Randomised Social Choice

Manuel Eberl

September 9, 2026

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 . . . 69
5.4 Neutrality . . . 69
5.5 Ex-post efficiency . . . 72
5.6 SD efficiency . . . 72
5.7 Weak strategyproofness . . . 73
5.8 Strong strategyproofness . . . 74

6 Lowering Social Decision Schemes 75

7 Random Dictatorship 84
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 . . . 96
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 99
9.1 Automatic definition of preference profiles from tables . . . 100

10 Automatic Fact Gathering for Social Decision Schemes 115

1 Order Relations as Binary Predicates

theory *Order-Predicates*
imports
Main
HOL−Library.Disjoint-Sets
HOL−Combinatorics.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 = (λ*x y*. *R* (*f x*) (*f y*))

definition *restrict-relation* :: *'a set* ⇒ *'a relation* ⇒ *'a relation* **where**
restrict-relation A R = (λ*x 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*)

lemma *preorder-on-restrict*:
preorder-on (*carrier* ∩ *A*) (*restrict-relation A le*)
by *unfold-locales* (*auto simp*: *restrict-relation-def refl intro*: *trans not-outside*)

lemma *preorder-on-restrict-subset*:
A ⊆ *carrier* ⟹ *preorder-on A* (*restrict-relation A le*)
using *preorder-on-restrict*[*of A*] **by** (*simp add*: *Int-absorb1*)

lemma *restrict-relation-carrier* [*simp*]:
restrict-relation carrier le = *le*
using *not-outside* **by** (*intro ext*) (*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* ∈ *carrier* ⟹ *y* ∈ *carrier* ⟹ *le x y* ∨ *le y x*
begin

lemma *total'*: ¬*le x y* ⟹ *x* ∈ *carrier* ⟹ *y* ∈ *carrier* ⟹ *le y x*
using *total*[*of x y*] **by** *blast*

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

lemma *total-preorder-on-restrict*:
total-preorder-on (*carrier* ∩ *A*) (*restrict-relation A le*)
proof −
interpret *R'*: *preorder-on carrier* ∩ *A restrict-relation A le*
by (*rule preorder-on-restrict*)
from *total* **show** *?thesis*
by *unfold-locales* (*auto simp*: *restrict-relation-def*)
qed

lemma *total-preorder-on-restrict-subset*:
A ⊆ *carrier* ⟹ *total-preorder-on A* (*restrict-relation A le*)
using *total-preorder-on-restrict*[*of A*] **by** (*simp add*: *Int-absorb1*)

end

Some fancy notation for order relations

abbreviation (*input*) *weakly-preferred* :: *'a* ⇒ *'a relation* ⇒ *'a* ⇒ *bool*
(‹- ⪯[-] -› [*51*,*10*,*51*] *60*) **where**
a ⪯[*R*] *b* ≡ *R a b*

definition *strongly-preferred* (‹- ≺[-] -› [*51*,*10*,*51*] *60*) **where**
a ≺[*R*] *b* ≡ (*a* ⪯[*R*] *b*) ∧ ¬(*b* ⪯[*R*] *a*)

definition *indifferent* (‹- ∼[-] -› [*51*,*10*,*51*] *60*) **where**
a ∼[*R*] *b* ≡ (*a* ⪯[*R*] *b*) ∧ (*b* ⪯[*R*] *a*)

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

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*: $y \in A \bigwedge z.\ z \in A \Longrightarrow le\ y\ z \Longrightarrow le\ z\ y$ **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*:
$carrier \neq \{\} \Longrightarrow$ *finite carrier* $\Longrightarrow$ *Max-wrt le* $\neq \{\}$
using *Max-wrt-among-nonempty*[*of UNIV*] **by** (*simp add*: *Max-wrt-def*)

lemma *Max-wrt-among-map-relation-vimage*:
$f -` Max\text{-}wrt\text{-}among\ le\ A \subseteq Max\text{-}wrt\text{-}among\ (map\text{-}relation\ f\ le)\ (f -` A)$
by (*auto simp*: *Max-wrt-among-def map-relation-def*)

lemma *Max-wrt-map-relation-vimage*:
$f -` Max\text{-}wrt\ le \subseteq Max\text{-}wrt\ (map\text{-}relation\ f\ le)$
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 $f ` B \subseteq the\text{-}inv\text{-}into\ A\ f -` B$
using *assms* **by** (*auto simp*: *the-inv-into-f-f*)

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

lemma *Max-wrt-among-map-relation-bij*:
assumes *bij f*
shows $f ` Max\text{-}wrt\text{-}among\ le\ A = Max\text{-}wrt\text{-}among\ (map\text{-}relation\ (inv\ f)\ le)\ (f ` A)$
proof (*intro equalityI Max-wrt-among-map-relation-bij-subset assms*)
interpret *R*: *preorder-on f ` carrier map-relation* (*inv f*) *le*
using *preorder-on-map*[*of inv f*] *assms*
by (*simp add*: *bij-imp-bij-inv bij-vimage-eq-inv-image inv-inv-eq*)
show $Max\text{-}wrt\text{-}among\ (map\text{-}relation\ (inv\ f)\ le)\ (f ` A) \subseteq f ` Max\text{-}wrt\text{-}among\ le\ A$
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*:
bij f $\Longrightarrow$ *f ' Max-wrt le* = *Max-wrt* (*map-relation* (*inv f*) *le*)
proof −
assume *bij*: *bij f*
interpret *R*: *preorder-on f ' carrier map-relation* (*inv f*) *le*
using *preorder-on-map*[*of inv f*] *bij*
by (*simp add*: *bij-imp-bij-inv bij-vimage-eq-inv-image inv-inv-eq*)
from *bij* **show** *?thesis*
unfolding *R.Max-wrt-preorder Max-wrt-preorder*
by (*auto simp*: *map-relation-def inv-f-f bij-is-inj*)
qed

lemma *Max-wrt-among-mono*:
le x y $\Longrightarrow$ *x* ∈ *Max-wrt-among le A* $\Longrightarrow$ *y* ∈ *A* $\Longrightarrow$ *y* ∈ *Max-wrt-among le A*
using *not-outside* **by** (*auto simp*: *Max-wrt-among-preorder intro*: *trans*)

lemma *Max-wrt-mono*:
le x y $\Longrightarrow$ *x* ∈ *Max-wrt le* $\Longrightarrow$ *y* ∈ *Max-wrt le*
unfolding *Max-wrt-def* **using** *Max-wrt-among-mono*[*of x y UNIV*] **by** *blast*

end

context *total-preorder-on*
begin

lemma *Max-wrt-among-total-preorder*:
Max-wrt-among le A = {*x*∈*carrier* ∩ *A*. ∀ *y*∈*carrier* ∩ *A*. *le y x*}
unfolding *Max-wrt-among-preorder* **using** *total* **by** *blast*

lemma *Max-wrt-total-preorder*:
Max-wrt le = {*x*∈*carrier*. ∀ *y*∈*carrier*. *le y x*}
unfolding *Max-wrt-preorder* **using** *total* **by** *blast*

lemma *decompose-Max*:
assumes *A*: *A* ⊆ *carrier*
defines *M* ≡ *Max-wrt-among le A*
shows *restrict-relation A le* = (λ*x y*. *x* ∈ *A* ∧ *y* ∈ *M* ∨ (*y* ∉ *M* ∧ *restrict-relation* (*A* − *M*) *le 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**

$i \leq j \Longrightarrow i < \mathit{length}\ \mathit{xs} \Longrightarrow j < \mathit{length}\ \mathit{xs} \Longrightarrow x \in \mathit{xs}\ !\ i \Longrightarrow y \in \mathit{xs}\ !\ j \Longrightarrow$
$x \succeq[\mathit{of\text{-}weak\text{-}ranking}\ \mathit{xs}]\ y$

lemma *of-weak-ranking-Nil* [*simp*]: *of-weak-ranking* [] = (λ- -. *False*)
by (*intro ext*) (*simp add*: *of-weak-ranking.simps*)

lemma *of-weak-ranking-Nil′* [*code*]: *of-weak-ranking* [] *x y* = *False*
by *simp*

lemma *of-weak-ranking-Cons* [*code*]:
$x \succeq[\mathit{of\text{-}weak\text{-}ranking}\ (z\#zs)]\ y \longleftrightarrow x \in z \wedge y \in \bigcup(\mathit{set}\ (z\#zs)) \vee x \succeq[\mathit{of\text{-}weak\text{-}ranking}\ zs]\ y$
(**is** *?lhs* ⟷ *?rhs*)
proof
assume *?lhs*
then obtain *i j*
where *ij*: $i < \mathit{length}\ (z\#zs)\ j < \mathit{length}\ (z\#zs)\ i \leq j\ x \in (z\#zs)\ !\ i\ y \in (z\#zs)\ !\ j$
by (*blast elim*: *of-weak-ranking.cases*)
thus *?rhs* **by** (*cases i*; *cases j*) (*force intro*: *of-weak-ranking.intros*)+
next
assume *?rhs*
thus *?lhs*
proof (*elim disjE conjE*)
assume $x \in z\ y \in \bigcup(\mathit{set}\ (z\ \#\ zs))$
then obtain *j* **where** $j < \mathit{length}\ (z\ \#\ zs)\ y \in (z\ \#\ zs)\ !\ j$
by (*subst* (*asm*) *set-conv-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** $i < \mathit{length}\ zs\ j < \mathit{length}\ zs\ i \leq j\ x \in zs\ !\ i\ y \in zs\ !\ j$
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
qed

lemma *of-weak-ranking-indifference*:
assumes $A \in \mathit{set}\ \mathit{xs}\ x \in A\ y \in A$
shows $x \preceq[\mathit{of\text{-}weak\text{-}ranking}\ \mathit{xs}]\ y$
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* ($\bigcup$(*set xs*))
shows *map-relation f* (*of-weak-ranking xs*) = *of-weak-ranking* (*map* ((‘) (*inv f*)) *xs*)
proof −
have *map-relation f* (*of-weak-ranking xs*) = *of-weak-ranking* (*map* ((−‘) *f*) *xs*)
by (*rule of-weak-ranking-map*)
also from *assms* **have** *map* ((−‘) *f*) *xs* = *map* ((‘) (*inv f*)) *xs*
by (*intro map-cong refl*) (*simp-all add*: *bij-vimage-eq-inv-image permutes-bij*)
finally show *?thesis* **.**
qed

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

definition *is-weak-ranking* **where**
is-weak-ranking xs ⟷ ({} ∉ *set xs*) ∧
(∀ *i j*. *i* < *length xs* ∧ *j* < *length xs* ∧ *i* ≠ *j* ⟶ *xs* ! *i* ∩ *xs* ! *j* = {})

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

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

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

lemma *is-weak-ranking-nonempty*: *is-weak-ranking xs* ⟹ {} ∉ *set xs*
by (*simp add*: *is-weak-ranking-def*)

