [simp]: lowered.is-pref-profile R ⇒ lowered R ∈ lotteries
  using lowered.sds-wf[of R] lowered-lotteries by blast

end

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

sublocale lowered: anonymous-sds agents' alts' lowered
proof
  fix π R assume perm: π permutes agents' and R-wf: lowered.is-pref-profile R
  from perm have lift (R o π) = lift R o π
    using agents'-subset by (rule lift-pref-profile-permute-agents)
  hence sds (lift (R o π)) = sds (lift R o π) by simp
  also from perm R-wf have π permutes agents is-pref-profile (lift R)
    using agents'-subset by (auto dest: permutes-subset)
  from anonymous[OF this] have sds (lift R o π) = sds (lift R)
    by (simp add: lowered-def)
  finally show lowered (R o π) = lowered R unfolding lowered-def o-def .
qed

end

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

sublocale lowered: neutral-sds agents' alts' lowered
proof
fix σ R assume perm: σ permutes alts' and R-wf: lowered.is-pref-profile R
from perm alts'-subset
  have lift (permute-profile σ R) = permute-profile σ (lift R)
  by (rule lift-pref-profile-permute-alts)
hence sds (lift (permute-profile σ R)) = sds (permute-profile σ (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 σ (lift R)) = map-pmf σ (sds (lift R))
  by (intro neutral) (auto intro: permutes-subset)
finally show lowered (permute-profile σ R) = map-pmf σ (lowered R)
  by (simp add: lowered-def o-def)
qed

locale sds-lowering-sd-efficient =
sd-efficient-sds agents alts sds +
sds-lowering agents alts sds agents' alts'
for agents :: 'agent set and alts :: 'alt set and sds agents' alts'
begin
sublocale sd-efficient-sds agents alts' lowered
proof
  fix R i Ri
  assume R-wf: lowered.is-pref-profile R
  interpret R: pref-profile-wf 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'
for agents :: 'agent set and alts :: 'alt set and 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 ∈ agents'
  assume Ri': total-preorder-on alts' Ri'
  assume manipulable: lowered.manipulable-profile R i Ri'
  from i agents'-subset have i': i ∈ agents by blast
  interpret R: pref-profile-wf agents' alts' R by fact
from \( R\text{-wf} \) interpret \( \text{liftR}: \text{pref-profile-wf agents alts lift R} \) by simp

define \( \text{lift-R}i' \)
where \( \text{lift-R}i' \ x \ y \longleftrightarrow x \in \text{alts} \land y \in \text{alts} \land (x = y \lor x \notin \text{alts}' \lor (y \in \text{alts}' \land \text{R}i' \ x \ y)) \)

for \( x \ y \) define \( S \) where \( S = (\text{lift R}(i := \text{lift-R}i)) \)
from \( \text{R}i' \) interpret \( \text{R}i': \text{total-preorder-on alts}' \text{R}i' \).
have \( w\text{-lift-R}i': \text{total-preorder-on alts lift-R}i' \) using \( \text{R}i'.\text{total} \)
by unfold-locales (auto simp: \( \text{lift-R}i'\text{-def intro: R}i'.\text{trans} \))
from \( \text{agents}'\text{-subset i have} \ S\text{-altdf: } S = (\text{lift R}(i := R)) \)
have \( \text{lowered } (R(i := R')) \in \text{lowered.lotteries} \)
by (intro lowered.sds-wf R.wf-update i R')
hence \( \text{sds-S-wf: } sds \ S \in \text{lowered.lotteries by (simp add: S-altdf lowered-def} \)

from \( \text{manipulable have} \ \text{lowered } R \prec[S\text{D (R i)] sds (lift R(i := R'))} \)
unfolding \( \text{lowered.manipulable-profile-def by (simp add: lowered-def} \)
also note \( S\text{-altdf [symmetric] \)
finally have \( \text{lowered } R \prec[S\text{D (lift R i)] sds S} \)
using \( R\text{-wf i lowered.sds-wf'[OF R-wf]} \) \( \text{sds-S-wf} \)
by (subst \( \text{lift-SD-strict-iff} \) (simp-all add: \( \text{agents}'\text{-subset alts}'\text{-subset} \))
hence \( \text{manipulable-profile (lift R) i lift-R}i' \)
by (simp add: \( \text{manipulable-profile-def lowered-def S-def} \))
with \( \text{strategyproof}[OF \text{lift-wf}[OF R-wf] i' \text{-wlf-lift-R}i'] \) show False by contradiction
qed

end

locale \( \text{sds-lowering-anonymous-neutral-sd-eff-stratproof} = \)
\( \text{sds-lowering-anonymous + sds-lowering-neutral +} \)
\( \text{sds-lowering-sd-efficient + sds-lowering-strategyproof} \)

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)
}

classification 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 assm 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-locale
  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 .

84
from nonempty-dom have card agents > 0 by (auto simp del: nonempty-agents)
hence ennreal (pmf (RD R) x) = 
    ennreal ((∑ i∈agents. pmf (pmf-of-set (favorites R i)) x) / real (card agents))
    (is _ = ennreal (?p / -)) unfolding random-dictatorship-def[OF assms]
    by (simp-all add: ennreal-pmf-bind nn-integral-pmf-of-set max-def 
also have ?p = (∑ 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 o 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 o 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-profileI)
    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.

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 
  \( i \leftarrow \text{pmf-of-set agents} \); 
  \( \text{random-serial-dictatorship} (\text{agents} - \{i\}) \, (\text{Max-wrt-among} (R \, i) \, \text{alts}) \, R \) 
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 \text{foldr} does not require generalisation over the set of alternatives and is therefore much easier than over \text{foldl}.

**definition rsd-winners** where
\[
\text{rsd-winners} \, R \, \text{alts} \, \text{agents} = \text{foldr} (\lambda i \, \text{alts}. \, \text{Max-wrt-among} (R \, i) \, \text{alts}) \, \text{agents} \, \text{alts}
\]

**lemma rsd-winners-empty** [simp]: \( \text{rsd-winners} \, R \, \text{alts} \, [] = \text{alts} \)
by (simp add: rsd-winners-def)

**lemma rsd-winners-Cons** [simp]:
\[
\text{rsd-winners} \, R \, \text{alts} \, (i \# \text{agents}) = \text{Max-wrt-among} (R \, i) \, (\text{rsd-winners} \, R \, \text{alts} \, \text{agents})
\]
by (simp add: rsd-winners-def)

**lemma rsd-winners-map** [simp]:
\[
\text{rsd-winners} \, R \, \text{alts} \, (\text{map} \, f \, \text{agents}) = \text{rsd-winners} \, (R \circ f) \, \text{alts} \, \text{agents}
\]
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) \, \text{alts} \, \text{alts}) happiness law)
\]

88
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 RSD-pareto-eqclass where**

\[
\text{RSD-pareto-eqclass } \text{agents alts R A} \iff \
A \neq \{\} \land A \subseteq \text{alts} \land (\forall x \in A. \forall y \in \text{alts}. \ y \in A \leftrightarrow (\forall i \in \text{agents}. \ R i \ x \ y))
\]