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

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

```
  show distinct xs
  proof (subst distinct-conv-nth, safe)
    fix i j assume ij: i < 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
  assume 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 dis-
joint-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-Nil [simp, code]: is-weak-ranking []
  by (auto simp: is-weak-ranking-def)

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

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

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

lemma *is-weak-ranking-singleton* [*simp*]:
  is-weak-ranking [*x*] ⟷ *x* ≠ {}
  by (*auto simp add*: *is-weak-ranking-def*)

lemma *is-finite-weak-ranking-singleton* [*simp*]:
  is-finite-weak-ranking [*x*] ⟷ *x* ≠ {} ∧ *finite x*
  by (*auto simp add*: *is-finite-weak-ranking-def*)

lemma *is-weak-ranking-append*:
  is-weak-ranking (*xs* @ *ys*) ⟷
      is-weak-ranking xs ∧ *is-weak-ranking ys* ∧
      (*set xs* ∩ *set ys* = {} ∧ ⋃(*set xs*) ∩ ⋃(*set ys*) = {})
  by (*simp only*: *is-weak-ranking-iff*)
     (*auto dest*: *disjointD disjoint-unionD1 disjoint-unionD2 intro*: *disjoint-union*)

lemma *is-weak-ranking-Cons*:
  is-weak-ranking (*x* # *xs*) ⟷
      x ≠ {} ∧ *is-weak-ranking xs* ∧ *x* ∩ ⋃(*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*) ⟷
      x ≠ {} ∧ *finite x* ∧ *is-finite-weak-ranking xs* ∧ *x* ∩ ⋃(*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 [] ⟷ *True*
| *is-weak-ranking-aux A* (*x*#*xs*) ⟷ *x* ≠ {} ∧
       A ∩ *x* = {} ∧ *is-weak-ranking-aux* (*A* ∪ *x*) *xs*

lemma *is-weak-ranking-aux*:
  is-weak-ranking-aux A xs ⟷ *A* ∩ ⋃(*set xs*) = {} ∧ *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 ⟷ *is-weak-ranking-aux* {} *xs*
  by (*subst is-weak-ranking-aux*) *auto*

lemma *of-weak-ranking-altdef*:
  assumes *is-weak-ranking xs x* ∈ ⋃(*set xs*) *y* ∈ ⋃(*set xs*)
  shows *of-weak-ranking xs x y* ⟷
          find-index ((∈) *x*) *xs* ≥ *find-index* ((∈) *y*) *xs*

proof −
from *assms*
have *A*: *find-index* ((∈) *x*) *xs* < *length xs find-index* ((∈) *y*) *xs* < *length xs*
by (*simp-all add*: *find-index-less-size-conv*)
from *this*[*THEN nth-find-index*]
have *B*: *x* ∈ *xs* ! *find-index* ((∈) *x*) *xs y* ∈ *xs* ! *find-index* ((∈) *y*) *xs* **.**
show *?thesis*
proof
assume *of-weak-ranking xs x y*
then 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*) *simp-all*
with *A B* **have** *i* = *find-index* ((∈) *x*) *xs j* = *find-index* ((∈) *y*) *xs*
using *assms*(*1*) **unfolding** *is-weak-ranking-def* **by** *blast*+
with *ij* **show** *find-index* ((∈) *x*) *xs* ≥ *find-index* ((∈) *y*) *xs* **by** *simp*
next
assume *find-index* ((∈) *x*) *xs* ≥ *find-index* ((∈) *y*) *xs*
from *this A*(*2*,*1*) *B*(*2*,*1*) **show** *of-weak-ranking xs x y*
by (*rule of-weak-ranking.intros*)
qed
qed

lemma *total-preorder-of-weak-ranking*:
assumes ⋃(*set xs*) = *A*
assumes *is-weak-ranking xs*
shows *total-preorder-on A* (*of-weak-ranking xs*)
proof
fix *x y* **assume** *x* ≼[*of-weak-ranking xs*] *y*
with *assms* **show** *x* ∈ *A y* ∈ *A*
by (*auto elim*!: *of-weak-ranking.cases*)
next
fix *x* **assume** *x* ∈ *A*
with *assms*(*1*) **obtain** *i* **where** *i* < *length xs x* ∈ *xs* ! *i*
by (*auto simp*: *set-conv-nth*)
thus *x* ≼[*of-weak-ranking xs*] *x* **by** (*auto intro*: *of-weak-ranking.intros*)
next
fix *x y* **assume** *x* ∈ *A y* ∈ *A*
with *assms*(*1*) **obtain** *i j* **where** *ij*: *i* < *length xs j* < *length xs x* ∈ *xs* ! *i y* ∈ *xs* ! *j*
by (*auto simp*: *set-conv-nth*)
consider *i* ≤ *j* | *j* ≤ *i* **by** *force*
thus *x* ≼[*of-weak-ranking xs*] *y* ∨ *y* ≼[*of-weak-ranking xs*] *x*
by *cases* (*insert ij*, (*blast intro*: *of-weak-ranking.intros*)+)
next
fix *x y z*
assume *A*: *x* ≼[*of-weak-ranking xs*] *y* **and** *B*: *y* ≼[*of-weak-ranking xs*] *z*
from *A* **obtain** *i j*
where *ij*: *i* ≥ *j i* < *length xs j* < *length xs x* ∈ *xs* ! *i y* ∈ *xs* ! *j*
by (*auto elim*!: *of-weak-ranking.cases*)

moreover from B **obtain** j' k
where $j'k$: $j' \geq k$ $j' < length\ xs$ $k < length\ xs$ $y \in xs\ !\ j'$ $z \in xs\ !\ k$
by (*auto elim*!: *of-weak-ranking.cases*)
moreover from *ij j'k is-weak-rankingD*[*OF assms*(*2*), *of j j'*]
have $j = j'$ **by** *blast*
ultimately show $x \preceq$[*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* ($\bigcup$(*set As*)) (*of-weak-ranking* (*A # As*)) = *of-weak-ranking As*
proof −
from *assms* **interpret** *R*: *total-preorder-on* $\bigcup$(*set As*) *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*::*nat*} (*of-weak-ranking* [{*1*,*3*},{*2*},{*4*}])
by (*simp add*: *of-weak-ranking-wf*)

context
fixes x :: *'alt set* **and** xs :: *'alt set list*
assumes *wf*: *is-weak-ranking* (*x#xs*)
begin

interpretation *R*: *total-preorder-on* $\bigcup$(*set* (*x#xs*)) *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 $a \in \bigcup$(*set xs*) $b \in \bigcup$(*set xs*)
using *assms* **by** (*fastforce elim*!: *of-weak-ranking.cases*)+

lemma *of-weak-ranking-Cons'*:
assumes $a \in \bigcup$(*set* (*x#xs*)) $b \in \bigcup$(*set* (*x#xs*))

shows *of-weak-ranking* (*x*#*xs*) *a b* ⟷ *b* ∈ *x* ∨ (*a* ∉ *x* ∧ *of-weak-ranking xs a b*)
proof
assume *of-weak-ranking* (*x* # *xs*) *a b*
with *wf of-weak-ranking-imp-in-set*[*of a b*]
show (*b* ∈ *x* ∨ *a* ∉ *x* ∧ *of-weak-ranking xs a b*)
by (*auto simp*: *is-weak-ranking-Cons of-weak-ranking-Cons*)
next
assume *b* ∈ *x* ∨ *a* ∉ *x* ∧ *of-weak-ranking xs a b*
with *assms* **show** *of-weak-ranking* (*x*#*xs*) *a b*
by (*fastforce simp*: *of-weak-ranking-Cons*)
qed

lemma *Max-wrt-among-of-weak-ranking-Cons1*:
assumes *x* ∩ *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* ⋃(*set xs*) *of-weak-ranking xs*
by (*intro total-preorder-of-weak-ranking*) (*simp-all add*: *is-weak-ranking-Cons*)
from *assms* **show** *?thesis*
by (*auto simp*: *R.Max-wrt-among-total-preorder*
R'.Max-wrt-among-total-preorder of-weak-ranking-Cons)
qed

lemma *Max-wrt-among-of-weak-ranking-Cons2*:
assumes *x* ∩ *A* ≠ {}
shows *Max-wrt-among* (*of-weak-ranking* (*x*#*xs*)) *A* = *x* ∩ *A*
proof −
from *wf* **interpret** *R'*: *total-preorder-on* ⋃(*set xs*) *of-weak-ranking xs*
by (*intro total-preorder-of-weak-ranking*) (*simp-all add*: *is-weak-ranking-Cons*)
from *assms* **obtain** *a* **where** *a* ∈ *x* ∩ *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* ∩ *A* = {} *then Max-wrt-among* (*of-weak-ranking xs*) *A else x* ∩ *A*)
using *Max-wrt-among-of-weak-ranking-Cons1 Max-wrt-among-of-weak-ranking-Cons2*
by *simp*

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

end

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

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

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

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

lemma *weak-ranking-aux-Union*:
A ⊆ *carrier* ⟹ ⋃(*set* (*weak-ranking-aux A*)) = *A*
proof (*induction A rule*: *weak-ranking-aux.induct* [*case-names empty nonempty*])
case (*nonempty A*)
with *Max-wrt-among-subset*[*of A*] **show** *?case* **by** *auto*

qed *simp-all*

lemma *weak-ranking-aux-wf*:
  $A \subseteq$ *carrier* $\Longrightarrow$ *is-weak-ranking* (*weak-ranking-aux A*)
proof (*induction A rule*: *weak-ranking-aux.induct* [*case-names empty nonempty*])
  case (*nonempty A*)
 have *is-weak-ranking* (*Max-wrt-among le A* # *weak-ranking-aux* (*A* − *Max-wrt-among le A*))
    unfolding *is-weak-ranking-Cons*
  proof (*intro conjI*)
    from *nonempty.prems nonempty.hyps* **show** *Max-wrt-among le A* ≠ {}
      by (*intro Max-wrt-among-nonempty*) *auto*
  next
   from *nonempty.prems* **show** *is-weak-ranking* (*weak-ranking-aux* (*A* − *Max-wrt-among le A*))
      by (*intro nonempty.IH*) *blast*
  next
    from *nonempty.prems nonempty.hyps* **have** *Max-wrt-among le A* ≠ {}
      by (*intro Max-wrt-among-nonempty*) *auto*
    moreover from *nonempty.prems*
    have $\bigcup$(*set* (*weak-ranking-aux* (*A* − *Max-wrt-among le A*))) = *A* − *Max-wrt-among le A*
      by (*intro weak-ranking-aux-Union*) *auto*
      ultimately show *Max-wrt-among le A* ∩ $\bigcup$(*set* (*weak-ranking-aux* (*A* − *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 $A \subseteq$ *carrier* $x \in A$ $y \in A$
  shows *of-weak-ranking* (*weak-ranking-aux A*) *x y* ⟷ *restrict-relation A le x y*
using *assms*
proof (*induction A rule*: *weak-ranking-aux.induct* [*case-names empty nonempty*])
  case (*nonempty A*)
  define *M* **where** *M* = *Max-wrt-among le A*
  from *nonempty.prems nonempty.hyps* **have** *M*: $M \subseteq A$ **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 ∈ M; cases y ∈ 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 ∈ carrier ∧ y ∈ 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 ... = le x y by auto
    finally show ?thesis .
  qed
qed

lemma weak-ranking-aux-unique':
  assumes ⋃(set As) ⊆ carrier is-weak-ranking As
          of-weak-ranking As = restrict-relation (⋃(set As)) le
  shows   As = weak-ranking-aux (⋃(set As))
using assms
proof (induction As)
  case (Cons A As)
  have restrict-relation (⋃(set As)) (of-weak-ranking (A # As)) = of-weak-ranking
```

```
As
    by (intro restrict-relation-of-weak-ranking-Cons Cons.prems)
  also have eq1: of-weak-ranking (A # As) = restrict-relation (⋃(set (A # As)))
le by fact
  finally have eq: of-weak-ranking As = restrict-relation (⋃(set As)) le
    by (simp add: Int-absorb2)
  with Cons.prems have eq2: weak-ranking-aux (⋃(set As)) = As
    by (intro sym [OF Cons.IH]) (auto simp: is-weak-ranking-Cons)

  from eq1 have
    Max-wrt-among le (⋃(set (A # As))) =
      Max-wrt-among (of-weak-ranking (A#As)) (⋃(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 (⋃(set (A # As))) = A .

  moreover from Cons.prems have A ≠ {} by (simp add: is-weak-ranking-Cons)
  ultimately have weak-ranking-aux (⋃(set (A # As))) = A # weak-ranking-aux
(A ∪ ⋃(set As) − A)
    using Cons.prems by simp
  also from Cons.prems have A ∪ ⋃(set As) − A = ⋃(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 ⋃(set As) of-weak-ranking As
    by (intro total-preorder-of-weak-ranking assms) simp-all
  from assms have x ∈ ⋃(set As) ⟷ x ∈ carrier for x
    using R.not-outside not-outside R.refl[of x] refl[of x]
    by blast
  hence eq: ⋃(set As) = carrier by blast
  from assms eq have As = weak-ranking-aux (⋃(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 ∃x. is-weak-ranking x ∧ le = of-weak-ranking x by auto
  hence is-weak-ranking (weak-ranking le) ∧ 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*) (*weak-ranking le*)
proof −
 from *assms* **have** *inv f* −‘ *carrier* = *carrier*
  by (*simp add*: *permutes-vimage permutes-inv*)
 then interpret *R*: *finite-total-preorder-on inv f* −‘ *carrier map-relation* (*inv f*)
le
  by (*intro finite-total-preorder-on-map*) (*simp-all add*: *finite-carrier*)
 from *assms* **have** *is-weak-ranking* (*map* ((‘) *f*) (*weak-ranking le*))
  by (*intro is-weak-ranking-map-inj*)
   (*simp-all add*: *weak-ranking-total-preorder permutes-inj-on*)
 with *assms* **show** *?thesis*
  by (*intro sym*[*OF R.weak-ranking-unique*])
   (*simp-all add*: *of-weak-ranking-permute weak-ranking-Union weak-ranking-total-preorder*)
qed

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

lemma *weak-ranking-eqclass2*:
assumes *A*: *A* ∈ *set* (*weak-ranking le*) *x* ∈ *A* **and** *le*: *le x y le y x*
shows *y* ∈ *A*
proof −
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 *A* **obtain** *k* **where** *k*: *k* < *length xs A* = *xs* ! *k*
by (*auto simp*: *xs-def set-conv-nth*)
ultimately have *k* = *i* **using** *ij i′j′ A*
by (*intro weak-ranking-index-unique*[*OF wf*, *of* - - *x*]) *auto*
with *ij i′j′ k eq* **show** *?thesis* **by** (*auto simp*: *xs-def*)
qed

lemma *hd-weak-ranking*:
assumes *x* ∈ *hd* (*weak-ranking le*) *y* ∈ *carrier*
shows *le y x*
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*) *y x* **using** *assms*(*1*)
by (*intro of-weak-ranking.intros*[*of 0 i*]) (*auto simp*: *hd-conv-nth*)
thus *?thesis* **by** (*simp add*: *weak-ranking-total-preorder*)
qed

lemma *last-weak-ranking*:

assumes $x \in last\ (weak\text{-}ranking\ le)$ $y \in carrier$
shows $le\ x\ y$
proof −
from *weak-ranking-Union assms* **obtain** i
where $i < length\ (weak\text{-}ranking\ le)$ $y \in weak\text{-}ranking\ le\ !\ i$
by (*auto simp*: *set-conv-nth*)
moreover from *assms*(*2*) *weak-ranking-Union* **have** $weak\text{-}ranking\ le \neq []$ **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-conv-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 $weak\text{-}ranking\text{-}index :: 'a \Rightarrow nat$ **where**
$weak\text{-}ranking\text{-}index\ x = find\text{-}index\ (\lambda A.\ x \in A)\ (weak\text{-}ranking\ le)$

lemma *nth-weak-ranking-index*:
assumes $x \in carrier$
shows $weak\text{-}ranking\text{-}index\ x < length\ (weak\text{-}ranking\ le)$
$x \in weak\text{-}ranking\ le\ !\ weak\text{-}ranking\text{-}index\ x$
proof −
from *assms weak-ranking-Union* **show** $weak\text{-}ranking\text{-}index\ x < length\ (weak\text{-}ranking\ le)$
unfolding *weak-ranking-index-def* **by** (*auto simp add*: *find-index-less-size-conv*)
thus $x \in weak\text{-}ranking\ le\ !\ weak\text{-}ranking\text{-}index\ x$ **unfolding** *weak-ranking-index-def*
by (*rule nth-find-index*)
qed

lemma *ranking-index-eqI*:
$i < length\ (weak\text{-}ranking\ le) \Longrightarrow x \in weak\text{-}ranking\ le\ !\ i \Longrightarrow weak\text{-}ranking\text{-}index\ x = i$
using *weak-ranking-index-unique'*[*of weak-ranking le i x*]
by (*simp add*: *weak-ranking-index-def weak-ranking-total-preorder*)

lemma *ranking-index-le-iff* [*simp*]:
assumes $x \in carrier$ $y \in carrier$
shows $weak\text{-}ranking\text{-}index\ x \geq weak\text{-}ranking\text{-}index\ y \longleftrightarrow le\ x\ y$
proof −
have $le\ x\ y \longleftrightarrow of\text{-}weak\text{-}ranking\ (weak\text{-}ranking\ le)\ x\ y$
by (*simp add*: *weak-ranking-total-preorder*)
also have $\ldots \longleftrightarrow weak\text{-}ranking\text{-}index\ x \geq weak\text{-}ranking\text{-}index\ y$
proof
assume $weak\text{-}ranking\text{-}index\ x \geq weak\text{-}ranking\text{-}index\ y$
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
    then obtain i j where
      i ≤ j i < length (weak-ranking le) j < length (weak-ranking le)
      x ∈ weak-ranking le ! j y ∈ weak-ranking le ! i
      by (elim of-weak-ranking.cases) blast
    with ranking-index-eqI[of i] ranking-index-eqI[of j]
      show weak-ranking-index x ≥ weak-ranking-index y by simp
  qed
  finally show ?thesis ..
qed

end

lemma weak-ranking-False [simp]: weak-ranking (λ- -. False) = []
proof −
  interpret finite-total-preorder-on {} λ- -. False
    by unfold-locales simp-all
  have [] = weak-ranking (λ- -. False) by (rule weak-ranking-unique) simp-all
  thus ?thesis ..
qed

lemmas of-weak-ranking-weak-ranking =
  finite-total-preorder-on.weak-ranking-total-preorder(2)

lemma finite-total-preorder-on-iff:
  finite-total-preorder-on A R ⟷ total-preorder-on A R ∧ finite A
  by (simp add: finite-total-preorder-on-def finite-total-preorder-on-axioms-def)

lemma finite-total-preorder-of-weak-ranking:
  assumes ⋃(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-finite-weak-ranking-def)
  from assms(1) and this interpret total-preorder-on A of-weak-ranking xs
    by (rule total-preorder-of-weak-ranking)
  from assms(2) show ?thesis
    by unfold-locales (simp add: assms(1)[symmetric] is-finite-weak-ranking-def)
qed

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

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

definition *preferred-alts* :: *'alt relation* ⇒ *'alt* ⇒ *'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** $A \neq \{\}$
using *weak-ranking-total-preorder*(*1*) *is-weak-ranking-iff* **by** *auto*
next
fix x y **assume** $x \in A$ $y \in A$
with *assms*
have $x \preceq$[*of-weak-ranking* (*weak-ranking le*)] y $y \preceq$[*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* ($\lambda x.\ \{x\}$) (*ranking le*)
unfolding *ranking-def map-map o-def*
proof (*rule sym*, *rule map-idI*)
fix A **assume** $A \in$ *set* (*weak-ranking le*)
hence *is-singleton A* **by** (*rule singleton-weak-ranking*)
thus {*the-elem A*} = A **by** (*auto elim*: *is-singletonE*)
qed

end

end

2 Preference Profiles

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

The type of preference profiles

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

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

lemma *not-in-dom'*: $i \notin$ *dom* ⟹ R i = (λ- -. *False*)
by (*simp add*: *fun-eq-iff*)

end

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

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

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

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

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

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

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

lemma *wf-permute-agents*:
  assumes *σ permutes agents*
  shows *pref-profile-wf agents alts* (*R* ∘ *σ*)
  unfolding *o-def* **using** *permutes-in-image*[*OF assms*(*1*)]

by (*intro pref-profile-wf.intro prefs-wf*) *simp-all*

lemma (**in** −) *pref-profile-eqI*:
assumes *pref-profile-wf agents alts R1 pref-profile-wf agents alts R2*
assumes $\bigwedge x.\ x \in agents \Longrightarrow R1\ x = R2\ x$
shows *R1 = R2*
proof
interpret *R1*: *pref-profile-wf agents alts R1* **by** *fact*
interpret *R2*: *pref-profile-wf agents alts R2* **by** *fact*
fix *x* **show** *R1 x = R2 x*
by (*cases x ∈ 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 σ R = (*λi x y. R i* (*inv σ x*) (*inv σ y*))

lemma *permute-profile-map-relation*:
permute-profile σ R = (*λi. map-relation* (*inv σ*) (*R i*))
by (*simp add*: *permute-profile-def map-relation-def*)

lemma *permute-profile-compose* [*simp*]:
permute-profile σ (*R* ∘ *π*) = *permute-profile σ R* ∘ *π*
by (*auto simp*: *fun-eq-iff permute-profile-def o-def*)

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

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

lemma (**in** *pref-profile-wf*) *wf-permute-alts*:
assumes *σ permutes alts*
shows *pref-profile-wf agents alts* (*permute-profile σ R*)
proof (*rule pref-profile-wf.intro*)
fix *i* **assume** *i ∈ agents*
with *assms* **interpret** *R*: *finite-total-preorder-on alts R i* **by** *simp*

from *assms* **have** [*simp*]: *inv σ x ∈ alts* ⟷ *x ∈ alts* **for** *x*
by (*simp add*: *permutes-in-image permutes-inv*)

show *finite-total-preorder-on alts* (*permute-profile σ R i*)
proof
fix *x y* **assume** *permute-profile σ R i x y*

```
    thus x ∈ alts y ∈ 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)
  qed (insert R.total R.refl R.finite-carrier, simp-all add: permute-profile-def)
qed (insert assms, simp-all add: permute-profile-def pref-profile-wf-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: Pareto-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** $x \preceq[R\ j]\ y$ **by** *simp*
hence $x \preceq[R\ j]\ x$ **using** *not-outside refl* **by** *blast*
with *A* **show** $x \preceq[Pareto(R)]\ y$ **by** (*auto simp*: *Pareto-def*)
qed

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

lemma *Pareto-strictI*:
assumes $\bigwedge i.\ i \in dom \Longrightarrow x \preceq[R\ i]\ y\ i \in dom\ x \prec[R\ i]\ y$
shows $x \prec[Pareto(R)]\ y$
using *assms* **by** (*auto simp*: *Pareto-strict-iff*)

lemma *Pareto-strictI'*:
assumes $\bigwedge i.\ i \in dom \Longrightarrow x \preceq[R\ i]\ y\ i \in dom \neg x \succeq[R\ i]\ y$
shows $x \prec[Pareto(R)]\ y$
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*)
qed

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

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

lemma *pareto-losersE*:

assumes $x \in$ *pareto-losers* R
obtains y **where** $y \in$ *carrier* $y \succ[Pareto(R)]\ x$
proof −
from *assms* **obtain** y **where** y: $y \succ[Pareto(R)]\ x$ **unfolding** *pareto-losers-def*
by *blast*
with *Pareto.not-outside*[*of* x y] **have** $y \in$ *carrier*
by (*simp add*: *strongly-preferred-def*)
with y **show** *?thesis* **using** *that* **by** *blast*
qed

end

2.2 Preferred alternatives

context *pref-profile-wf*
begin

lemma *preferred-alts-subset-alts*: *preferred-alts* $(R\ i)\ x \subseteq$ *alts* (**is** *?A*)
and *finite-preferred-alts* [*simp*,*intro*!]: *finite* (*preferred-alts* $(R\ i)\ x$) (**is** *?B*)
proof −
have *?A* ∧ *?B*
proof (*cases* $i \in$ *agents*)
assume $i \in$ *agents*
then interpret *total-preorder-on alts R i* **by** *simp*
have *preferred-alts* $(R\ i)\ x \subseteq$ *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*:
$i \in$ *agents* ⟹ *preferred-alts* $(R\ i)\ x = \{y \in alts.\ y \succeq[R\ i]\ x\}$
by (*simp add*: *preorder-on.preferred-alts-altdef*)

end

2.3 Favourite alternatives

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

lemma *favorites-no-agent* [*simp*]: *i ∉ agents ⟹ favorites R i = {}*
  by (*auto simp*: *favorites-def Max-wrt-def Max-wrt-among-def*)

lemma *favorites-altdef′*:
  favorites R i = {x∈alts. ∀ y∈alts. x ≽[R i] y}
proof (*cases i ∈ agents*)
  assume *i ∈ agents*
  then interpret *finite-total-preorder-on alts R i* **by** *simp*
  show *?thesis* **using** *Max-wrt-among-nonempty*[*of alts*] *Max-wrt-among-subset*[*of alts*]
    by (*auto simp*: *favorites-altdef Max-wrt-among-total-preorder*)
qed *simp-all*

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

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

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

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

lemma *has-unique-favorites-altdef*:

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

end

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* ∘ *R*) (*mset-set* {*i*. *R i* ≠ (λ- -. *False*)})

lemma (**in** *pref-profile-wf*) *agents-eq*:
agents = {*i*. *R i* ≠ (λ- -. *False*)}
proof *safe*
fix *i* **assume** *i*: *i* ∈ *agents* **and** *Ri*: *R i* = (λ- -. *False*)
from *i* **interpret** *preorder-on alts R i* **by** *simp*
from *carrier-eq Ri nonempty-alts* **show** *False* **by** *simp*
next
fix *i* **assume** *R i* ≠ (λ- -. *False*)
thus *i* ∈ *agents* **using** *prefs-undefined'*[*of i*] **by** (*cases i* ∈ *agents*) *auto*
qed

lemma (**in** *pref-profile-wf*) *anonymous-profile-def*:
anonymous-profile R = *image-mset* (*weak-ranking* ∘ *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 {#*weak-ranking* (*R x*). *x* ∈# ...#} =
{#*weak-ranking* (*R x*). *x* ∈# *mset-set agents*#} − {#*weak-ranking* (*R i*)#}
by (*subst image-mset-Diff*) (*simp-all add*: *in-multiset-in-set mset-subset-eq-single*)
also have {#*weak-ranking Ri'*#} + ... =
anonymous-profile R − {#*weak-ranking* (*R i*)#} + {#*weak-ranking Ri'*#}
by (*simp add*: *anonymous-profile-def add-ac o-def*)
finally show *?thesis* .
qed

2.5 Preference profiles from lists

definition *prefs-from-table* :: (*'agent* × *'alt set list*) *list* ⇒ (*'agent*, *'alt*) *pref-profile*
where
prefs-from-table xss = (λ*i*. *case-option* (λ- -. *False*) *of-weak-ranking* (*map-of xss i*))

definition *prefs-from-table-wf* **where**
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 *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 *prefs-from-table-wf agents alts xss*
using *assms* **unfolding** *prefs-from-table-wf-def* **by** *auto*

lemma *prefs-from-table-wfD*:
assumes *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** *prefs-from-table-wf-def* **by** *auto*

lemma *pref-profile-from-tableI*:
prefs-from-table-wf agents alts xss ⟹ *pref-profile-wf agents alts* (*prefs-from-table xss*)
proof (*intro pref-profile-wf.intro*)
assume *wf*: *prefs-from-table-wf agents alts xss*
fix *i* **assume** *i*: *i* ∈ *agents*
with *wf* **have** *i* ∈ *set* (*map fst xss*) **by** (*simp add*: *prefs-from-table-wf-def*)
then obtain *xs* **where** *xs*: *xs* ∈ *set* (*map snd xss*) *prefs-from-table xss i* =

```
of-weak-ranking xs
    by (cases map-of xss i)
     (fastforce dest: map-of-SomeD simp: prefs-from-table-def map-of-eq-None-iff)+
  with wf show finite-total-preorder-on alts (prefs-from-table xss i)
   by (auto simp: prefs-from-table-wf-def intro!: finite-total-preorder-of-weak-ranking)
next
  assume wf: prefs-from-table-wf agents alts xss
  fix i x y assume i: i ∉ agents
  with wf have i ∉ set (map fst xss) by (simp add: prefs-from-table-wf-def)
  hence map-of xss i = None by (simp add: map-of-eq-None-iff)
  thus ¬prefs-from-table xss i x y by (simp add: prefs-from-table-def)
qed (simp-all add: prefs-from-table-wf-def)

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

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

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

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

```
  proof cases
    assume k: k = i
    moreover from k have y = x if (i, y) ∈ set xs' for y
      using that by (auto simp: xs'-def split: if-splits)
    ultimately show ?thesis using assms(1) k xs'
      by (auto simp add: prefs-from-table-def map-of-eq-None-iff
            dest!: map-of-SomeD split: option.splits)
  next
    assume k: k ∉ set (map fst xs)
    with assms(1) have k': k ≠ i by auto
    with k xs' have map-of xs k = None map-of xs' k = None
      by (simp-all add: map-of-eq-None-iff)
    thus ?thesis by (simp add: prefs-from-table-def k')
  next
    assume k: k ≠ i k ∈ set (map fst xs)
    with k(1) have map-of xs k = map-of xs' k unfolding xs'-def
      by (induction xs) fastforce+
    with k show ?thesis by (simp add: prefs-from-table-def)
  qed
qed

lemma prefs-from-table-swap:
  x ≠ y ⟹ prefs-from-table ((x,x')#(y,y')#xs) = prefs-from-table ((y,y')#(x,x')#xs)
  by (intro ext) (auto simp: prefs-from-table-def)

lemma permute-prefs-from-table:
  assumes σ permutes fst ‘ set xs
  shows   prefs-from-table xs ∘ σ = prefs-from-table (map (λ(x,y). (inv σ x, y))
xs)
proof
  fix i
  have (prefs-from-table xs ∘ σ) i =
         (case map-of xs (σ i) of
            None ⇒ λ- -. False
          | Some x ⇒ of-weak-ranking x)
    by (simp add: prefs-from-table-def o-def)
  also have map-of xs (σ i) = map-of (map (λ(x,y). (inv σ x, y)) xs) i
    using map-of-permute[OF assms] by (simp add: o-def fun-eq-iff)
  finally show (prefs-from-table xs ∘ σ) i = prefs-from-table (map (λ(x,y). (inv σ
x, y)) xs) i
    by (simp only: prefs-from-table-def)
qed

lemma permute-profile-from-table:
  assumes wf: prefs-from-table-wf agents alts xss
  assumes perm: σ permutes alts
  shows   permute-profile σ (prefs-from-table xss) =
          prefs-from-table (map (λ(x,y). (x, map ((‘) σ) y)) xss) (is ?f = ?g)
proof
```

fix *i*
have *wf'*: *prefs-from-table-wf agents alts* (*map* (λ(*x*, *y*). (*x*, *map* ((‘) σ) *y*)) *xss*)
proof (*intro prefs-from-table-wfI*, *goal-cases*)
case (*5 xs*)
then obtain *y* **where** *y* ∈ *set xss xs* = *map* ((‘) σ) (*snd y*)
by (*auto simp add*: *o-def case-prod-unfold*)
with *assms* **show** *?case*
by (*simp add*: *image-Union* [*symmetric*] *prefs-from-table-wf-def permutes-image o-def case-prod-unfold*)
next
case (*6 xs*)
then obtain *y* **where** *y* ∈ *set xss xs* = *map* ((‘) σ) (*snd y*)
by (*auto simp add*: *o-def case-prod-unfold*)
with *assms* **show** *?case*
by (*auto simp*: *is-finite-weak-ranking-def is-weak-ranking-iff prefs-from-table-wf-def distinct-map permutes-inj-on inj-on-image intro*!: *disjoint-image*)
qed (*insert assms*, *simp-all add*: *image-Union* [*symmetric*] *prefs-from-table-wf-def permutes-image o-def case-prod-unfold*)
show *?f i* = *?g i*
proof (*cases i* ∈ *agents*)
assume *i* ∉ *agents*
with *assms wf'* **show** *?thesis*
by (*simp add*: *permute-profile-def prefs-from-table-undef*)
next
assume *i*: *i* ∈ *agents*
define *xs* **where** *xs* = *the* (*map-of xss i*)
from *i wf* **have** *xs*: *map-of xss i* = *Some xs*
by (*cases map-of xss i*) (*auto simp*: *prefs-from-table-wf-def xs-def*)
have *xs-in-xss*: *xs* ∈ *snd* ‘ *set xss*
using *xs* **by** (*force dest*!: *map-of-SomeD*)
with *wf* **have** *set-xs*: ⋃(*set xs*) = *alts*
by (*simp add*: *prefs-from-table-wfD*)

from *i* **have** *prefs-from-table* (*map* (λ(*x*,*y*). (*x*, *map* ((‘) σ) *y*)) *xss*) *i* = *of-weak-ranking* (*the* (*map-of* (*map* (λ(*x*,*y*). (*x*, *map* ((‘) σ) *y*)) *xss*) *i*))
using *wf'* **by** (*intro prefs-from-table-map-of*) *simp-all*
also have ... = *of-weak-ranking* (*map* ((‘) σ) *xs*)
by (*subst map-of-map*) (*simp add*: *xs*)
also have ... = (λ*a b*. *of-weak-ranking xs* (*inv* σ *a*) (*inv* σ *b*))
by (*intro ext*) (*simp add*: *of-weak-ranking-permute map-relation-def set-xs perm*)
also have ... = *permute-profile* σ (*prefs-from-table xss*) *i*
by (*simp add*: *prefs-from-table-def xs permute-profile-def*)
finally show *?thesis* **..**
qed
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 = {*y*. *of-weak-ranking xs y x*}

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*) = (λ*y*.
(*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 (*simp add*: *pref-profile-from-tableI*)

```
  from assms have agents: agents = fst ‘ set xs
    by (simp add: prefs-from-table-wf-def)
  hence [simp]: finite agents by auto
  have anonymous-profile (prefs-from-table xs) =
        {#weak-ranking (prefs-from-table xs x). x ∈# mset-set agents#}
    by (simp add: o-def anonymous-profile-def)
  also from assms have ... = {#the (map-of xs i). i ∈# mset-set agents#}
  proof (intro image-mset-cong)
    fix i assume i: i ∈# mset-set agents
    from i assms
      have weak-ranking (prefs-from-table xs i) =
            weak-ranking (of-weak-ranking (the (map-of xs i)))
      by (simp add: prefs-from-table-map-of)
    also from assms i have ... = the (map-of xs i)
      by (intro weak-ranking-of-weak-ranking)
         (auto simp: prefs-from-table-wf-def)
    finally show weak-ranking (prefs-from-table xs i) = the (map-of xs i) .
  qed
  also from agents have mset-set agents = mset-set (set (map fst xs)) by simp
  also from assms have ... = mset (map fst xs)
    by (intro mset-set-set) (simp-all add: prefs-from-table-wf-def)
  also from assms have {#the (map-of xs i). i ∈# mset (map fst xs)#} = mset
(map snd xs)
    by (intro image-mset-map-of) (simp-all add: prefs-from-table-wf-def)
  finally show ?thesis .
qed

lemma prefs-from-table-agent-permutation:
  assumes wf: prefs-from-table-wf agents alts xs prefs-from-table-wf agents alts ys
  assumes mset-eq: mset (map snd xs) = mset (map snd ys)
  obtains π where π permutes agents prefs-from-table xs ∘ π = prefs-from-table
ys
proof −
  from wf(1) have agents: agents = set (map fst xs)
    by (simp-all add: prefs-from-table-wf-def)
  from wf(2) have agents': agents = set (map fst ys)
    by (simp-all add: prefs-from-table-wf-def)
  from agents agents' wf(1) wf(2) have mset (map fst xs) = mset (map fst ys)
   by (subst set-eq-iff-mset-eq-distinct [symmetric]) (simp-all add: prefs-from-table-wfD)
  hence same-length: length xs = length ys by (auto dest: mset-eq-length simp del:
mset-map)