**lemma RSD-pareto-eqclassI:**

**assumes** $A \neq \{\} \ A \subseteq \text{alts} \land (\forall x \in A. \ y \in \text{alts} \Rightarrow y \in A \leftrightarrow (\forall i \in \text{agents}. \ R i \ x \ y))$

**shows** RSD-pareto-eqclass agents alts R A

**using** assms unfolding RSD-pareto-eqclass-def by simp-all

**lemma RSD-pareto-eqclassD:**

**assumes** RSD-pareto-eqclass agents alts R A

**shows** $A \neq \{\} \ A \subseteq \text{alts} \land (\forall x \in A. \ y \in \text{alts} \Rightarrow y \in A \leftrightarrow (\forall i \in \text{agents}. \ R i \ x \ y))$

**using** assms unfolding RSD-pareto-eqclass-def by simp-all

**lemma RSD-pareto-eqclass-indiff-set:**

**assumes** RSD-pareto-eqclass agents alts R A \ i \in \text{agents} \ x \in A \ y \in A

**shows** R i x y

**using** assms unfolding RSD-pareto-eqclass-def by blast

**lemma RSD-pareto-eqclass-empty [simp, intro!]:**

\[
\text{alts} \neq \{\} \Rightarrow \text{RSD-pareto-eqclass } \{\} \ \text{alts} \ R \ \text{alts}
\]

**by** (auto intro!: RSD-pareto-eqclassI)

**lemma (in pref-profile-wf) RSD-pareto-eqclass-insert:**

**assumes** RSD-pareto-eqclass agents\' alts R A finite alts

\[
i \in \text{agents} \ agents' \subseteq \text{agents}
\]

**shows** RSD-pareto-eqclass (insert i agents\') alts R (Max-wrt-among R i) A

**proof**

**from** assms interpret total-preorder-on alts R i by simp

**show** ?thesis

**proof** (intro RSD-pareto-eqclassI Max-wrt-among-nonempty Max-wrt-among-subset,
goal-cases

  case (3 x y)
    with RSD-pareto-eclassD[of assms(1)]
    show ?case unfolding Max-wrt-among-total-preorder
      by (blast intro: trans)
  qed

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 wfs
    show ?case by auto
  qed
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 ⊆ A implies 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
\[ \text{RSD-set-rel } S A B \leftrightarrow \text{indiff-set } S B \land (\forall x \in A. \forall y \in B. S x y) \]

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** \text{indiff-set-Max-wrt-among}:
\text{assumes finite carrier } A \subseteq \text{carrier } A \neq \{\} \text{ total-preorder-on carrier } S
\text{shows } \text{indiff-set } S (\text{Max-wrt-among } S A)
\text{unfolding } \text{indiff-set-def}
\text{proof}
\text{from } \text{assms}(1-3) \text{ interpret total-preorder-on carrier } S .
\text{from } \text{assms}(1-3) \text{ show Max-wrt-among } S A \neq \{\} \text{ by (intro Max-wrt-among-nonempty) auto}
\text{from } \text{assms}(1-3) \text{ show } \forall x \in \text{Max-wrt-among } S A. \forall y \in \text{Max-wrt-among } S A. S x y
\text{by (auto simp: indiff-set-def Max-wrt-among-total-preorder)}
\text{qed}

We now consider the set of RSD winners in the setting of a preference profile \( R \) and a manipulated profile \( R'(i := Ri') \). 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** \text{rsd-winners-manipulation-aux}:
\text{assumes wf: total-preorder-on alts } Ri'
\text{and i: i } \in \text{agents and set agents'} \subseteq \text{agents finite agents}
\text{and finite: finite alts and alts } \neq \{\}
\text{defines [simp]: } w' \equiv \text{rsd-winners } (R(i := Ri')) \text{ alts and [simp]}: w \equiv \text{rsd-winners } R \text{ alts}
\text{shows } w' \text{ agents'} = w \text{ agents'} \lor \text{RSD-set-rel } (R i) (w' \text{ agents'}) (w \text{ agents'})
\text{using (set agents' } \subseteq \text{agents)}
\text{proof (induction agents')}
\text{case } (\text{Cons } j \text{ agents'})
\text{from } \text{wf i interpret } Ri: \text{total-preorder-on alts } Ri \text{ by simp}
\text{from } \text{wf Cons. prems interpret } Rj: \text{total-preorder-on alts } Rj \text{ by simp}
\text{from } \text{wf interpret } Ri': \text{total-preorder-on alts } Ri' .
\text{from } \text{wf assms Cons. prems}
\text{have indiff-set: indiff-set } (R i) (\text{Max-wrt-among } (R i) (\text{rsd-winners } R \text{ alts agents'}))
\text{by (intro indiff-set-Max-wrt-among[OF finite] rsd-winners-wf) simp-all}
\text{show ?case}
\text{proof (cases } j = i\text{)}
\text{assume j [simp]: } j = i
\text{from } \text{indiff-set Cons have RSD-set-rel } (R i) (w'(j \# \text{agents'})) (w (j \# \text{agents'}))
\text{unfolding RSD-set-rel-def}
\text{by (auto simp: Ri.\text{Max-wrt-among-total-preorder } Ri'.\text{Max-wrt-among-total-preorder})}
\text{thus } ?\text{case ..}
next
  assume j [simp]: j ≠ i
  from Cons have w' agents' = w agents' ∨ RSD-set-rel (R i) (w' agents') (w agents') by simp
  thus ´case
proof
  assume rel: RSD-set-rel (R i) (w' agents') (w agents')
  hence indiff-set: indiff-set (R i) (w agents') by (simp add: RSD-set-rel-def)
  moreover from Cons.prems finite [alts ≠ {}]
    have w agents' ⊆ alts w agents' ≠ {} unfolding w-def
    by (intro rsd-winners-wf; simp)+
  with finite have Max-wrt-among (R j) (w agents') ≠ {}
    by (intro Rj.Max-wrt-among-nonempty) auto
  ultimately have indiff-set (R i) (w (j ≠ agents'))
    (simp add: Rj.Max-wrt-among-subset)
  moreover from rel have ∀ x∈w' (j ≠ agents'). ∀ y∈w (j ≠ agents'). R i x y
    by (auto simp: RSD-set-rel-def Rj.Max-wrt-among-total-preorder)
  ultimately have RSD-set-rel (R i) (w' (j ≠ agents')) (w (j ≠ agents'))
    unfolding RSD-set-rel-def ..
  thus ´case ..
qed simp-all
qed simp-all

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 af: total-preorder-on alts Ri'
    and i: i ∈ agents and set agents' = agents finite agents
    and finite: finite alts and alts ≠ {}
  defines [simp]: w' ≡ rsd-winners (R(i := Ri')) alts and [simp]: w ≡ rsd-winners R alts
  shows ∀ x∈w' agents'. ∀ y∈w agents'. x ≤[R i] y
proof –
  have w' agents' = w agents' ∨ RSD-set-rel (R i) (w' agents') (w agents')
    using rsd-winners-manipulation-aux[OF assms(1-2) - assms(4-6)] assms(3)
    by simp
  thus ?thesis
proof
  assume eq: w' agents' = w agents'
  from assms have RSD-pareto-eqclass (set agents') alts R (w agents') unfolding w-def
    by (intro RSD-pareto-eqclass-rsd-winners-aux) simp-all
  from RSD-pareto-eqclass-indiff-set[OF this, of i'] i eq assms(3) show ?thesis
  by auto
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 permuts-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 RSD-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 ...
    (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)
  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}