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

  from mset-eq g
    have mset (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* ⟷ *x* < *length xs*
f x < *length ys* ⟷ *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* ∈ *agents* **for** *x*
using *that* **by** (*simp add*: *idx-def agents*)
have [*simp*]: *unidx i* ∈ *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* ∈ *agents* **for** *x*
using *x* **unfolding** *idx-def unidx-def* **using** *nth-index*[*of x map fst xs*]
by (*simp add*: *agents set-map* [*symmetric*] *nth-map* [*symmetric*] *del*: *set-map*)
have *idx-unidx*: *idx* (*unidx i*) = *i* **if** *i*: *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 *π* **where** *π x* = (*if x* ∈ *agents then* (*unidx* ∘ *f* ∘ *idx*) *x else x*) **for** *x*
define *π′* **where** *π′ x* = (*if x* ∈ *agents then* (*unidx* ∘ *inv f* ∘ *idx*) *x else x*) **for** *x*
have *bij-betw* (*unidx* ∘ *f* ∘ *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-wf-def same-length*)
also have *?P* ⟷ *bij-betw π agents agents*
by (*intro bij-betw-cong*) (*simp add*: *π-def*)
finally have *perm*: *π permutes agents*
by (*intro bij-imp-permutes*) (*simp-all add*: *π-def*)

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

have *inv-π*: *inv π* = *π′*
proof (*rule permutes-invI*[*OF perm*])
fix *x* **assume** *x* ∈ *agents*
with *f*(*1*) **show** *π′* (*π x*) = *x*
by (*simp add*: *π-def π′-def idx-unidx unidx-idx inv-f-f permutes-inj*)
qed (*simp add*: *π-def π′-def*)
with *perm* **have** *inv-π′*: *inv π′* = *π* **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-eqI*)

```
    (simp-all add: prefs-from-table-wfD permute-list-map [symmetric])
  also have permute-list h ys = permute-list h (zip (map fst ys) (map snd ys))
    by (simp add: zip-map-fst-snd)
  also from same-length f g
    have permute-list h (zip (map fst ys) (map snd ys)) =
          zip (permute-list f (map fst xs)) (map snd xs)
    by (subst permute-list-zip[OF h]) (simp-all add: h-def permute-list-compose)
  also {
    fix i assume i: i < length xs
    from i have permute-list f (map fst xs) ! i = unidx (f i)
      using permutes-in-image[OF f(1)] f(1)
      by (subst permute-list-nth) (simp-all add: same-length unidx-def)
    also from i have ... = 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 ∘ π ∘ index xs) x else x) for x
  have bij-betw ((!) xs ∘ π ∘ 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 xs)
  also have ?P ⟷ bij-betw π' A A
```

by (*intro bij-betw-cong*) (*simp-all add*: π'*-def*)
finally have π' *permutes A*
by (*rule bij-imp-permutes*) (*simp-all add*: π'*-def*)
moreover from π *xs*(*1*)[*symmetric*] *xs*(*2*) **have** $g\ (\pi'\ x) = f\ x$ **if** $x \in A$ **for** x
by (*simp add*: *permute-list-map permute-list-distinct*
permutes-image π'*-def that atLeast0LessThan*)
ultimately show *?thesis* **by** (*rule that*)
qed

lemma *anonymous-profile-agent-permutation*:
assumes *eq*: *anonymous-profile R1* = *anonymous-profile R2*
assumes *wf*: *pref-profile-wf agents alts R1 pref-profile-wf agents alts R2*
assumes *fin*: *finite agents*
obtains π **where** π *permutes agents R2* $\circ$ π = *R1*
proof −
interpret *R1*: *pref-profile-wf agents alts R1* **by** *fact*
interpret *R2*: *pref-profile-wf agents alts R2* **by** *fact*

from *eq* **have** {#*weak-ranking* (*R1 x*). *x* ∈# *mset-set agents*#} =
{#*weak-ranking* (*R2 x*). *x* ∈# *mset-set agents*#}
by (*simp add*: *R1.anonymous-profile-def R2.anonymous-profile-def o-def*)
from *image-mset-eq-permutation*[*OF this fin*] **obtain** π
where π: π *permutes agents*
⋀*x*. *x* ∈ *agents* ⟹ *weak-ranking* (*R2* (π *x*)) = *weak-ranking* (*R1 x*) **by** *auto*
from π **have** *wf*$'$: *pref-profile-wf agents alts* (*R2* $\circ$ π)
by (*intro R2.wf-permute-agents*)
then interpret *R2*$'$: *pref-profile-wf agents alts R2* $\circ$ π **.**
have *R2* $\circ$ π = *R1*
proof (*intro pref-profile-eqI*[*OF wf*$'$ *wf*(*1*)])
fix *x* **assume** *x*: *x* ∈ *agents*
with π **have** *weak-ranking* ((*R2 o* π) *x*) = *weak-ranking* (*R1 x*) **by** *simp*
with *wf*$'$ *wf*(*1*) *x* **show** (*R2* $\circ$ π) *x* = *R1 x*
by (*intro weak-ranking-eqD*[*of alts*] *R2*$'$*.prefs-wf*) *simp-all*
qed
from π(*1*) **and** *this* **show** *?thesis* **by** (*rule that*)
qed

end
theory *Elections*
imports *Preference-Profiles*
begin

An election consists of a finite set of agents and a finite non-empty set of alternatives.

locale *election* =
fixes *agents* :: *'agent set* **and** *alts* :: *'alt set*
assumes *finite-agents* [*simp*, *intro*]: *finite agents*
assumes *finite-alts* [*simp*, *intro*]: *finite alts*
assumes *nonempty-agents* [*simp*]: *agents* ≠ {}

assumes *nonempty-alts* [*simp*]: *alts* $\neq$ {}
begin

abbreviation *is-pref-profile* $\equiv$ *pref-profile-wf agents alts*

lemma *finite-total-preorder-on-iff'* [*simp*]:
finite-total-preorder-on alts R $\longleftrightarrow$ *total-preorder-on alts R*
by (*simp add*: *finite-total-preorder-on-iff*)

lemma *pref-profile-wfI'* [*intro?*]:
($\bigwedge i$. $i \in$ *agents* $\Longrightarrow$ *total-preorder-on alts* (*R i*)) $\Longrightarrow$
($\bigwedge i$. $i \notin$ *agents* $\Longrightarrow$ *R i* = (λ- -. *False*)) $\Longrightarrow$ *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* $\in$ *agents*
shows *is-pref-profile* (*R*(*i* := *Ri'*))
using *assms* **by** (*auto intro*!: *pref-profile-wf.wf-update*)

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

context
fixes *R* **assumes** *R*: *total-preorder-on alts R*
begin

interpretation *R*: *total-preorder-on alts R* **by** *fact*

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

lemma *Max-wrt-prefs-nonempty*: *Max-wrt R* $\neq$ {}
using *R.Max-wrt-nonempty* **by** *simp*

lemma *maximal-imp-preferred*:
x $\in$ *alts* $\Longrightarrow$ *Max-wrt R* $\subseteq$ *preferred-alts R x*
using *R.total*
by (*auto simp*: *R.Max-wrt-total-preorder preferred-alts-def strongly-preferred-def*)

end

end

end

3 Auxiliary facts about PMFs

theory *Lotteries*

imports *Complex-Main HOL−Probability.Probability*
begin

The type of lotteries (a probability mass function)

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

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

lemma *pmf-of-set-lottery*:
A ≠ {} ⟹ finite A ⟹ A ⊆ B ⟹ pmf-of-set A ∈ lotteries-on B
unfolding *lotteries-on-def* **by** *auto*

lemma *pmf-of-list-lottery*:
pmf-of-list-wf xs ⟹ set (map fst xs) ⊆ A ⟹ pmf-of-list xs ∈ lotteries-on A
using *set-pmf-of-list[of xs]* **by** (*auto simp*: *lotteries-on-def*)

lemma *return-pmf-in-lotteries-on* [*simp*,*intro*]:
x ∈ A ⟹ return-pmf x ∈ lotteries-on A
by (*simp add*: *lotteries-on-def*)

end
theory *Utility-Functions*
imports
Complex-Main
HOL−Probability.Probability
Lotteries
Preference-Profiles
begin

3.1 Definition of von Neumann–Morgenstern utility functions

locale *vnm-utility = finite-total-preorder-on +*
fixes *u* :: *'a ⇒ real*
assumes *utility-le-iff*: *x ∈ carrier ⟹ y ∈ carrier ⟹ u x ≤ u y ⟷ x ⪯[le] y*
begin

lemma *utility-le*: *x ⪯[le] y ⟹ u x ≤ u y*
using *not-outside[of x y] utility-le-iff* **by** *simp*

lemma *utility-less-iff*:
x ∈ carrier ⟹ y ∈ carrier ⟹ u x < u y ⟷ x ≺[le] y
using *utility-le-iff[of x y] utility-le-iff[of y x]*
by (*auto simp*: *strongly-preferred-def*)

lemma *utility-less*: *x ≺[le] y ⟹ u x < u y*
using *not-outside[of x y] utility-less-iff* **by** (*simp add*: *strongly-preferred-def*)

The following lemma allows us to compute the expected utility by summing

over all indifference classes, using the fact that alternatives in the same indifference class must have the same utility.

lemma *expected-utility-weak-ranking*:
assumes $p \in$ *lotteries-on carrier*
shows *measure-pmf.expectation p u* =
$(\sum A \leftarrow$*weak-ranking le*. *u* (*SOME x*. $x \in A$) $*$ *measure-pmf.prob p A*)
proof −
from *assms* **have** *measure-pmf.expectation p u* = $(\sum a \in$*carrier*. *u a* $*$ *pmf p a*)
by (*subst integral-measure-pmf*[*OF finite-carrier*])
(*auto simp*: *lotteries-on-def ac-simps*)
also have *carrier*: *carrier* = $\bigcup$(*set* (*weak-ranking le*)) **by** (*simp add*: *weak-ranking-Union*)
also from *carrier* **have** *finite*: *finite A* **if** $A \in$ *set* (*weak-ranking le*) **for** *A*
using *that* **by** (*blast intro*!: *finite-subset*[*OF - finite-carrier*, *of A*])
hence $(\sum a \in \bigcup$(*set* (*weak-ranking le*)). *u a* $*$ *pmf p a*) =
$(\sum A \leftarrow$*weak-ranking le*. $\sum a \in A$. *u a* $*$ *pmf p a*) (**is** - = *sum-list ?xs*)
using *weak-ranking-total-preorder*
by (*subst sum.Union-disjoint*)
(*auto simp*: *is-weak-ranking-iff disjoint-def sum.distinct-set-conv-list*)
also have *?xs* = *map* (λA. $\sum a \in A$. *u* (*SOME a*. $a \in A$) $*$ *pmf p a*) (*weak-ranking le*)
proof (*intro map-cong HOL.refl sum.cong*)
fix *x A* **assume** *x*: $x \in A$ **and** *A*: $A \in$ *set* (*weak-ranking le*)
have (*SOME x*. $x \in A$) $\in A$ **by** (*rule someI-ex*) (*insert x*, *blast*)
from *weak-ranking-eqclass1*[*OF A x this*] *weak-ranking-eqclass1*[*OF A this x*] *x this A*
have *u x* = *u* (*SOME x*. $x \in A$)
by (*intro antisym*; *subst utility-le-iff*) (*auto simp*: *carrier*)
thus *u x* $*$ *pmf p x* = *u* (*SOME x*. $x \in A$) $*$ *pmf p x* **by** *simp*
qed
also have ... = *map* (λA. *u* (*SOME a*. $a \in A$) $*$ *measure-pmf.prob p A*) (*weak-ranking le*)
using *finite* **by** (*intro map-cong HOL.refl*)
(*auto simp*: *sum-distrib-left measure-measure-pmf-finite*)
finally show *?thesis* **.**
qed

lemma *scaled*: $c > 0 \Longrightarrow$ *vnm-utility carrier le* (λx. c $*$ *u x*)
by *unfold-locales* (*insert utility-le-iff*, *auto*)

lemma *add-right*:
assumes $\bigwedge x\ y$. *le x y* $\Longrightarrow$ $f\ x \leq f\ y$
shows *vnm-utility carrier le* (λx. *u x* + *f x*)
proof
fix *x y* **assume** *xy*: $x \in$ *carrier* $y \in$ *carrier*
from *assms*[*of x y*] *utility-le-iff*[*OF xy*] *assms*[*of y x*] *utility-le-iff*[*OF xy*(*2*,*1*)]
show (*u x* + *f x* ≤ *u y* + *f y*) = *le x y* **by** *auto*
qed

lemma *add-left*:

$(\bigwedge x\ y.\ le\ x\ y \Longrightarrow f\ x \leq f\ y) \Longrightarrow$ *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 ε), again yielding a consistent utility function.

lemma *add-epsilon*:
assumes *A*: $\bigwedge x\ y.\ le\ x\ y \Longrightarrow le\ y\ x \Longrightarrow f\ x = f\ y$
shows $\exists \varepsilon{>}0.$ *vnm-utility carrier le* $(\lambda x.\ u\ x + \varepsilon * f\ x)$
proof −
let *?A* = $\{(u\ y - u\ x)\ /\ (f\ x - f\ y)\ |x\ y.\ x \prec[le]\ y \wedge f\ x > f\ y\}$
have *?A* = $(\lambda(x,y).\ (u\ y - u\ x)\ /\ (f\ x - f\ y))$ ‘ $\{(x,y)\ |x\ y.\ x \prec[le]\ y \wedge f\ x > f\ y\}$ **by** *auto*
also have *finite* $\{(x,y)\ |x\ y.\ x \prec[le]\ y \wedge f\ x > f\ y\}$
by (*rule finite-subset*[*of* - *carrier* × *carrier*])
(*insert not-outside*, *auto simp*: *strongly-preferred-def*)
hence *finite* $((\lambda(x,y).\ (u\ y - u\ x)\ /\ (f\ x - f\ y))$ ‘ $\{(x,y)\ |x\ y.\ x \prec[le]\ y \wedge f\ x > f\ y\})$
by *simp*
finally have *finite*: *finite ?A* **.**

define ε **where** ε = *Min* (*insert 1 ?A*) / *2*
from *finite* **have** *Min* (*insert 1 ?A*) > *0*
by (*auto intro*!: *divide-pos-pos simp*: *utility-less*)
hence ε: $\varepsilon > 0$ **unfolding** ε*-def* **by** *simp*

have *mono*: $u\ x + \varepsilon * f\ x < u\ y + \varepsilon * f\ y$ **if** *xy*: $x \prec[le]\ y$ **for** *x y*
proof (*cases f x* > *f y*)
assume *less*: *f x* > *f y*
from ε **have** $\varepsilon <$ *Min* (*insert 1 ?A*) **unfolding** ε*-def* **by** *linarith*
also from *less xy finite* **have** *Min* (*insert 1 ?A*) $\leq (u\ y - u\ x)\ /\ (f\ x - f\ y)$ **unfolding** ε*-def*
by (*intro Min-le*) *auto*
finally show *?thesis* **using** *less* **by** (*simp add*: *field-simps*)
next
assume $\neg f\ x > f\ y$
with *utility-less*[*OF xy*] ε **show** *?thesis*
by (*simp add*: *algebra-simps not-less add-less-le-mono*)
qed
have *eq*: $u\ x + \varepsilon * f\ x = u\ y + \varepsilon * f\ y$ **if** *xy*: $x \preceq[le]\ y\ y \preceq[le]\ x$ **for** *x y*
using *xy*[*THEN utility-le*] *A*[*OF xy*] **by** *simp*
have *vnm-utility carrier le* $(\lambda x.\ u\ x + \varepsilon * f\ x)$
proof
fix *x y* **assume** *xy*: $x \in$ *carrier* $y \in$ *carrier*
show $(u\ x + \varepsilon * f\ x \leq u\ y + \varepsilon * f\ y) \longleftrightarrow le\ x\ y$
using *total*[*OF xy*] *mono*[*of x y*] *mono*[*of y x*] *eq*[*of x y*]
by (*cases le x y*; *cases le y x*) (*auto simp*: *strongly-preferred-def*)
qed
from ε *this* **show** *?thesis* **by** *blast*

qed

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

end

end

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* :: *'alt relation* ⇒ *'alt lottery relation* **where**
$p \succeq[SD(R)]\ q \longleftrightarrow p \in lotteries\text{-}on\ \{x.\ R\ x\ x\} \wedge q \in lotteries\text{-}on\ \{x.\ R\ x\ x\} \wedge$
$(\forall x.\ R\ x\ x \longrightarrow measure\text{-}pmf.prob\ p\ \{y.\ y \succeq[R]\ x\} \geq measure\text{-}pmf.prob\ q\ \{y.\ y \succeq[R]\ x\})$

lemma *SD-empty* [*simp*]: $SD\ (\lambda\text{-}\ \text{-}.\ False) = (\lambda\text{-}\ \text{-}.\ False)$
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-refl*: $p \preceq[SD(R)]\ p \longleftrightarrow p \in lotteries\text{-}on\ \{x.\ R\ x\ x\}$
by (*simp add*: *SD-def*)

lemma *SD-trans* [*simp*, *trans*]: $p \preceq[SD(R)]\ q \Longrightarrow q \preceq[SD(R)]\ r \Longrightarrow p \preceq[SD(R)]\ r$
unfolding *SD-def* **by** (*auto intro*: *order.trans*)

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

context *preorder-on*
begin

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

lemma *SD-preorderI* [*intro?*]:
  assumes $p \in$ *lotteries-on carrier* $q \in$ *lotteries-on carrier*
  assumes $\bigwedge x.\ x \in$ *carrier* $\Longrightarrow$
    measure-pmf.prob p (*preferred-alts le x*) $\geq$ *measure-pmf.prob q*
(*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 $p \in$ *lotteries-on carrier* $q \in$ *lotteries-on carrier*
  and $\bigwedge x.\ x \in$ *carrier* $\Longrightarrow$
    measure-pmf.prob p (*preferred-alts le x*) $\geq$ *measure-pmf.prob q*
(*preferred-alts le x*)
  using *assms* **unfolding** *SD-preorder* **by** *simp-all*

lemma *SD-refl'* [*simp*]: $p \preceq[SD(le)]\ p \longleftrightarrow p \in$ *lotteries-on carrier*
  by (*simp add*: *SD-def carrier-eq*)

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

lemma *SD-singleton-left*:
  assumes $x \in$ *carrier* $q \in$ *lotteries-on carrier*
  shows *return-pmf x* $\preceq[SD(le)]\ q \longleftrightarrow (\forall\, y{\in}$*set-pmf q*. $x \preceq[le]\ y)$
proof
  assume *SD*: *return-pmf x* $\preceq[SD(le)]\ q$
  from *assms SD-preorderD*(*3*)[*OF SD*, *of x*]
    have *measure-pmf.prob* (*return-pmf x*) (*preferred-alts le x*) $\leq$
        measure-pmf.prob q (*preferred-alts le x*) **by** *simp*
  also from *assms* **have** *measure-pmf.prob* (*return-pmf x*) (*preferred-alts le x*) =
1
    by (*simp add*: *indicator-def*)
  finally have *AE y in q*. $y \succeq[le]\ x$
   by (*simp add*: *measure-pmf.measure-ge-1-iff measure-pmf.prob-eq-1 preferred-alts-def*)
  thus $\forall\, y{\in}$*set-pmf q*. $y \succeq[le]\ x$ **by** (*simp add*: *AE-measure-pmf-iff*)
next
  assume *A*: $\forall\, y{\in}$*set-pmf q*. $x \preceq[le]\ y$
  show *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 (simp-all add: preferred-alts-def indicator-def measure-nonneg)
  qed (insert assms, simp-all add: lotteries-on-def)
qed

lemma SD-singleton-right:
  assumes x: x ∈ carrier and q: q ∈ lotteries-on carrier
  shows   q ≼[SD(le)] return-pmf x ⟷ (∀ y∈set-pmf q. y ≼[le] x)
proof safe
  fix y assume SD: q ≼[SD(le)] return-pmf x and y: y ∈ set-pmf q
  from y assms have [simp]: y ∈ carrier by (auto simp: lotteries-on-def)

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

lemma SD-strict-singleton-left:
  assumes x ∈ carrier 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 $x \in$ *carrier* $q \in$ *lotteries-on carrier*
shows $q \prec[SD(le)]$ *return-pmf* $x \longleftrightarrow (\forall\, y{\in}$*set-pmf* $q.\ y \preceq[le]\ x) \wedge (\exists\, y{\in}$*set-pmf* $q.\ (y \prec[le]\ x))$
using *assms* **by** (*auto simp add*: *strongly-preferred-def SD-singleton-left SD-singleton-right*)

lemma *SD-singleton* [*simp*]:
$x \in$ *carrier* $\Longrightarrow y \in$ *carrier* $\Longrightarrow$ *return-pmf* $x \preceq[SD(le)]$ *return-pmf* $y \longleftrightarrow x \preceq[le]$ y
by (*subst SD-singleton-left*) (*simp-all add*: *lotteries-on-def*)

lemma *SD-strict-singleton* [*simp*]:
$x \in$ *carrier* $\Longrightarrow y \in$ *carrier* $\Longrightarrow$ *return-pmf* $x \prec[SD(le)]$ *return-pmf* $y \longleftrightarrow x \prec[le]$ y
by (*simp add*: *strongly-preferred-def*)

end

context *pref-profile-wf*
begin

context
fixes i **assumes** i: $i \in$ *agents*
begin

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

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

end
end

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

4.2 Stochastic Dominance for preference profiles

context *pref-profile-wf*
begin

lemma *SD-pref-profile*:
assumes $i \in$ *agents*
shows $p \succeq[SD(R\ i)]\ q \longleftrightarrow p \in$ *lotteries-on alts* $\wedge\ q \in$ *lotteries-on alts* $\wedge$
$(\forall\, x{\in}$*alts*. *measure-pmf.prob* p (*preferred-alts* $(R\ i)\ x) \geq$
measure-pmf.prob q (*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** *not-outside*
by (*auto simp*: *preferred-alts-def*)
thus *?thesis* **by** (*simp add*: *SD-preorder preferred-alts-def*)
qed

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

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

end

4.3 SD efficient lotteries

definition *SD-efficient* :: ('*agent*, '*alt*) *pref-profile* ⇒ '*alt lottery* ⇒ *bool* **where**
SD-efficient-auxdef:
SD-efficient R p ⟷ ¬(∃ *q*∈*lotteries-on* {*x*. ∃ *i*. *R i x x*}. *q* ≻[*Pareto* (*SD* ∘ *R*)] *p*)

context *pref-profile-wf*
begin

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

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

lemma *SD-efficient-def*:
SD-efficient R p ⟷ ¬(∃ *q*∈*lotteries-on alts*. *q* ≻[*Pareto* (*SD* ∘ *R*)] *p*)
proof −
have *SD-efficient R p* ⟷ ¬(∃ *q*∈*lotteries-on* {*x*. ∃ *i*. *R i x x*}. *q* ≻[*Pareto* (*SD* ∘ *R*)] *p*)
unfolding *SD-efficient-auxdef* ..
also from *nonempty-agents* **obtain** *i* **where** *i*: *i* ∈ *agents* **by** *blast*

with *preorder-on.refl*[*of alts R i*]
have $\{x.\ \exists\, i.\ R\ i\ x\ x\} = alts$ **by** (*auto intro*!: *exI*[*of - i*] *not-outside*)
finally show *?thesis* **.**
qed

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

lemma *SD-inefficientI*:
assumes $q \in lotteries\text{-}on\ alts\ \bigwedge i.\ i \in agents \Longrightarrow q \succeq[SD(R\ i)]\ p$
$i \in agents\ q \succ[SD(R\ i)]\ p$
shows $\neg SD\text{-}efficient\ R\ p$
using *assms* **unfolding** *SD-efficient-def'* **by** *blast*

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

lemma *SD-inefficientE*:
assumes $\neg SD\text{-}efficient\ R\ p$
obtains $q\ i$ **where**
$q \in lotteries\text{-}on\ alts\ \bigwedge i.\ i \in agents \Longrightarrow q \succeq[SD(R\ i)]\ p$
$i \in agents\ q \succ[SD(R\ i)]\ p$
using *assms* **unfolding** *SD-efficient-def'* **by** *blast*

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

lemma *SD-efficient-singleton-iff*:
assumes [*simp*]: $x \in alts$
shows $SD\text{-}efficient\ R\ (return\text{-}pmf\ x) \longleftrightarrow x \notin pareto\text{-}losers\ R$
proof −
{
assume $x \in pareto\text{-}losers\ R$
then obtain y **where** $y \in alts\ x \prec[Pareto\ R]\ y$
by (*rule pareto-losersE*)
then have $\neg SD\text{-}efficient\ R\ (return\text{-}pmf\ x)$
by (*intro SD-inefficientI'*[*of return-pmf y*]) (*force simp*: *Pareto-strict-iff*)+
} moreover {
assume $\neg SD\text{-}efficient\ R\ (return\text{-}pmf\ x)$

from *SD-inefficientE*[*OF this*] **obtain** *q i*
where *q*:
q ∈ *lotteries-on alts*
⋀*i*. *i* ∈ *agents* ⟹ *SD* (*R i*) (*return-pmf x*) *q*
i ∈ *agents*
return-pmf x ≺[*SD* (*R i*)] *q*
by *blast*
from *q* **obtain** *y* **where** *y* ∈ *set-pmf q y* ≻[*R i*] *x*
by (*auto simp*: *SD-strict-singleton-left*)
with *q* **have** *y* ≻[*Pareto*(*R*)] *x*
by (*fastforce simp*: *Pareto-strict-iff SD-singleton-left*)
hence *x* ∈ *pareto-losers R* **by** *simp*
}
ultimately show *?thesis* **by** *blast*
qed

end

4.4 Equivalence proof

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

context *finite-total-preorder-on*
begin

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

lemma *utility-weak-ranking-index*:
is-vnm-utility (λ*x*. *real* (*length* (*weak-ranking le*) − *weak-ranking-index x*))
proof
fix *x y* **assume** *xy*: *x* ∈ *carrier y* ∈ *carrier*
with *this*[*THEN nth-weak-ranking-index*(*1*)] *this*[*THEN nth-weak-ranking-index*(*2*)]
show (*real* (*length* (*weak-ranking le*) − *weak-ranking-index x*)
≤ *real* (*length* (*weak-ranking le*) − *weak-ranking-index y*)) ⟷ *le x y*
by (*simp add*: *le-diff-iff'*)
qed