\textbf{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: \(\sigma\) permutes alts and wf: is-pref-profile \(R\) from \texttt{wf} interpret pref-profile-wf agents alts \(R\). from \texttt{perm} show RSD (permute-profile \(\sigma\) \texttt{R}) = map-pmf \(\sigma\) (RSD \texttt{R}) by \texttt{(auto intro!: bind-pmf-cong dest!: permutations-of-setD(1) simp: random-serial-dictatorship-altdef rsd-winners-permute-profile map-bind-pmf map-pmf-of-set-inj permutes-inj-on rsd-winners-wf))}}
\texttt{qed}

\textbf{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 \texttt{wf: is-pref-profile \texttt{R}} then interpret pref-profile-wf agents alts \texttt{R}.}
\texttt{from \texttt{x(2)} obtain \(y\) where \texttt{[simp]: \(y\) \texttt{\in alts \texttt{and pareto: \(y \succ \texttt{Pareto}(\texttt{R})\)}}} x
\texttt{by \texttt{(cases rule: pareto-losersE)}}
\texttt{from \texttt{x} have \texttt{[simp]: \(x\) \texttt{\in alts \texttt{using pareto-loser-in-alts by simp}}}
\texttt{from \texttt{x(1)}} obtain \(\texttt{agents'}\) \texttt{where \texttt{agents': set agents' = alts \texttt{and \(x \in set-pmf (pmf-of-set (rsd-winners \texttt{R alts agents'})\)}}} by \texttt{(auto simp: random-serial-dictatorship-altdef dest: permutations-of-setD)}
\texttt{with \texttt{wf have \texttt{x': \(x \in rsd-winners \texttt{R alts agents'\)}} using rsd-winners-wf [where \texttt{alts' = alts and agents' = agents'}] by \texttt{(subst (asm) set-pmf-of-set) (auto simp: permutations-of-setD)}}
\texttt{from \texttt{wf agents' have RSD-pareto-eclass agents alts \texttt{R (rsd-winners \texttt{R alts agents'}) by (intro RSD-pareto-eclass-rsd-winners) simp-all hence \texttt{winner-iff: \(y \in rsd-winners \texttt{R alts agents'\)}} \(\iff \forall i \in \texttt{agents. \(x \preceq (R \texttt{i) y}\)}} if \texttt{x}} \texttt{in rsd-winners \texttt{R alts agents'}} \texttt{\(y \in alts \texttt{for x y\)}} using \texttt{that unfolding RSD-pareto-eclass-def by blast}}
\texttt{from \texttt{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 \( (RSD \ R) \cap \text{pareto-losers} \ R = {} \) 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 \( i \) \[ simp: i \in \text{agents} \and x: x \in \text{alts} \and \]
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 \( wf = wf \)

let \( ?A = \text{preferred-alts} \ (R \ i) \ x \)
from wf interpret \( Ri: \) total-preorder-on alts \( R \ i \) by simp

\[
\{ \\
\text{fix agents'} assume agents': agents' \in \text{permutations-of-set agents} \\
\text{from agents' have} [simp]: \text{set agents'} = \text{agents} \\
\text{by (simp add: permutations-of-set-def)}
\}

let \( ?W = \text{rsd-winners} \ R \text{ alts agents'} \) and \( ?W' = \text{rsd-winners} \ R' \text{ alts agents'} \)
have indiff-set: RSD-pareto-eqclass agents alts \( R \) ?W
by (rule R.RSD-pareto-eqclass-rsd-winners; simp add: wf)+
from R.rsd-winners-uf R'.rsd-winners-uf
have winners: \( ?W \subseteq \text{alts} \ ?W \neq {} \) finite \( ?W \ ?W' \subseteq \text{alts} \ ?W' \neq {} \) finite \( ?W' \)
by simp-all

97
from $\{?W \neq \emptyset\}$ obtain $y$ where $y: y \in ?W$ by blast
with winners have [simp]: $y \in \text{alts}$ by blast
from $\text{wf}^i$ i have mono: $\forall x \in ?W^i. \forall y \in ?W. R \ i \ x \ y$ unfolding $R^i$-def

by (intro $R.\text{rsd-winners-manipulation}$) simp-all

have lottery-prob ($\text{pmf-of-set} \ ?W$) $?A \geq$ lottery-prob ($\text{pmf-of-set} \ ?W^i$) $?A$

proof (cases $y \geq [R \ i] \ x$)

  case True
  with $y$ RSD-pareto-eqclass-indiff-set[OF indiff-set(1), of i] winners
  have $?W \subseteq \text{preferred-alts} \ (R \ i) \ x$
  by (auto intro: Ri.trans simp: preferred-alts-def)
  with winners show $?\text{thesis}$
  by (subst (2) measure-pmf-of-set) (simp-all add: Int-absorb2)

next

case False

  with $y$ mono have $?W^i \cap \text{preferred-alts} \ (R \ i) \ x = \{}$
  by (auto intro: Ri.trans simp: preferred-alts-def)
  with winners show $?\text{thesis}$
  by (subst (1) measure-pmf-of-set)
  (simp-all add: Int-absorb2 one-real-def measure-nonneg)

qed

hence emeasure ($\text{measure-pmf} \ (\text{pmf-of-set} \ ?W)$) $?A \geq$ emeasure ($\text{measure-pmf}$ ($\text{pmf-of-set} \ ?W^i$)) $?A$

  by (simp add: measure-pmf.emeasure-eq-measure)

} } 

hence emeasure ($\text{measure-pmf} \ (\text{RS} \ R)$) $?A \geq$ emeasure ($\text{measure-pmf} \ (\text{RS} \ R^i)$) $?A$

  by (auto simp: random-serial-dictatorship-altdef $AE$-measure-pmf-iff
        intro!: $\text{nn-integral-mono-AE}$)

thus lottery-prob ($\text{RS} \ R$) $?A \geq$ lottery-prob ($\text{RS} \ R^i$) $?A$

  by (simp add: measure-pmf.emeasure-eq-measure)

qed

end

end

theory Randomised-Social-Choice
imports
  Complex-Main
  SDS-Lowering
  Random-Dictatorship
  Random-Serial-Dictatorship
begin

end

9 Automatic definition of Preference Profiles

theory Preference-Profile-Cmd

98
imports
  Complex-Main
  ../Elections

keywords
  preference-profile :: thy-goal
begin

ML-file preference-profiles.ML

class 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))) x
    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-SomeD)+
  from prefs-from-table-wfD (5) [OF assms(1)]
  have Collect (of-weak-ranking (the (map-of xs i))) x =
    {y ∈ alts. of-weak-ranking (the (map-of xs i)) x y}
  by safe (force elim!: of-weak-ranking.cases)
  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
unfolding favorites-def using assms
by (simp add: prefs-from-table-def is-finite-weak-ranking-def
Max-wrt-of-weak-ranking prefs-from-table-wfD)