lemma *SD-iff-expected-utilities-le*:
assumes *p* ∈ *lotteries-on carrier q* ∈ *lotteries-on carrier*
shows *p* ≼[*SD*(*le*)] *q* ⟷
(∀ *u*. *is-vnm-utility u* ⟶ *measure-pmf.expectation p u* ≤ *measure-pmf.expectation q u*)
proof *safe*
fix *u* **assume** *SD*: *p* ≼[*SD*(*le*)] *q* **and** *is-utility*: *is-vnm-utility u*
from *is-utility* **interpret** *vnm-utility carrier le u* **.**
define *xs* **where** *xs* = *weak-ranking le*
have *le*: *is-weak-ranking xs le* = *of-weak-ranking xs*
by (*simp-all add*: *xs-def weak-ranking-total-preorder*)

```
  let ?pref = λp x. measure-pmf.prob p {y. x ≼[le] y} and
      ?pref' = λp x. measure-pmf.prob p {y. x ≺[le] y}
  define f where f i = (SOME x. x ∈ xs ! i) for i
  have xs-wf: is-weak-ranking xs
    by (simp add: xs-def weak-ranking-total-preorder)
  hence f: f i ∈ xs ! i if i < length xs for i
    using that unfolding f-def is-weak-ranking-def
    by (intro someI-ex[of λx. x ∈ xs ! i]) (auto simp: set-conv-nth)
  have f': f i ∈ carrier if i < length xs for i
   using that f weak-ranking-Union unfolding xs-def by (auto simp: set-conv-nth)
  define n where n = length xs − 1
  from assms weak-ranking-Union have carrier-nonempty: carrier ≠ {} and xs ≠
[]
    by (auto simp: xs-def lotteries-on-def set-pmf-not-empty)
  hence n: length xs = Suc n and xs-nonempty: xs ≠ [] by (auto simp add: n-def)
  have SD': ?pref p (f i) ≤ ?pref q (f i) if i < length xs for i
    using f'[OF that] SD by (auto simp: SD-preorder preferred-alts-def)
  have f-le: le (f i) (f j) ⟷ i ≥ j if i < length xs j < length xs for i j
    using that weak-ranking-index-unique[OF xs-wf that(1) - f]
             weak-ranking-index-unique[OF xs-wf that(2) - f]
   by (auto simp add: le intro: f elim!: of-weak-ranking.cases intro!: of-weak-ranking.intros)

  have measure-pmf.expectation p u =
        (∑ i<n. (u (f i) − u (f (Suc i))) ∗ ?pref p (f i)) + u (f n)
    if p: p ∈ lotteries-on carrier for p
  proof −
    from p have measure-pmf.expectation p u =
                  (∑ i<length xs. u (f i) ∗ measure-pmf.prob p (xs ! i))
       by (simp add: f-def expected-utility-weak-ranking xs-def sum-list-sum-nth
atLeast0LessThan)
    also have ... = (∑ i<length xs. u (f i) ∗ (?pref p (f i) − ?pref' p (f i)))
    proof (intro sum.cong HOL.refl)
      fix i assume i: i ∈ {..<length xs}
      have ?pref p (f i) − ?pref' p (f i) =
            measure-pmf.prob p ({y. f i ≼[le] y} − {y. f i ≺[le] y})
        by (subst measure-pmf.finite-measure-Diff [symmetric])
           (auto simp: strongly-preferred-def)
      also have {y. f i ≼[le] y} − {y. f i ≺[le] y} =
                  {y. f i ≼[le] y ∧ y ≼[le] f i} (is - = ?A)
        by (auto simp: strongly-preferred-def)
      also have ... = xs ! i
      proof safe
        fix x assume le: le (f i) x le x (f i)
        from i f show x ∈ xs ! i
          by (intro weak-ranking-eqclass2[OF - - le]) (auto simp: xs-def)
      next
        fix x assume x ∈ 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) * measure-pmf.prob p (xs ! i) =
                  u (f i) * (?pref p (f i) − ?pref' p (f i)) by simp
  qed
  also have ... = (∑ i<length xs. u (f i) * ?pref p (f i)) −
                  (∑ i<length xs. u (f i) * ?pref' p (f i))
    by (simp add: sum-subtractf ring-distribs)
  also have (∑ i<length xs. u (f i) * ?pref p (f i)) =
            (∑ i<n. u (f i) * ?pref p (f i)) + u (f n) * ?pref p (f n) (is - = ?sum)
    by (simp add: n)
  also have (∑ i<length xs. u (f i) * ?pref' p (f i)) =
            (∑ i<n. u (f (Suc i)) * ?pref' p (f (Suc i))) + u (f 0) * ?pref' p (f
0)
    unfolding n sum.lessThan-Suc-shift by simp
  also have (∑ i<n. u (f (Suc i)) * ?pref' p (f (Suc i))) =
            (∑ i<n. u (f (Suc i)) * ?pref p (f i))
  proof (intro sum.cong HOL.refl)
    fix i assume i: i ∈ {..<n}
    have f (Suc i) ≺[le] y ⟷ f i ⪯[le] y for y
    proof (cases y ∈ carrier)
      assume y ∈ carrier
      with weak-ranking-Union obtain j where j: j < length xs y ∈ xs ! j
        by (auto simp: set-conv-nth xs-def)
      with weak-ranking-eqclass1[OF - f j(2)] weak-ranking-eqclass1[OF - j(2) f]
        have iff: le y z ⟷ le (f j) z le z y ⟷ le z (f j) for z
        by (auto intro: trans simp: xs-def)
      thus ?thesis using i j unfolding n-def
        by (auto simp: iff f-le strongly-preferred-def)
    qed (insert not-outside, auto simp: strongly-preferred-def)
    thus u (f (Suc i)) * ?pref' p (f (Suc i)) = u (f (Suc i)) * ?pref p (f i) by
simp
  qed
  also have ?sum − (... + u (f 0) * ?pref' p (f 0)) =
    (∑ i<n. (u (f i) − u (f (Suc i))) * ?pref p (f i)) −
     u (f 0) * ?pref' p (f 0) + u (f n) * ?pref p (f n)
      by (simp add: ring-distribs sum-subtractf)
  also have {y. f 0 ≺[le] y} = {}
    using hd-weak-ranking[of f 0] xs-nonempty f not-outside
    by (auto simp: hd-conv-nth xs-def strongly-preferred-def)
  also have {y. le (f n) y} = carrier
    using last-weak-ranking[of f n] xs-nonempty f not-outside
    by (auto simp: last-conv-nth xs-def n-def)
  also from p have measure-pmf.prob p carrier = 1
    by (subst measure-pmf.prob-eq-1)
       (auto simp: AE-measure-pmf-iff lotteries-on-def)
  finally show ?thesis by simp
qed
```

```
 from assms[THEN this] show measure-pmf.expectation p u ≤ measure-pmf.expectation
q u
    unfolding SD-preorder n-def using f'
    by (auto intro!: sum-mono mult-left-mono SD' simp: utility-le-iff f-le)

next
 assume ∀ u. is-vnm-utility u ⟶ measure-pmf.expectation p u ≤ measure-pmf.expectation
q u
 hence expected-utility-le: measure-pmf.expectation p u ≤ measure-pmf.expectation
q u
    if is-vnm-utility u for u using that by blast
  define xs where xs = weak-ranking le
  have le: le = of-weak-ranking xs and [simp]: is-weak-ranking xs
    by (simp-all add: xs-def weak-ranking-total-preorder)
  have carrier: carrier = ⋃(set xs)
    by (simp add: xs-def weak-ranking-Union)
  from assms have carrier-nonempty: carrier ≠ {}
    by (auto simp: lotteries-on-def set-pmf-not-empty)

  {
    fix x assume x: x ∈ carrier
    let ?idx = λy. length xs − weak-ranking-index y
    have preferred-subset-carrier: {y. le x y} ⊆ carrier
      using not-outside x by auto
    have measure-pmf.prob p {y. le x y} / real (length xs) ≤
          measure-pmf.prob q {y. le x y} / real (length xs)
    proof (rule field-le-epsilon)
      fix ε :: real assume ε: ε > 0
      define u where u y = indicator {y. y ≽[le] x} y + ε * ?idx y for y
      have is-utility: is-vnm-utility u unfolding u-def xs-def
     proof (intro vnm-utility.add-left vnm-utility.scaled utility-weak-ranking-index)
        fix y z assume le y z
        thus indicator {y. y ≽[le] x} y ≤ (indicator {y. y ≽[le] x} z :: real)
          by (auto intro: trans simp: indicator-def split: if-splits)
      qed fact+

      have (∑ y|le x y. pmf p y) ≤
            (∑ y|le x y. pmf p y) + ε * (∑ y∈carrier. ?idx y * pmf p y)
        using ε by (auto intro!: mult-nonneg-nonneg sum-nonneg pmf-nonneg)
      also from expected-utility-le is-utility have
        measure-pmf.expectation p u ≤ measure-pmf.expectation q u .
      with assms
        have (∑ a∈carrier. u a * pmf p a) ≤ (∑ a∈carrier. u a * pmf q a)
        by (subst (asm) (1 2) integral-measure-pmf[OF finite-carrier])
           (auto simp: lotteries-on-def set-pmf-eq ac-simps)
      hence (∑ y|le x y. pmf p y) + ε * (∑ y∈carrier. ?idx y * pmf p y) ≤
            (∑ y|le x y. pmf q y) + ε * (∑ y∈carrier. ?idx y * pmf q y)
        using x preferred-subset-carrier not-outside
        by (simp add: u-def sum.distrib finite-carrier algebra-simps sum-distrib-left
```

```
Int-absorb1 cong: rev-conj-cong)
      also have (∑ y∈carrier. ?idx y * pmf q y) ≤ (∑ y∈carrier. length xs * pmf
q y)
        by (intro sum-mono mult-right-mono) (simp-all add: pmf-nonneg)
      also have ... = measure-pmf.expectation q (λ-. length xs)
        using assms by (subst integral-measure-pmf[OF finite-carrier])
                       (auto simp: lotteries-on-def set-pmf-eq ac-simps)
      also have ... = length xs by simp
      also have (∑ y | le x y. pmf p y) = measure-pmf.prob p {y. le x y}
        using finite-subset[OF preferred-subset-carrier finite-carrier]
        by (simp add: measure-measure-pmf-finite)
      also have (∑ y | le x y. pmf q y) = measure-pmf.prob q {y. le x y}
        using finite-subset[OF preferred-subset-carrier finite-carrier]
        by (simp add: measure-measure-pmf-finite)
      finally show measure-pmf.prob p {y. le x y} / length xs ≤
                   measure-pmf.prob q {y. le x y} / length xs + ε
        using ε by (simp add: divide-simps)
    qed
    moreover from carrier-nonempty carrier have xs ≠ [] by auto
    ultimately have measure-pmf.prob p {y. le x y} ≤
                    measure-pmf.prob q {y. le x y}
      by (simp add: field-simps)
  }
  with assms show p ≼[SD(le)] q unfolding SD-preorder preferred-alts-def by
blast
qed

lemma not-strict-SD-iff:
  assumes p ∈ lotteries-on carrier q ∈ lotteries-on carrier
  shows   ¬(p ≺[SD(le)] q) ⟷
        (∃ u. is-vnm-utility u ∧ measure-pmf.expectation q u ≤ measure-pmf.expectation
p u)
proof
  let ?E = measure-pmf.expectation :: 'a pmf ⇒ - ⇒ real
  assume ∃ u. is-vnm-utility u ∧ ?E p u ≥ ?E q u
  then obtain u where u: is-vnm-utility u ?E p u ≥ ?E q u by blast
  interpret u: vnm-utility carrier le u by fact

  show ¬ p ≺[SD le] q
  proof
    assume less: p ≺[SD le] q
    with assms have pq: ?E p u ≤ ?E q u if 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': is-vnm-utility u' ?E p u' < ?E q u'
      by (auto simp: SD-iff-expected-utilities-le strongly-preferred-def not-le)
    interpret u': vnm-utility carrier le u' by fact

    have ∃ε>0. is-vnm-utility (λx. u x − ε * u' x)
```

```
    by (intro u.diff-epsilon antisym u'.utility-le)
  then obtain ε where ε: ε > 0 is-vnm-utility (λx. u x − ε * u' x) by auto
  define u'' where u'' x = u x − ε * u' x for x
  interpret u'': vnm-utility carrier le u'' unfolding u''-def by fact
  have exp-u'': ?E p u'' = ?E p u − ε * ?E p u' if p ∈ lotteries-on carrier for p
using that
    by (subst (1 2 3) integral-measure-pmf[of carrier])
     (auto simp: lotteries-on-def u''-def algebra-simps sum-subtractf sum-distrib-left)
  from assms ε have ?E p u'' > ?E q u''
    by (simp-all add: exp-u'' algebra-simps u-eq u')
  with 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 ≺[SD(le)] q) ⟷
              (∀ u. is-vnm-utility u ⟶ measure-pmf.expectation p u < mea-
sure-pmf.expectation q u)
 using not-strict-SD-iff[OF assms] by auto

end

end


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



context pref-profile-wf
begin

lemma SD-inefficient-support-subset:
 assumes inefficient: ¬SD-efficient R p'
 assumes support: set-pmf p' ⊆ set-pmf p
 assumes lotteries: p ∈ lotteries-on alts
 shows  ¬SD-efficient R p
proof −
 from assms have p'-wf: p' ∈ lotteries-on alts by (simp add: lotteries-on-def)
 from inefficient obtain q' i where q': q' ∈ lotteries-on alts i ∈ agents
   ⋀i. i ∈ agents ⟹ q' ≿[SD(R i)] p' q' ≻[SD(R i)] p'
   unfolding SD-efficient-def' by blast

 have subset: {x. pmf p' x > pmf q' x} ⊆ set-pmf p' by (auto simp: set-pmf-eq)
```

also have ... $\subseteq$ *set-pmf p* **by** *fact*
also have ... $\subseteq$ *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*})
define *supp* **where** *supp* = *set-pmf p* ∪ *set-pmf q'*
from *lotteries finite-alts q'*(*1*) **have** *finite-supp*: *finite supp*
by (*auto simp*: *lotteries-on-def supp-def dest*: *finite-subset*)
from *support* **have** [*simp*]: *pmf p x* = *0 pmf p' x* = *0 pmf q' x* = *0* **if** *x* ∉ *supp* **for** *x*
using *that* **by** (*auto simp*: *supp-def set-pmf-eq*)

from *finite support subset* **have** ε: ε > *0* **unfolding** ε*-def*
by (*auto simp*: *field-simps set-pmf-eq'*)
have *nonneg*: *pmf p x* + ε ∗ (*pmf q' x* − *pmf p' x*) ≥ *0* **for** *x*
proof (*cases pmf p' x* > *pmf q' x*)
case *True*
with *finite* **have** ε ≤ *pmf p x* / (*pmf p' x* − *pmf q' x*)
unfolding ε*-def* **by** (*intro Min-le*) *auto*
with *True* **show** *?thesis* **by** (*simp add*: *field-simps*)
next
case *False*
with *pmf-nonneg*[*of p x*] ε **show** *?thesis* **by** *simp*
qed

define *q* **where** *q* = *embed-pmf* (λ*x*. *pmf p x* + ε ∗ (*pmf q' x* − *pmf p' x*))
have (∫$^+$ *x*. *ennreal* (*pmf p x* + ε ∗ (*pmf q' x* − *pmf p' x*)) ∂*count-space UNIV*) = *1*
proof (*subst nn-integral-count-space'*)
have (∑ *x*∈*supp*. *ennreal* (*pmf p x* + ε ∗ (*pmf q' x* − *pmf p' x*))) =
ennreal ((∑ *x*∈*supp*. *pmf p x*) + ε ∗ ((∑ *x*∈*supp*. *pmf q' x*) − (∑ *x*∈*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-supp support* **have** ... = *1*
by (*subst* (*1 2 3*) *sum-pmf-eq-1*) (*auto simp*: *supp-def*)
finally show (∑ *x*∈*supp*. *ennreal* (*pmf p x* + ε ∗ (*pmf q' x* − *pmf p' x*))) = *1*
.
qed (*insert nonneg finite-supp*, *simp-all*)
with *nonneg* **have** *pmf-q*: *pmf q x* = *pmf p x* + ε ∗ (*pmf q' x* − *pmf p' x*) **for** *x*
unfolding *q-def* **by** (*intro pmf-embed-pmf*) *simp-all*
with *support* **have** *support-q*: *set-pmf q* ⊆ *supp*
unfolding *supp-def* **by** (*auto simp*: *set-pmf-eq*)
with *lotteries support q'*(*1*) **have** *q-wf*: *q* ∈ *lotteries-on alts*
by (*auto simp add*: *lotteries-on-def supp-def*)

from *support-q support* **have** *expected-utility*:

```
    measure-pmf.expectation q u = measure-pmf.expectation p u +
        ε * (measure-pmf.expectation q' u − 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 ≽[SD(R i)] p if i: i ∈ 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'-wf ε show ?thesis
      by (fastforce simp: SD-iff-expected-utilities-le expected-utility)
  qed
  moreover from ‹i ∈ agents› interpret finite-total-preorder-on alts R i by simp
    from lotteries q'(1,4) q-wf p'-wf ε have q ≻[SD(R i)] p
   by (force simp: SD-iff-expected-utilities-le expected-utility not-le strongly-preferred-def)
  ultimately show ?thesis using q-wf ‹i ∈ agents› unfolding SD-efficient-def'
by blast
qed

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

lemma SD-efficient-same-support:
  assumes set-pmf p = set-pmf p' p ∈ lotteries-on alts
  shows   SD-efficient R p ⟷ 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)

lemma SD-efficient-iff:
  assumes p ∈ lotteries-on alts
  shows   SD-efficient R p ⟷ SD-efficient R (pmf-of-set (set-pmf p))
  using assms finite-alts
  by (intro SD-efficient-same-support)
     (simp, subst set-pmf-of-set,
        auto simp: set-pmf-not-empty lotteries-on-def intro: finite-subset[OF - fi-
nite-alts])

lemma SD-efficient-no-pareto-loser:
  assumes efficient: SD-efficient R p and p-wf: p ∈ lotteries-on alts
  shows   set-pmf p ∩ pareto-losers R = {}
proof −
  have x ∉ pareto-losers R if x: x ∈ set-pmf p for x
  proof −
    from x have set-pmf (return-pmf x) ⊆ 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 ∈ alts by (auto simp: lotteries-on-def)
    ultimately show x ∉ 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 ∈ lotteries-on alts q ∈ lotteries-on alts q ≻[Pareto(SD ∘ R)] p
        set-pmf p = set-pmf q
  obtains r where r ∈ lotteries-on alts r ≽[Pareto(SD ∘ R)] q set-pmf r ⊂ set-pmf
p
proof −
  have subset: {x. pmf p x > pmf q x} ⊆ set-pmf p by (auto simp: set-pmf-eq)
  also have ... ⊆ alts using assms by (simp add: lotteries-on-def)
  finally have finite: finite {x. pmf p x > pmf q x}
    using finite-alts by (rule finite-subset)

  from assms have q ≠ p by (auto simp: strongly-preferred-def)
  hence ex-less: ∃ x. pmf p x > pmf q x by (rule pmf-neq-exists-less)

  define ε where ε = Min {pmf p x / (pmf p x − pmf q x) |x. pmf p x > pmf q
x}
  define supp where supp = set-pmf p
  from assms finite-alts have finite-supp: finite supp
    by (auto simp: lotteries-on-def supp-def dest: finite-subset)
  from assms have [simp]: pmf p x = 0 pmf q x = 0 if x ∉ supp for x
    using that by (auto simp: supp-def set-pmf-eq)

  from finite subset ex-less have ε: ε ≥ 1 unfolding ε-def
    by (intro Min.boundedI) (auto simp: field-simps pmf-nonneg)
  have nonneg: pmf p x + ε ∗ (pmf q x − pmf p x) ≥ 0 for x
  proof (cases pmf p x > pmf q x)
    case True
    with finite have ε ≤ pmf p x / (pmf p x − pmf q x)
      unfolding ε-def by (intro Min-le) auto
    with True show ?thesis by (simp add: field-simps)
  next
    case False
    with pmf-nonneg[of p x] ε show ?thesis by simp
  qed

  define r where r = embed-pmf (λx. pmf p x + ε ∗ (pmf q x − pmf p x))
  have (∫⁺ x. ennreal (pmf p x + ε ∗ (pmf q x − pmf p x)) ∂count-space UNIV)
= 1
```

proof (*subst nn-integral-count-space′*)
have ($\sum x{\in}supp.\ ennreal\ (pmf\ p\ x + \varepsilon * (pmf\ q\ x - pmf\ p\ x))) =$
ennreal $((\sum x{\in}supp.\ pmf\ p\ x) + \varepsilon * ((\sum x{\in}supp.\ pmf\ q\ x) - (\sum x{\in}supp.$
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-supp* **have** $\ldots = 1$
by (*subst* (*1 2 3*) *sum-pmf-eq-1*) (*auto simp*: *supp-def assms*)
finally show $(\sum x{\in}supp.\ ennreal\ (pmf\ p\ x + \varepsilon * (pmf\ q\ x - pmf\ p\ x))) = 1$ **.**
qed (*insert nonneg finite-supp*, *simp-all*)
with *nonneg* **have** *pmf-r*: *pmf r x* = *pmf p x* + ε * (*pmf q x* − *pmf p x*) **for** *x*
unfolding *r-def* **by** (*intro pmf-embed-pmf*) *simp-all*

with *assms* **have** *set-pmf r* ⊆ *supp*
unfolding *supp-def* **by** (*auto simp*: *set-pmf-eq*)
from *finite ex-less* **have** $\varepsilon \in \{pmf\ p\ x\ /\ (pmf\ p\ x - pmf\ q\ x)\ |x.\ pmf\ p\ x > pmf\ q\ x\}$
unfolding ε*-def* **by** (*intro Min-in*) *auto*
then obtain *x* **where** $\varepsilon = pmf\ p\ x\ /\ (pmf\ p\ x - pmf\ q\ x)\ pmf\ p\ x > pmf\ q\ x$
by *blast*
hence *pmf r x* = *0* **by** (*simp add*: *pmf-r field-simps*)
moreover from ‹*pmf p x* > *pmf q x*› *pmf-nonneg*[*of q x*]
have *pmf p x* > *0* **by** *linarith*
ultimately have *x* ∈ *set-pmf p* − *set-pmf r* **by** (*auto simp*: *set-pmf-iff*)
with ‹*set-pmf r* ⊆ *supp*› **have** *support-r*: *set-pmf r* ⊂ *set-pmf p* **unfolding** *supp-def* **by** *blast*
from *this assms* **have** *r-wf*: *r* ∈ *lotteries-on alts* **by** (*simp add*: *lotteries-on-def*)

have *r* ≿[*Pareto*(*SD*∘*R*)] *q* **unfolding** *SD.Pareto-iff* **unfolding** *o-def*
proof
fix *i* **assume** *i*: *i* ∈ *agents*
then interpret *finite-total-preorder-on alts R i* **by** *simp*
show *r* ≿[*SD*(*R i*)] *q*
proof (*subst SD-iff-expected-utilities-le*; *safe?*)
fix *u* **assume** *u*: *is-vnm-utility u*
from *support-r* **have** *expected-utility-r*:
measure-pmf.expectation r u = *measure-pmf.expectation p u* +
ε * (*measure-pmf.expectation q u* − *measure-pmf.expectation p u*)
by (*subst* (*1 2 3 4*) *integral-measure-pmf*[*OF finite-supp*])
(*auto simp*: *supp-def assms pmf-r sum.distrib sum-distrib-left*
sum-distrib-right sum-subtractf algebra-simps)
from *assms i* **have** *q* ≿[*SD*(*R i*)] *p* **by** (*simp add*: *SD.Pareto-strict-iff*)
with *assms u* **have** *measure-pmf.expectation q u* ≥ *measure-pmf.expectation p u*
by (*simp add*: *SD-iff-expected-utilities-le r-wf*)
hence (ε − *1*) * *measure-pmf.expectation p u* ≤ (ε − *1*) * *measure-pmf.expectation q u*
using ε **by** (*intro mult-left-mono*) *simp-all*
thus *measure-pmf.expectation q u* ≤ *measure-pmf.expectation r u*

by (*simp add*: *algebra-simps expected-utility-r*)
qed *fact+*
qed
from *that*[*OF r-wf this support-r*] **show** *?thesis* **.**
qed

4.5 Existence of SD-efficient lotteries

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

context
fixes p :: *'alt lottery*
assumes *lott*: $p \in$ *lotteries-on alts*
begin

private definition *improve-lottery* :: *'alt lottery* $\Rightarrow$ *'alt lottery* **where**
improve-lottery q = (*let* $A = \{r \in$ *lotteries-on alts*. $r \succ$[*Pareto*(*SD*$\circ R$)] $q\}$ *in*
(*SOME r*. $r \in A \wedge \neg(\exists\, r' \in A$. *set-pmf* $r' \subset$ *set-pmf* r)))

private lemma *improve-lottery*:
assumes $\neg$*SD-efficient* R q
defines $r \equiv$ *improve-lottery* q
shows $r \in$ *lotteries-on alts* $r \succ$[*Pareto*(*SD*$\circ R$)] q
$\bigwedge r'$. $r' \in$ *lotteries-on alts* $\Longrightarrow r' \succ$[*Pareto*(*SD*$\circ R$)] $q \Longrightarrow \neg$(*set-pmf* $r' \subset$ *set-pmf* r)
proof −
define A **where** $A = \{r \in$ *lotteries-on alts*. $r \succ$[*Pareto*(*SD*$\circ R$)] $q\}$
have *subset-alts*: $X \subseteq$ *alts* **if** $X \in$ *set-pmf*‘A **for** X **using** *that*
by (*auto simp*: *A-def lotteries-on-def*)
have *r-altdef*: r = (*SOME r*. $r \in A \wedge \neg(\exists\, r' \in A$. *set-pmf* $r' \subset$ *set-pmf* r))
unfolding *r-def improve-lottery-def Let-def A-def* **by** *simp*
from *assms* **have** *nonempty*: $A \neq \{\}$ **by** (*auto simp*: *A-def SD-efficient-def*)
hence *nonempty'*: *set-pmf*‘$A \neq \{\}$ **by** *simp*
have *set-pmf* ‘ $A \subseteq$ *Pow alts* **by** (*auto simp*: *A-def lotteries-on-def*)

from *finite-alts* **have** *wf*: *wf* $\{(X, Y).\ X \subset Y \wedge Y \subseteq$ *alts*$\}$
by (*rule finite-subset-wf*)
obtain X
where $X \in$ *set-pmf*‘A $\bigwedge Y$. $Y \subset X \wedge X \subseteq$ *alts* $\Longrightarrow Y \notin$ *set-pmf* ‘ A
by (*rule wfE-min'*[*OF wf nonempty'*]) *simp-all*
hence $\exists\, r$. $r \in A \wedge \neg(\exists\, r' \in A$. *set-pmf* $r' \subset$ *set-pmf* r)
by (*auto simp*: *subset-alts*[*of X*])
from *someI-ex*[*OF this*, *folded r-altdef*]
show $r \in$ *lotteries-on alts* $r \succ$[*Pareto*(*SD*$\circ R$)] q
$\bigwedge r'$. $r' \in$ *lotteries-on alts* $\Longrightarrow r' \succ$[*Pareto*(*SD*$\circ R$)] $q \Longrightarrow \neg$(*set-pmf* $r' \subset$ *set-pmf* r)
unfolding *A-def* **by** *blast+*
qed

private fun *sd-chain* :: *nat* ⇒ *'alt lottery option* **where**
  sd-chain 0 = *Some p*
| *sd-chain* (*Suc n*) =
    (*case sd-chain n of*
      None ⇒ *None*
    | *Some p* ⇒ *if SD-efficient R p then None else Some* (*improve-lottery p*))

private lemma *sd-chain-None-propagate*:
  m ≥ *n* ⟹ *sd-chain n* = *None* ⟹ *sd-chain m* = *None*
  by (*induction rule*: *inc-induct*) *simp-all*

private lemma *sd-chain-Some-propagate*:
  m ≥ *n* ⟹ *sd-chain m* = *Some q* ⟹ ∃ *q'*. *sd-chain n* = *Some q'*
  by (*cases sd-chain n*) (*auto simp*: *sd-chain-None-propagate*)

private lemma *sd-chain-NoneD*:
  sd-chain n = *None* ⟹ ∃ *n p*. *sd-chain n* = *Some p* ∧ *SD-efficient R p*
  by (*induction n*) (*auto split*: *option.splits if-splits*)

private lemma *sd-chain-lottery*: *sd-chain n* = *Some q* ⟹ *q* ∈ *lotteries-on alts*
  by (*induction n*) (*insert lott*, *auto split*: *option.splits if-splits simp*: *improve-lottery*)

private lemma *sd-chain-Suc*:
  assumes *sd-chain m* = *Some q*
  assumes *sd-chain* (*Suc m*) = *Some r*
  shows *q* ≺[*Pareto*(*SD*∘*R*)] *r*
  using *assms* **by** (*auto split*: *if-splits simp*: *improve-lottery*)

private lemma *sd-chain-strictly-preferred*:
  assumes *m* < *n*
  assumes *sd-chain m* = *Some q*
  assumes *sd-chain n* = *Some s*
  shows *q* ≺[*Pareto*(*SD*∘*R*)] *s*
  using *assms*
proof (*induction arbitrary*: *q rule*: *strict-inc-induct*)
  case (*base k q*)
  with *sd-chain-Suc*[*of k q s*] **show** *?case* **by** (*simp del*: *sd-chain.simps add*: *o-def*)
next
  case (*step k q*)
  from *step.hyps* **have** *Suc k* ≤ *n* **by** *simp*
  from *sd-chain-Some-propagate*[*OF this*, *of s*] *step.prems* **obtain** *r*
    where *r*: *sd-chain* (*Suc k*) = *Some r* **by** (*auto simp del*: *sd-chain.simps*)
  with *step.prems* **have** *q* ≺[*Pareto* (*SD* ∘ *R*)] *r* **by** (*intro sd-chain-Suc*)
  moreover from *r step.prems* **have** *r* ≺[*Pareto* (*SD* ∘ *R*)] *s* **by** (*intro step.IH*)
simp-all
  ultimately show *?case* **by** (*rule SD.Pareto.strict-trans*)
qed

```
private lemma sd-chain-preferred:
  assumes m ≤ n
  assumes sd-chain m = Some q
  assumes sd-chain n = Some s
  shows   q ≼[Pareto(SD∘R)] s
proof (cases m < n)
  case True
  from sd-chain-strictly-preferred[OF this assms(2,3)] show ?thesis
    by (simp add: strongly-preferred-def)
next
  case False
  with assms show ?thesis by (auto intro: SD.Pareto.refl sd-chain-lottery)
qed

lemma SD-efficient-lottery-exists:
  obtains q where q ∈ lotteries-on alts q ≽[Pareto(SD∘R)] p SD-efficient R q
proof −
  consider ∃ n. sd-chain n = None | ∀ n. ∃ q. sd-chain n = Some q
    using option.exhaust by metis
  thus ?thesis
  proof cases
    case 1
    define m where m = (LEAST m. sd-chain m = None)
    define k where k = m − 1
    from LeastI-ex[OF 1] have m: sd-chain m = None by (simp add: m-def)
    from m have nz: m ≠ 0 by (intro notI) simp-all
    from nz have m-altdef: m = Suc k by (simp add: k-def)
    from nz Least-le[of λm. sd-chain m = None m − 1, folded m-def]
      obtain q where q: sd-chain k = Some q by (cases sd-chain (m − 1)) (auto
simp: k-def)
   from sd-chain-preferred[OF - sd-chain.simps(1) this] have q ≽[Pareto(SD∘R)]
p by simp
    moreover from q have q ∈ lotteries-on alts by (simp add: sd-chain-lottery)
     moreover from q m have SD-efficient R q by (auto split: if-splits simp:
m-altdef)
    ultimately show ?thesis using that[of q] by blast
  next
    case 2
    have range (set-pmf ∘ the ∘ sd-chain) ⊆ Pow alts unfolding o-def
    proof safe
      fix n x assume A: x ∈ set-pmf (the (sd-chain n))
      from 2 obtain q where sd-chain n = Some q by auto
      with sd-chain-lottery[of n q] have set-pmf (the (sd-chain n)) ⊆ alts
        by (simp add: lotteries-on-def)
      with A show x ∈ alts by blast
    qed
    hence finite (range (set-pmf ∘ the ∘ sd-chain)) by (rule finite-subset) simp-all
    from pigeonhole-infinite[OF infinite-UNIV-nat this]
       obtain m where infinite {n. set-pmf (the (sd-chain n)) = set-pmf (the
```

(*sd-chain m*))}
by *auto*
hence *infinite* ({*n*. *set-pmf* (*the* (*sd-chain n*)) = *set-pmf* (*the* (*sd-chain m*))}
− {*k*. ¬(*k* > *m*)})
by (*simp add*: *not-less*)
hence ({*n*. *set-pmf* (*the* (*sd-chain n*)) = *set-pmf* (*the* (*sd-chain m*))} − {*k*.
¬(*k* > *m*)}) ≠ {}
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* ≺[*Pareto*(*SD*∘*R*)] *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*: ¬*SD-efficient R r q* = *improve-lottery r*
by (*auto simp del*: *sd-chain.simps split*: *if-splits option.splits*)

from *pq* **have** *p* ∈ *lotteries-on alts q* ∈ *lotteries-on alts*
by (*simp-all add*: *sd-chain-lottery*)
from *improve-lottery-support-subset*[*OF this less supp-eq*]
obtain *s* **where** *s*: *s* ∈ *lotteries-on alts Pareto* (*SD* ∘ *R*) *q s set-pmf s* ⊂ *set-pmf*
p .
from *improve-lottery*(*2*)[*of r*] *r s* **have** *s* ≻[*Pareto*(*SD*∘*R*)] *r*
by (*auto intro*: *SD.Pareto.strict-weak-trans*)
from *improve-lottery*(*3*)[*OF r*(*1*) *s*(*1*) *this*] *supp-eq r*
have ¬*set-pmf s* ⊂ *set-pmf p* **by** *simp*
with *s*(*3*) **show** *?thesis* **by** *contradiction*
qed
qed

end

lemma
assumes *p* ∈ *lotteries-on alts*
shows ∃ *q*∈*lotteries-on alts*. *q* ≿[*Pareto*(*SD*∘*R*)] *p* ∧ *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.Probability
Preference-Profiles
Elections
Order-Predicates
Stochastic-Dominance
SD-Efficiency
begin

5.1 Basic Social Choice definitions

context *election*
begin

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

abbreviation *lotteries* **where**
lotteries ≡ *lotteries-on alts*

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

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

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

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

end

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

5.2 Social Decision Schemes

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

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

5.3 Anonymity

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

locale *anonymous-sds* = *social-decision-scheme agents alts sds*
  for *agents* :: *'agent set* **and** *alts* :: *'alt set* **and** *sds* +
  assumes *anonymous*: π *permutes agents* $\Longrightarrow$ *is-pref-profile R* $\Longrightarrow$ *sds* $(R \circ \pi)$ = *sds R*
begin