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 auto
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
function weak-ranking-prefs-from-table where

\[ \text{weak-ranking-prefs-from-table \((i, x)\#xs\) \(j\) = weak-ranking-prefs-from-table \(xs\ j\)} \]

<table>
<thead>
<tr>
<th>(i = j) (\implies) weak-ranking-prefs-from-table ((i, x)#xs) (j = x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>weak-ranking-prefs-from-table ([]\ j = [])</td>
</tr>
<tr>
<td>by (metis list.exhaust old.prod.exhaust) auto</td>
</tr>
</tbody>
</table>

termination by lexicographic-order

lemma eval-weak-ranking-prefs-from-table:

assumess

\[ \text{prefs-from-table-wf agents alts xs} \]

shows

\[ \text{weak-ranking-prefs-from-table \(xs\ i\) = weak-ranking (prefs-from-table \(xs\) \(i\))} \]

proof

(cases \(i \in\) agents)

assume \(i: i \in\) 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 \(i\) have \(i \in\) 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 \(i: i \notin\) agents

with assms have \(i': i \notin\) 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 \(xs\ i\) = (\(\lambda\ -.\ False\))

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:

assumess

\[ R \equiv \text{prefs-from-table \(xs\) \text{prefs-from-table-wf agents alts xs}} \]

shows

\[ R i a b \leftrightarrow \text{prefs-from-table \(xs\) \(i\ a b\)} \]
\[ a \prec [R \ i] \ b \iff \text{prefs-from-table } xs \ i \ a \ b \ \land \neg \text{prefs-from-table } xs \ i \ b \ a \]

\[
\text{anonymous-profile } R = \text{mset (map snd } xs) \\
\text{election agents } alts \implies i \in \text{set (map fst } xs) \implies \\
\text{preferred-alts (R } i) \ x = \\
\text{of-weak-ranking-Collect-ge (rev the (map-of } xs i)) \ x \\
\text{election agents } alts \implies i \in \text{set (map fst } xs) \implies \\
\text{favorites } R \ i = \text{favorites-prefs-from-table } xs \ i \\
\text{election agents } alts \implies i \in \text{set (map fst } xs) \implies \\
\text{weak-ranking (R } i) = \text{weak-ranking-prefs-from-table } xs \ i \\
\text{election agents } alts \implies i \in \text{set (map fst } xs) \implies \\
\text{favorite } R \ i = \text{the-elem (favorites-prefs-from-table } xs \ i) \\
\text{election agents } alts \implies \\
\text{has-unique-favorites } R \iff \text{list-all (\lambda z. is-singleton (hd (snd z))) } xs
\]

\textbf{using} \text{assms \prefs-from-table-wfD [OF assms(2)]}

\textbf{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)

\textbf{lemma prefer-profile-from-tableI'}:
\textbf{assumes} \ R1 \equiv \text{prefs-from-table } xs \ ss \prefs-from-table-wf agents \ alts \ ss
\textbf{shows} \ prefer-profile-wf agents \ alts \ R1
\textbf{using} \text{assms by (simp add: prefer-profile-from-tableI')}

\section*{ML}

\text{signature PREFERENCE-PROFILES-CMD =}

\text{sig}

\text{type info}

\text{val preferencePROFILE :}
\begin{verbatim}
(term * term) * ((binding * (term * term list list) list) list) \rightarrow \text{Proof.context}
\rightarrow \text{Proof.state}
\end{verbatim}

\text{val preference-profile-cmd :}
\begin{verbatim}
(string * string) * ((binding * (string * string list list) list) list) \rightarrow 
\text{Proof.context} \rightarrow \text{Proof.state}
\end{verbatim}

\text{val get-info : term \rightarrow \text{Proof.context} \rightarrow \text{info}
val add-info : term \rightarrow \text{info} \rightarrow \text{Context.generic} \rightarrow \text{Context.generic}
val transform-info : info \rightarrow \text{morphism} \rightarrow \text{info}

end

\text{structure Preference-Profiles-Cmd : PREFERENCE-PROFILES-CMD =}

\text{struct}

\text{open Preference-Profiles}
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 }
end

structure Data = Generic-Data
{
  type T = (term * info) Item-Net.T
  val empty = Item-Net.init (op acnev 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

103
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.mk-setT altT --->
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 add_simps
          @{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}
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
  fun notes thmss suffix attrs lthy =
    Local-Theory.notes (prep-thmss attrs suffix thmss) lthy |
      snd
  fun note thms suffix attrs lthy =
    notes (map single thms) suffix attrs lthy
  fun eval-thmss = map2 (fn def => fn wf =>
    map (fn thm => thm OF [def, wf]) @{thms eval-prefs-from-table-aux})
    defs wf-thms-raw
  fun mk-infos =
    let
      fun aux acc (bdg::bdgs) (t::ts) (r::rws) (def::def-thms) (wf::wf-thms) =
        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 rws def-thms wf-thms wf-raw-thms eval-thms
      in
      aux [] |
        aux - - - - - - = raise Match
    end
  val mk-infos = mk-infos bindings prefs-terms rws defs wf-thms wf-thms-raw
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

in
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

theory QSOpt-Exact
imports Complex-Main
begin
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 : ('a -> string) -> 'a infty -> string
val print-comparison : comparison -> string
val print-optimization-mode : optimization-mode -> string

val save-program : string -> 'a prog -> unit
val solve-program : 'a prog -> 'a result
val read-result : string -> 'a result
val read-result-file : string -> 'a result

end;

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 -> 'a prog -> unit
  val solve-program : 'a prog -> 'a result
  val read-result : string -> 'a result
  val read-result-file : string -> 'a result

end;

structure Linear-Program-Common : LINEAR-PROGRAM-COMMON =
struct

exception QSOpt-Parse

109
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

type var = string
type 'a bound = 'a infty * var * 'a infty
type 'a linterm = ('a * var) list
type 'a constraint = 'a linterm * comparison * 'a
type 'a prog = optimization-mode * 'a linterm * 'a constraint list * 'a 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-Infinity = INF
| print-infty (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 ^ free
| gen-print-bound f (Neg-Infty, v, u) = v ^ <= ^ print-infty f u
| gen-print-bound f (l, v, Pos-Infty) = print-infty f l ^ <= ^ v
| gen-print-bound f (l, v, u) = print-infty f l ^ <= ^ v ^ <= ^ 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)
val 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 ^
  else if mone then -
  else if one then +
  else if neg then - ^ print (negate c) ^
  else + ^ print c ^

110
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
    Stringimplode o chop o rev o chop o rev o Stringexplode
  end

fun gen-read-assignment read x : (string * 'a) option =
  x |> try (Substring.full
      #> Substring.splitl (fn x => x <> #=)
      #> apply2 Substring.string
      #> apsnd (unprefix =)
      #> apply2 trim)
    |> Option.mapPartial (fn (x,y) =>> Option.map (fn y =>> (x, y)) (read y))

112
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)
      s
      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)
end

113
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 \ 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
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 \textit{SD-Efficiency}. For conditions arising from \textit{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 \textit{SD-Efficiency}.

derive-strategyproofness-conditions to automatically derive all conditions arising from weak \textit{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 \texttt{Simplifier}, possibly with a few additional rules, depending on the context.

\begin{verbatim}
lemma disj-False-right: P \lor False \iff P by simp

lemmas multiset-add-ac = add-ac[where \?a = 'a multiset]

lemma less-or-eq-real:
  (x::real) < y \lor x = y \iff x \leq y \forall x < y \lor y = x \iff x \leq y by linarith+

lemma multiset-Diff-single-normalize:
  fixes a c assumes a \neq c
  shows (\{#a\} + B) - \{#c\} = \{#a\} + (B - \{#c\})
  using assms by auto

lemma ex-post-efficient-aux:
  assumes prefs-from-table-wf agents alts xss R \equiv prefs-from-table xss
  assumes as: as \neq [] set as \subseteq alts distinct as A = set as
  assumes ys: \forall x\in set (map snd ys). 0 \leq x sum-list (map snd ys) = 1 set (map fst ys) \subseteq alts

proof
  assume ex-post: ex-post-efficient-sds agents alts sds
  from assms(1,2) have wf: pref-profile-wf agents alts R
    by (simp add: pref-profile-from-tableI''
  from ex-post interpret ex-post-efficient-sds agents alts sds .
  from assms(2-) show pmf (sds R) x = 0
    by (intro ex-post-efficient"[OF wf; of i x]) simp-all
  qed

lemma SD-inefficient-support-aux:
  assumes R: prefs-from-table-wf agents alts xss R \equiv prefs-from-table xss
  assumes as: as \neq [] set as \subseteq alts distinct as A = set as
  assumes ys: \forall x\in set (map snd ys). 0 \leq x sum-list (map snd ys) = 1 set (map fst ys) \subseteq alts

qed
\end{verbatim}
assumes $i : i \in \text{agents}$

assumes $SD1: \forall i \in \text{agents}. \forall x \in \text{alts}.$

\text{sum-list} (\text{map snd} (\text{filter} (\lambda y. \text{prefs-from-table \text{xss} \text{ i} \ x \ \text{ (fst \ y)) \ \text{ys}})) \geq 

\text{real} (\text{length} (\text{filter} (\text{prefs-from-table \text{zss} \text{ i} \ x \ \text{as}))) / \text{real} (\text{length} \ as)

assumes $SD2: \exists x \in \text{alts}. \text{sum-list} (\text{map snd} (\text{filter} (\lambda y. \text{prefs-from-table \text{zss} \text{ i} \ x \ \text{ (fst \ y)) \ \text{ys}})) > 

\text{real} (\text{length} (\text{filter} (\text{prefs-from-table \text{zss} \text{ i} \ x \ \text{as}))) / \text{real} (\text{length} \ as)

shows $sd\text{-efficient-sds agents alts sds} \rightarrow (\exists x \in A. \text{pmf} (\text{sds} R) x = 0)

proof

assume $sd\text{-efficient-sds agents alts sds}$

from $R$ have $uf: \text{pref-profile-uf agents alts R}$

by $(\text{simp add: pref-profile-from-tableI'})$

then interpret $\text{pref-profile-wf agents alts R}.$

interpret $sd\text{-efficient-sds agents alts sds by fact}$

from $ys$ have $ys': \text{pmf-of-list-uf \text{ys} by (intro pmf-of-list-ufI)}$ auto

{

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

  with $ys'$ have $\text{lottery-prob} \ (\text{pmf-of-list ys}) \ (\text{preferred-alts} \ (R i) \ x) =$

  \text{sum-list} (\text{map snd} (\text{filter} (\lambda y. \text{prefs-from-table \text{zss} \text{ i} \ x \ \text{ (fst \ y)) \ \text{ys}}))

  by $(\text{subst measure-pmf-of-list} (\text{simp-all add: preferred-alts-def R})$

} note $A = \text{this}$

{

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

  with $\text{as}$ have $\text{lottery-prob} \ (\text{pmf-of-set} \ (\text{set as})) \ (\text{preferred-alts} \ (R i) \ x) =$

  $\text{real} (\text{card} (\text{set as} \cap \text{preferred-alts} \ (R i) \ x)) / \text{real} (\text{card} (\text{set as}))$

  by $(\text{subst measure-pmf-of-set} \ \text{simp-all}$$)$

  also have $\text{set as} \cap \text{preferred-alts} \ (R i) \ x = \text{set} (\text{filter} (\lambda y. R \ i y y) \ as)$

  by $(\text{auto simp add: preferred-alts-def})$

  also have $\text{card} \ldots = \text{length} (\text{filter} (\lambda y. R \ i y y) \ as)$

  by $(\text{intro distinct-card distinct-filter assms})$

  also have $\text{card} (\text{set as}) = \text{length as} by (\text{intro distinct-card assms})$

  finally have $\text{lottery-prob} \ (\text{pmf-of-set} \ (\text{set as})) \ (\text{preferred-alts} \ (R i) \ x) =$

  $\text{real} (\text{length} (\text{filter} (\text{prefs-from-table \text{zss} \text{ i} \ x \ \text{as}))) / \text{real} (\text{length} \ as)$

  by $(\text{simp add: R})$

} note $B = \text{this}$

from $uf$ show $\exists x \in A. \text{pmf} (\text{sds} R) x = 0$

proof $(\text{rule SD\text{-inefficient-support')}$

from $ys$ $ys'$ show $\text{lottery1: pmf-of-list \text{ys} \in \text{lotteries by (intro pmf-of-list-lottery)}$

show $i: i \in \text{agents by fact}$

from $\text{as}$ have $\text{lottery2: pmf-of-set} \ (\text{set as}) \in \text{lotteries}$

by $(\text{intro pmf-of-set-lottery} \ \text{simp-all}$

from $i$ as $SD2$ $\text{lottery1 \text{lottery2 show ¬SD} (R i) \ (\text{pmf-of-list \text{ys}) \ (\text{pmf-of-set A)})}$

by $(\text{subst preorder-on\text{-SD-preorder[alts]} \ (auto simp: A B not-le})$

from $\text{as}$ $SD1$ $\text{lottery1 \text{lottery2}$ show $\forall i \in \text{agents} \ . \text{SD} (R i) \ (\text{pmf-of-set A}) \ (\text{pmf-of-list \text{ys})}$

117
by safe (auto simp: preorder-on.SD-preorder[of alts] A B)

qed (insert as, simp-all)

qed

definition pref-classes where
pref-classes alts le = preferred-alts le ' alts - {alts}

primrec pref-classes-lists where
pref-classes-lists [] = {}
| pref-classes-lists (xs#xss) = insert (UN (set (xs#xss))) (pref-classes-lists xss)

fun pref-classes-lists-aux where
pref-classes-lists-aux [] = {}
| pref-classes-lists-aux acc (xs#xss) = insert acc (pref-classes-lists-aux (acc ∪ xs) xss)

lemma pref-classes-lists-append:
pref-classes-lists (xs @ ys) = (UN (set ys)) ' pref-classes-lists xs ∪ pref-classes-lists ys
by (induction xs) auto

lemma pref-classes-lists-aux:
assumes is-weak-ranking xss acc (UN (set xss)) = {}
shows pref-classes-lists-aux acc xss =
(insert acc ((λA. A ∪ acc) ' pref-classes-lists (rev xss)) - {acc ∪ (UN (set 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 A: acc ∩ (xs ∪ set xss) = {}
by (simp-all add: is-weak-ranking-Cons)
from Cons.prems have pref-classes-lists-aux (acc ∪ xs) xss =
insert acc (xs ∪ (λA. A ∪ (acc ∪ xs)) ' pref-classes-lists (rev xss)) -
{acc ∪ xs ∪ set xss}
by (intro Cons.IH) (auto simp: is-weak-ranking-Cons)
with Cons.prems have pref-classes-lists-aux acc (xs # xss) =
insert acc (insert acc (xs ∪ (λA. A ∪ (acc ∪ xs)) ' pref-classes-lists (rev xss)) -
{acc ∪ (xs ∪ set xss)})
by (simp-all add: is-weak-ranking-Cons pref-classes-lists-append image-image Un-ac)
also from A have ... = insert acc (insert acc (xs ∪ (λA. A ∪ (acc ∪ xs)) ' pref-classes-lists (rev xss)) -
{acc ∪ (xs ∪ set xss)})
by blast
finally show ?case

118
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 ≠ []
shows pref-classes-lists-aux (hd xss) (tl xss) = pref-classes-lists (rev xss) − {∪ set xss}
proof −
from assms have A: xss = hd xss ≠ tl xss by simp
from assms have hd xss ∩ {set (tl xss)} = {} ∧ 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) ((λA. A ∪ hd xss) ′ pref-classes-lists (rev (tl xss))) −
{hd xss ∪ {set (tl xss)}} by (intro pref-classes-lists-aux) simp-all
also have hd xss ∪ {set (tl xss)} = {set xss} by (subst (3) A, subst set-simps) simp-all
also have insert (hd xss) ((λA. A ∪ hd xss) ′ 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 ′ (∪ set xss) = pref-classes-lists xss
proof safe
fix X x assume x ∈ X X ∈ set xss
with assms show of-weak-ranking-Collect-ge xss x ∈ pref-classes-lists xss
by (induction xss) (auto simp: is-weak-ranking-Cons of-weak-ranking-Collect-ge-Cons′)
next
fix x assume x ∈ pref-classes-lists xss
with assms show x ∈ of-weak-ranking-Collect-ge xss ′ ∪ set xss
proof (induction xss)
case (Cons x xss)
from Cons.prems consider x = xs ∪ {set xss | x ∈ pref-classes-lists xss} by auto
thus ?case
proof cases
assume x = xs ∪ {set xss
with Cons.prems show ?thesis
by (auto simp: is-weak-ranking-Cons of-weak-ranking-Collect-ge-Cons′)
next
assume x: x ∈ pref-classes-lists xss
from Cons.prems x have x ∈ of-weak-ranking-Collect-ge xss ′ ∪ set xss
by (intro Cons.IH) (simp-all add: is-weak-ranking-Cons)
moreover from Cons.prems have xs ∩ {set xss} = {}
by (simp add: is-weak-ranking-Cons)
ultimately have x ∈ of-weak-ranking-Collect-ge xss ′
\[(\{x. x \not\in xs\} \cup \bigcup \text{set } xss) \cap \{x. x \notin xs\}\] by \texttt{blast}

thus \texttt{thesis by (simp add: of-weak-ranking-Collect-ge-Cons')}

qed

qed simp-all

qed

\textbf{lemma} eval-pref-classes-of-weak-ranking:
\begin{itemize}
  \item \textbf{assumes} \bigcup \text{set } xss = \text{alts is-weak-ranking } xss \text{ alts} \neq \{\}
  \item \textbf{shows} \text{pref-classes alts } (\text{of-weak-ranking } xss) = \text{pref-classes-lists-aux } (hd xss)
\end{itemize}

\textbf{proof} –
\begin{itemize}
  \item \textbf{have} \text{pref-classes alts } (\text{of-weak-ranking } xss) = \text{preferred-alts } (\text{of-weak-ranking } xss) \setminus (\bigcup \text{set } xss)
    \texttt{by (simp add: pref-classes-def \textbf{assms})}
  \item \textbf{also} \{ \textbf{have} \text{of-weak-ranking-Collect-ge } (\text{rev } xss) \setminus (\bigcup \text{set } xss) = \text{pref-classes-lists } (\text{rev } xss)
    \texttt{using \textbf{assms by (intro pref-classes-of-weak-ranking-aux simp-all)} simp-all}
  \item \textbf{also have} \text{of-weak-ranking-Collect-ge } (\text{rev } xss) = \text{preferred-alts } (\text{of-weak-ranking } xss)
    \texttt{by (intro ext) (simp-all add: of-weak-ranking-Collect-ge-def preferred-alts-def)}
  \item \textbf{finally have} \text{preferred-alts } (\text{of-weak-ranking } xss) \setminus (\bigcup \text{set } xss) = \text{pref-classes-lists } (\text{rev } xss) .
\}\}
\item \textbf{also from} \textbf{assms} \text{have} \text{pref-classes-lists } (\text{rev } xss) \setminus (\bigcup \text{set } xss) = \text{pref-classes-lists-aux } (hd xss)
\texttt{by (intro pref-classes-list-aux-hd-tl [symmetric]) auto}
\item \textbf{finally show} \texttt{thesis by simp}
\end{itemize}

qed

\textbf{context preorder-on}

\textbf{begin}

\textbf{lemma} SD-iff-pref-classes:
\begin{itemize}
  \item \textbf{assumes} \text{p} \in \text{lotteries-on carrier } q \in \text{lotteries-on carrier}
  \item \textbf{shows} \text{p} \preceq_{[SD(\leq)]} q \iff \forall A\in\text{pref-classes carrier } le. \text{measure-pmf.prob } p A \leq \text{measure-pmf.prob } q A
\end{itemize}

\textbf{proof} \texttt{safe}
\begin{itemize}
  \item \textbf{fix} \text{A} \textbf{assume} \text{p} \preceq_{[SD(\leq)]} q \text{ A} \in \text{pref-classes carrier } le
  \item \textbf{thus} \text{measure-pmf.prob } p A \leq \text{measure-pmf.prob } q A
    \texttt{by (auto simp: SD-preorder pref-classes-def)}
\end{itemize}

\textbf{next}
\begin{itemize}
  \item \textbf{assume} \text{A: } \forall A\in\text{pref-classes carrier } le. \text{measure-pmf.prob } p A \leq \text{measure-pmf.prob } q A
  \item \textbf{show} \text{p} \preceq_{[SD(\leq)]} q
    \texttt{proof (rule SD-preorder1)}
    \begin{itemize}
      \item \textbf{fix} \text{x} \textbf{assume} \text{x: } x \in \text{carrier}
    \end{itemize}
\end{itemize}
show \text{measure-pmf.prob} p (\text{preferred-alts le} x) \leq \text{measure-pmf.prob} q (\text{preferred-alts le} x)

proof (cases \text{preferred-alts le} x = \text{carrier})
  case False
  with \text{x have} \text{preferred-alts le} x \in \text{pref-classes carrier le}
  unfolding \text{pref-classes-def by} (\text{intro DiffI imageI}) simp-all
  with A show \text{thesis by simp}

next
  case True
  from \text{assms have} \text{measure-pmf.prob} p \text{carrier} = 1 \text{measure-pmf.prob} q \text{carrier} = 1
  by (auto simp: \text{measure-pmf.prob-eq-1 lotteries-on-def AE-measure-pmf-iff})
  with True show \text{thesis by simp}

qed

end

lemma (in strategyproof-an-sds) strategyproof':
\text{assumes wf: \text{is-pref-profile} R total-preorder-on alts \text{Ri'} and \text{i: i \in agents}}
\text{shows } (\exists A \in \text{pref-classes alts} (R \text{ i}) \cdot \text{lottery-prob} (\text{sds} (R(i := \text{Ri'}))) \ A < \text{lottery-prob} (\text{sds} R) A) \lor
(\forall A \in \text{pref-classes alts} (R \text{ i}) \cdot \text{lottery-prob} (\text{sds} (R(i := \text{Ri'}))) \ A = \text{lottery-prob} (\text{sds} R) A)

proof
  from \text{wf(1) interpret R: \text{pref-profile-wf agents alts R} .}
  from \text{i interpret total-preorder-on alts R i by simp}
  from \text{assms have \neg \text{manipulable-profile} R i \text{Ri'} by (intro strategyproof)}
  moreover from \text{wf i have} \text{sds R \in lotteries sds (R(i := \text{Ri'}) \in lotteries}
  by (simp-all add: sds-wf)
  ultimately show \text{thesis}
    by (fastforce simp: \text{manipulable-profile-def strongly-preferred-def SD-iff-pref-classes not-le not-less})

qed

lemma \text{pref-classes-lists-aux-finite}:
A \in \text{pref-classes-lists-aux aux \text{acc xss} \Rightarrow \text{finite xss} \Rightarrow (\forall A. A \in \text{set xss} \Rightarrow \text{finite A})}
  \Rightarrow \text{finite A}

by (induction acc xss rule: \text{pref-classes-lists-aux.induct}) auto

lemma strategyproof-aux:
\text{assumes wf: \text{prefs-from-table-wf agents alts xss1 R1 = prefs-from-table xss1}}
\text{prefs-from-table-wf agents alts xss2 R2 = \text{prefs-from-table xss2}}
\text{assumes sds: strategyproof-an-sds agents alts sds and i: \text{i \in agents}}
\text{and j: \text{j \in agents}}
\text{assumes eq: R1(i := R2 j) = R2 the (map-of xss1 i) = xs}
\text{pref-classes-lists-aux (hd xs) (tl xs) = ps}

121
shows \[ (∃ A ∈ ps. (∑ x ∈ A. pmf (sds R2) x < (∑ x ∈ A. pmf (sds R1) x)) \lor \\
∀ A ∈ ps. (∑ x ∈ A. pmf (sds R2) x = (∑ x ∈ A. pmf (sds R1) x)) \]

proof –

from sds interpret strategyproof-an-sds agents alts sds .

let ?Ri' = R2 j

from wf j have wf': is-pref-profile R1 total-preorder-on alts ?Ri'

  by (auto intro: pref-profile-from-tableI pref-profile-wf prefs-wf'

from wf(1) i have i ∈ set (map fst xss1) by (simp add: prefs-from-table-wf-def)

with prefs-from-table-wfD(3)[OF wf(1)] eq

have xs ∈ set (map snd xss1) by force

note xs = prefs-from-table-wfD(2)[OF wf(1)]

this]

{ fix p A assume A: A ∈ pref-classes-lists-aux (hd xs) (tl xs)

  from xs have xs ≠ [] by auto

  with xs have finite A

    by (intro pref-classes-lists-aux-finite[OF A])

    (auto simp: is-finite-weak-ranking-def list.sel)

    hence lottery-prob p A = (∑ x ∈ A. pmf p x)

    by (rule measure-measure-pmf-finite)

  } note A = this

from strategyproof"[OF wf' i] eq have

(∃ A ∈ pref-classes alts (R1 i). lottery-prob (sds R2) A < lottery-prob (sds R1) A)

(∀ A ∈ pref-classes alts (R1 i). lottery-prob (sds R2) A = lottery-prob (sds R1) A)

by simp

also from wf eq i have R1 i = of-weak-ranking xs

  by (simp add: prefs-from-table-map-of)

also from xs have pref-classes alts (of-weak-ranking xs) = pref-classes-lists-aux (hd xs) (tl xs)

  unfolding is-finite-weak-ranking-def by (intro eval-pref-classes-of-weak-ranking)

  simp-all

finally show ?thesis by (simp add: A eq)

qed

lemma strategyproof-aux':

assumes wf: 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 ∈ agents and j: j ∈ agents

assumes perm: list-permutes ys alts

defines σ ≡ permutation-of-list ys and σ' ≡ inverse-permutation-of-list ys

defines xs ≡ the (map-of xss1 i)

defines xs': xs' ≡ map ((') σ) (the (map-of xss2 j))

defines Ri' ≡ of-weak-ranking xs'}
assumes distinct-ps: \( \forall A \in ps, \) distinct \( A \)
assumes eq: \( \text{mset} (\text{map} \ \text{snd} \ \text{xs}s1) - \{\text{#the} (\text{map-of} \ \text{xs}s1 \ i)\#\} + \{\text{#xs}'\#\} = \text{mset} (\text{map} (\text{\texttt{(')} } \sigma) \circ \text{snd} \ \text{xs}s2) \)

\[
\text{pref-classes-lists-aux} \left( \text{hd} \ \text{xs} \right) \left( \text{tl} \ \text{xs} \right) = \text{set} \ ' \ \text{ps}
\]

shows \( \text{list-permutates} \ \text{ys} \ \text{alts} \land \)

\[
(\exists A \in ps. (\sum x \mapsto A. \ \text{pmf} (\text{mset} R2) (\sigma' x)) < (\sum x \mapsto A. \ \text{pmf} (\text{mset} R1) x)) \lor \]

\[
(\forall A \in ps. (\sum x \mapsto A. \ \text{pmf} (\text{mset} R2) (\sigma' x)) = (\sum x \mapsto A. \ \text{pmf} (\text{mset} R1) x))
\]

(proof)

from \( \text{perm} \) \text{have} \( \text{perm}' : \sigma \text{ permutes} \ \text{alts} \) by (simp add: \( \text{def} \))
from \( \text{sds} \) interpret strategy-proof-an-sds agents \( \text{alts} \) \( \text{sds} \).
from \( \text{wf}(3) j \) \text{have} \( j \in \text{set} \ (\text{map} \ \text{fst} \ \text{xs}s2) \) by (simp add: \( \text{pref-from-table-wf-def} \))

with \( \text{pref-from-table-wfD}(3)(\text{OF} \ \text{wf}(3)) \)

\( \text{have} \ \text{xs}'-aux: \text{the} (\text{map-of} \ \text{xs}s2 \ j) \in \text{set} \ (\text{map} \ \text{snd} \ \text{xs}s2) \) (by force)

with \( \text{wf}(3) \) \text{have} \( \text{xs}'-aux' : \text{is-finite-weak-ranking} \ (\text{the} (\text{map-of} \ \text{xs}s2 \ j)) \)

by (auto simp: \( \text{pref-from-table-wf-def} \))

hence \( : \text{is-weak-ranking} \ \text{xs}' \) unfolding \( \text{xs}' \)

by (intro is-weak-ranking-map-inj permutes-inj-on[\( \text{OF} \ \text{perm}' \)])

(auto simp add: \( \text{is-finite-weak-ranking-def} \))

moreover from \( * \) \( \text{xs}'-aux' \) \text{have} \( \text{is-finite-weak-ranking} \ \text{xs}' \)

by (auto simp: \( \text{xs}' \) is-finite-weak-ranking-def)

moreover from \( \text{pref-from-table-wfD}(5)(\text{OF} \ \text{wf}(3)) \text{xs}'-aux \)

\( \text{have} \ \text{set} \ \text{xs}' = \text{alts} \) unfolding \( \text{xs}' \)

by (simp add: \( \text{image-Union} \) [symmetric] \( \text{permutes-image}[\text{OF} \ \text{perm}'] \))

ultimately have \( \text{wf-} \text{xs}' : \text{is-weak-ranking} \ \text{xs}' \) \( \text{is-finite-weak-ranking} \ \text{xs}' \) \( \text{set} \ \text{xs}' = \text{alts} \)

by (simp-all add: \( \text{is-finite-weak-ranking-def} \))

from \( \text{this} \ \text{wf} \ j \) \text{have} \( \text{wf}' : \text{is-pref-profile} \ \text{R1} \) \text{total-preorder-on} \( \text{alts} \) \( \text{Ri}' \)

\( \text{is-pref-profile} \ \text{R2} \) \text{finite-total-preorder-on} \( \text{alts} \) \( \text{Ri}' \)

unfolding \( \text{Ri}'-def \) by (auto intro: \( \text{pref-profile-from-tableI} \) \( \text{pref-profile-wf} \) \( \text{pref-wf'(1)} \)

\( \text{total-preorder-of-weak-ranking} \))

interpret \( \text{R1} : \text{pref-profile-wf} \) agents \( \text{alts} \) \( \text{R1} \) by \( \text{fact} \)
interpret \( \text{R2} : \text{pref-profile-wf} \) agents \( \text{alts} \) \( \text{R2} \) by \( \text{fact} \)

from \( \text{wf}(1) i \) \text{have} \( i \in \text{set} \ (\text{map} \ \text{fst} \ \text{xs}s1) \) by (simp add: \( \text{pref-from-table-wf-def} \))

with \( \text{pref-from-table-wfD}(3)(\text{OF} \ \text{wf}(1)) \) eq(2)

\( \text{have} \ \text{xs} \in \text{set} \ (\text{map} \ \text{snd} \ \text{xs}s1) \) unfolding \( \text{xs-def} \) by force

note \( \text{pref-from-table-wfD}(2)(\text{OF} \ \text{wf}(1)) \) \text{pref-from-table-wfD}(5,6)(\text{OF} \ \text{wf}(1) \) this)

from \( \text{wf} i \) \( \text{wf}' \) \( \text{wf-} \text{xs}' \) \( \text{xs} \) eq

\( \text{have} \ \text{eq} : \text{anonymous-profile} \ (\text{R1}(i := \text{Ri}')) = \text{image-mset} \ (\text{map} ((\text{'} ) \ \sigma)) \)

(anonymous-profile \( \text{R2} \))

by (subst \( \text{R1.anonymous-profile-update} \))

(simp-all add: \( \text{Ri}'-def \) weak-ranking-of-weak-ranking mset-map \( \text{multiset.map-comp} \)

\( \text{xs-def} \)

123
\begin{align*}
\text{anonymise-prefs-from-table} & \text{ prefs-from-table-map-of) } \\
\{ & \text{fix } p \ A \text{ assume } A : A \in \text{pref-classes-lists-aux} \ (\text{hd } xs) \ (\text{tl } xs) \\
& \text{from } xs \ \text{have } xs \neq [] \ \text{by } \text{auto} \\
& \text{with } xs \ \text{have } \text{finite } A \\
& \text{by } (\text{intro } \text{pref-classes-lists-aux-finite}[\text{OF } A]) \\
& \text{(auto simp: is-finite-weak-ranking-def list.set.sel)} \\
& \text{hence } \text{lottery-prob } p A = \left(\sum x \in A. \ \text{pmf } p x\right) \\
& \text{by } (\text{rule } \text{measure-measure-pmf-finite}) \\
\} & \text{note } A = \text{this} \\
& \text{from } \text{strategyproof} [\text{OF } \text{wf'}(1,2) \ i] \ \text{eq'} \ \text{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) \ \lor \\
& (\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) \\
& \text{by simp} \\
& \text{also from } \text{eq'} \ i \ \text{have } \text{sds} (R1(i := R'i)) = \text{map-pmf } \sigma \ (\text{sds } R2) \\
& \text{unfolding } \sigma'-\text{def by } (\text{intro } \text{sds-anonymous-neutral permutation-of-list-permates} \\
& \text{perm } \text{wf'}) \ \text{pref-profile-wf'} \\
& \text{also from } \text{wf eq i} \ \text{have } R1 i = \text{of-weak-ranking } xs \\
& \text{by } (\text{simp add: prefs-from-table-map-of } xs\text{-def}) \\
& \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) \\
& \text{unfolding } \text{is-finite-weak-ranking-def by } (\text{intro } \text{eval-pref-classes-of-weak-ranking} \\
& \text{simp-all}) \\
& \text{finally have } (\exists A \in ps. \ \left(\sum x \in A. \ \text{pmf} (\text{map-pmf } \sigma \ (\text{sds } R2)) x\right) < (\sum x \in A. \ \text{pmf} \\
& (\text{sds } R1) x) \ \lor \\
& (\forall A \in ps. \ \left(\sum x \in A. \ \text{pmf} (\text{map-pmf } \sigma \ (\text{sds } R2)) x\right) = (\sum x \in A. \ \text{pmf} \\
& (\text{sds } R1) x)) \\
& \text{using } \text{distinct-ps} \\
& \text{by } (\text{simp add: } A \ \text{eq sum}.\text{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)) \\
& \text{using } \text{pmf-map-inj'}[\text{of } \sigma - \text{ inv } \sigma x \ \text{for } x] \\
& \text{by } (\text{simp add: fun-eq-iff permutes-inj permutes-inverses}) \\
& \text{also from } \text{perm} \ \text{have } \text{inv } \sigma = \sigma' \ \text{unfolding } \sigma'-\text{def} \\
& \text{by } (\text{rule } \text{inverse-permutation-of-list-correct [symmetric]}) \\
& \text{finally show } \text{?th}. \\
& \text{qed fact+}
\end{align*}
References