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 −
  from *assms* **obtain** π **where** π *permutes agents prefs-from-table xs* $\circ$ π = *prefs-from-table ys*
    by (*rule prefs-from-table-agent-permutation*)
  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* $\Longrightarrow$ *is-pref-profile R* $\Longrightarrow$
    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 $a \in alts$
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-wf*
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*
by (*intro pmf-map-inj'*) (*simp-all add*: *permutes-inj*)
finally show *?thesis* **.**
qed

end

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

lemma *sds-anonymous-neutral*:
assumes *perm*: σ *permutes alts* **and** *wf*: *is-pref-profile R1 is-pref-profile R2*
assumes *eq*: *anonymous-profile R1* =
image-mset (*map* ((') σ)) (*anonymous-profile R2*)
shows *sds R1* = *map-pmf* σ (*sds R2*)
proof −
interpret *R1*: *pref-profile-wf agents alts R1* **by** *fact*
interpret *R2*: *pref-profile-wf agents alts R2* **by** *fact*
from *perm* **have** *wf'*: *is-pref-profile* (*permute-profile* σ *R2*)
by (*rule R2.wf-permute-alts*)
from *eq perm* **have** *anonymous-profile R1* = *anonymous-profile* (*permute-profile* σ *R2*)
by (*simp add*: *R2.anonymous-profile-permute*)
from *anonymous-profile-agent-permutation*[*OF this wf*(*1*) *wf'*]
obtain π **where** π *permutes agents permute-profile* σ *R2* ∘ π = *R1* **by** *auto*
have *sds* (*permute-profile* σ *R2* ∘ 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* ∘ 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 ((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)
            distinct (map fst xs)
      and x: x ∈ alts y = permutation-of-list xs x
  shows   pmf (sds R) x = pmf (sds R) y
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) = anony-
mous-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 $\Longrightarrow$ *set-pmf* (*sds R*) $\cap$ *pareto-losers R* = {}
begin

lemma *ex-post-efficient'*:
  assumes *is-pref-profile R y* $\succ$[*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* $\in$ *agents* $\forall$ *i*$\in$*agents*. *y* $\succeq$[*R i*] *x* $\neg$*y* $\preceq$[*R i*] *x*
  shows *pmf* (*sds R*) *x* = *0*
proof −
  from *assms*(*1*) **interpret** *pref-profile-wf agents alts R* **.**
  from *assms*(*2*−) **show** *?thesis*
    by (*intro ex-post-efficient'*[*OF assms*(*1*), *of - y*])
      (*auto simp*: *Pareto-iff strongly-preferred-def*)
qed

end

5.6 SD efficiency

An SDS is SD-efficient if it returns an SD-efficient lottery for every preference profile, i.e. if the SDS outputs a lottery, it is never the case that there is another lottery that is weakly preferred by all agents an 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* $\Longrightarrow$ *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* $\in$ *lotteries*
  assumes $\bigwedge$*i*. *i* $\in$ *agents* $\Longrightarrow$ *q* $\succeq$[*SD*(*R i*)] *sds R i* $\in$ *agents q* $\succ$[*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*
assume ¬(∃ *x*∈*A*. *pmf* (*sds R*) *x* = *0*)
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*
¬*p* ≾[*SD*(*R i*)] *pmf-of-set A*
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*:
  is-pref-profile R ⟹ *i* ∈ *agents* ⟹ *total-preorder-on alts Ri'* ⟹
    ¬*manipulable-profile R i Ri'*

5.8 Strong strategyproofness

context *social-decision-scheme*
begin

The SDS is said to be strongly strategyproof for a particular preference profile, a particular agent, and a particular alternative preference ordering for that agent if the lottery obtained if the agent submits the alternative preferences is SD-dominated by the one obtained if the original preferences are submitted. (SD-dominated w.r.t. the original preferences)

In other words: the SDS is strategyproof w.r.t the preference profile R and the agent i and the alternative preference relation 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*) *pref-profile* ⇒ '*agent* ⇒ '*alt relation* ⇒ *bool* **where**
 strongly-strategyproof-profile R i Ri' ⟷ *sds R* ≽[*SD* (*R i*)] *sds* (*R*(*i* := *Ri'*))

lemma *strongly-strategyproof-profileI* [*intro*]:
 assumes *is-pref-profile R total-preorder-on alts Ri' i* ∈ *agents*
 assumes ⋀*x*. *x* ∈ *alts* ⟹ *lottery-prob* (*sds* (*R*(*i* := *Ri'*))) (*preferred-alts* (*R i*) *x*)
                ≤ *lottery-prob* (*sds R*) (*preferred-alts* (*R i*) *x*)
 shows *strongly-strategyproof-profile R i Ri'*
proof −
 interpret *pref-profile-wf agents alts R* **by** *fact*
 show *?thesis*

unfolding *strongly-strategyproof-profile-def*
by *rule* (*auto intro*!: *sds-wf assms pref-profile-wf.wf-update*)
qed

lemma *strongly-strategyproof-imp-not-manipulable*:
assumes *strongly-strategyproof-profile R i Ri'*
shows ¬*manipulable-profile R i Ri'*
using *assms* **unfolding** *strongly-strategyproof-profile-def manipulable-profile-def*
by (*auto simp*: *strongly-preferred-def*)

end

An SDS is strongly strategyproof if it is strongly strategyproof for all combinations of preference profiles, agents, and alternative preference relations.

locale *strongly-strategyproof-sds* = *social-decision-scheme agents alts sds*
for *agents* :: *'agent set* **and** *alts* :: *'alt set* **and** *sds* +
assumes *strongly-strategyproof*:
is-pref-profile R ⟹ *i* ∈ *agents* ⟹ *total-preorder-on alts Ri'* ⟹
strongly-strategyproof-profile R i Ri'
begin

Any SDS that is strongly strategyproof is also weakly strategyproof.

sublocale *strategyproof-sds*
by *unfold-locales*
(*simp add*: *strongly-strategyproof-imp-not-manipulable strongly-strategyproof*)

end

locale *strategyproof-an-sds* =
strategyproof-sds agents alts sds + *an-sds agents alts sds*
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 = (λ*i 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* $\subseteq$ *agents'* *alts* $\subseteq$ *alts'* *finite alts'*
defines *R'* $\equiv$ *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* $\in$ *agents'* **for** *i*
proof (*cases i* $\in$ *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' i* = (λ- -. *False*) **if** *i* $\notin$ *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 *π permutes agents agents* $\subseteq$ *agents'*
shows *lift-pref-profile agents alts agents' alts' (R* ∘ *π)* =
lift-pref-profile agents alts agents' alts' R ∘ *π*
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 *σ permutes alts alts* $\subseteq$ *alts'*
shows *lift-pref-profile agents alts agents' alts' (permute-profile σ R)* =
permute-profile σ (*lift-pref-profile agents alts agents' alts' R*)
proof −
from *assms* **have** *inv*: *inv σ permutes alts* **by** (*intro permutes-inv*)
from *this assms*(*2*) **have** *inv σ permutes alts'* **by** (*rule permutes-subset*)
with *inv* **show** *?thesis* **using** *assms permutes-inj*[*OF* ‹*inv σ permutes alts*›]
by (*fastforce simp add*: *lift-pref-profile-def permutes-in-image*
permute-profile-def fun-eq-iff dest: *injD*)
qed

lemma *lotteries-on-subset*: *A* $\subseteq$ *B* $\Longrightarrow$ *p* $\in$ *lotteries-on A* $\Longrightarrow$ *p* $\in$ *lotteries-on B*
unfolding *lotteries-on-def* **by** *blast*

lemma *lottery-prob-carrier*: *p* $\in$ *lotteries-on A* $\Longrightarrow$ *measure-pmf.prob p A* = *1*
by (*auto simp*: *measure-pmf.prob-eq-1 lotteries-on-def AE-measure-pmf-iff*)

context
fixes *agents alts R agents′ alts′ R′*
assumes *R-wf*: *pref-profile-wf agents alts R*
assumes *election*: *agents* ⊆ *agents′ alts* ⊆ *alts′ alts* ≠ {} *agents* ≠ {} *finite alts′*
defines *R′* ≡ *lift-pref-profile agents alts agents′ alts′ R*
begin

interpretation *R*: *pref-profile-wf agents alts R* **by** *fact*
interpretation *R′*: *pref-profile-wf agents′ alts′ R′*
using *election R-wf* **by** (*simp add*: *R′-def lift-pref-profile-wf*)

lemma *lift-pref-profile-strict-iff*:
x ≺[*lift-pref-profile agents alts agents′ alts′ R i*] *y* ⟷
i ∈ *agents* ∧ ((*y* ∈ *alts* ∧ *x* ∈ *alts′* − *alts*) ∨ *x* ≺[*R i*] *y*)
proof (*cases i* ∈ *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* ∈ *agents′* **and** *x*: *x* ∈ *alts′*
shows *preferred-alts* (*R′ i*) *x* =
(*if i* ∈ *agents* ∧ *x* ∈ *alts then preferred-alts* (*R i*) *x else alts′*)
proof (*cases i* ∈ *agents*)
assume *i*: *i* ∈ *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 ⪯[*Pareto*(*R′*)] *y* ⟷ *x* ∈ *alts′* ∧ *y* ∈ *alts′* ∧ (*x* ∉ *alts* ∨ *x* ⪯[*Pareto*(*R*)] *y*)
proof −
from *R.nonempty-agents* **obtain** *i* **where** *i*: *i* ∈ *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 ≺[*Pareto*(*R′*)] *y* ⟷ *x* ∈ *alts′* ∧ *y* ∈ *alts′* ∧ (*x* ∉ *alts* ∧ *y* ∈ *alts* ∨ *x* ≺[*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 *pareto-losers* $R' =$ *pareto-losers* $R \cup (alts' - alts)$
proof −
  have *A*: $x \in alts$ $y \in alts$ **if** $x \prec[Pareto(R)]\ y$ **for** $x\ y$
    using *that R.Pareto.not-outside* **unfolding** *strongly-preferred-def* **by** *auto*
  have *B*: $x \in alts'$ **if** $x \in alts$ **for** x **using** *election that* **by** *blast*
  from *R.nonempty-alts* **obtain** x **where** *x*: $x \in alts$ **by** *blast*
 thus *?thesis* **unfolding** *pareto-losers-def lift-pref-profile-Pareto-strict-iff* [*abs-def*]
    by (*auto dest*: *A B*)
qed

context
begin
private lemma *lift-SD-iff-agent*:
  assumes $p \in$ *lotteries-on alts* $q \in$ *lotteries-on alts* **and** *i*: $i \in agents$
  shows $p \preceq[SD(R'\ i)]\ q \longleftrightarrow p \preceq[SD(R\ i)]\ q$
proof −
  from *i* **interpret** *Ri*: *preorder-on alts R i* **by** *simp*
  from *i election* **have** *i'*: $i \in agents'$ **by** *blast*
  then interpret *R'i*: *preorder-on alts' R' i* **by** *simp*
  from *assms election* **have** $p \in$ *lotteries-on alts'* $q \in$ *lotteries-on alts'*
    by (*auto intro*: *lotteries-on-subset*)
  with *assms election i'* **show** *?thesis*
    by (*auto simp*: *Ri.SD-preorder R'i.SD-preorder*
        preferred-alts-lift-pref-profile lottery-prob-carrier)
qed

private lemma *lift-SD-iff-nonagent*:
  assumes $p \in$ *lotteries-on alts* $q \in$ *lotteries-on alts* **and** *i*: $i \in agents' - agents$
  shows $p \preceq[SD(R'\ i)]\ q$
proof −
  from *i election* **have** *i'*: $i \in agents'$ **by** *blast*
  then interpret *R'i*: *preorder-on alts' R' i* **by** *simp*
  from *assms election* **have** $p \in$ *lotteries-on alts'* $q \in$ *lotteries-on alts'*
    by (*auto intro*: *lotteries-on-subset*)
  with *assms election i'* **show** *?thesis*
   by (*auto simp*: *R'i.SD-preorder preferred-alts-lift-pref-profile lottery-prob-carrier*)
qed

lemmas *lift-SD-iff* = *lift-SD-iff-agent lift-SD-iff-nonagent*

lemma *lift-SD-iff'*:
  $p \in$ *lotteries-on alts* $\Longrightarrow$ $q \in$ *lotteries-on alts* $\Longrightarrow$ $i \in agents' \Longrightarrow$
    $p \preceq[SD(R'\ i)]\ q \longleftrightarrow i \notin agents \vee p \preceq[SD(R\ i)]\ q$
  by (*cases* $i \in agents$) (*simp-all add*: *lift-SD-iff*)

end

lemma *lift-SD-strict-iff*:

assumes $p \in$ *lotteries-on alts* $q \in$ *lotteries-on alts* **and** i: $i \in$ *agents*
shows $p \prec[SD(R'\ i)]\ q \longleftrightarrow p \prec[SD(R\ i)]\ q$
using *assms* **by** (*simp add*: *strongly-preferred-def lift-SD-iff*)

lemma *lift-Pareto-SD-iff*:
assumes $p \in$ *lotteries-on alts* $q \in$ *lotteries-on alts*
shows $p \preceq[Pareto(SD \circ R')]\ q \longleftrightarrow p \preceq[Pareto(SD \circ R)]\ q$
using *assms election* **by** (*auto simp*: *R.SD.Pareto-iff R'.SD.Pareto-iff lift-SD-iff'*)

lemma *lift-Pareto-SD-strict-iff*:
assumes $p \in$ *lotteries-on alts* $q \in$ *lotteries-on alts*
shows $p \prec[Pareto(SD \circ R')]\ q \longleftrightarrow p \prec[Pareto(SD \circ R)]\ q$
using *assms* **by** (*simp add*: *strongly-preferred-def lift-Pareto-SD-iff*)

lemma *lift-SD-efficient-iff*:
assumes p: $p \in$ *lotteries-on alts*
shows *SD-efficient* $R'\ p \longleftrightarrow$ *SD-efficient* $R\ p$
proof
assume *eff*: *SD-efficient* $R'\ p$
have $\neg(q \succ[Pareto(SD \circ R)]\ p)$ **if** q: $q \in$ *lotteries-on alts* **for** q
proof −
from q *election* **have** q': $q \in$ *lotteries-on alts'* **by** (*blast intro*: *lotteries-on-subset*)
with *eff* **have** $\neg(q \succ[Pareto(SD \circ R')]\ p)$ **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 $\neg(q \succ[Pareto(SD \circ R')]\ p)$ **if** q: $q \in$ *lotteries-on alts'* **for** q
proof
assume *less*: $q \succ[Pareto(SD \circ R')]\ p$
from *R'.SD-efficient-lottery-exists*[*OF* q]
obtain q' **where** q': $q' \in$ *lotteries-on alts'* *Pareto* $(SD \circ R')\ q\ q'$ *SD-efficient* $R'\ q'$ **.**
have $x \notin$ *set-pmf* q' **if** x: $x \in alts' - alts$ **for** x
proof −
from x **have** $x \in$ *pareto-losers* R' **by** (*simp add*: *pareto-losers-lift-pref-profile*)
with *R'.SD-efficient-no-pareto-loser*[*OF* $q'(3,1)$] **show** $x \notin$ *set-pmf* q' **by** *blast*
qed
with q' **have** $q' \in$ *lotteries-on alts* **by** (*auto simp*: *lotteries-on-def*)
moreover from q' *less* **have** $q' \succ[Pareto(SD \circ R')]\ p$
by (*auto intro*: *R'.SD.Pareto.strict-weak-trans*)
with ‹$q' \in$ *lotteries-on alts*› p **have** $q' \succ[Pareto(SD \circ R)]\ p$
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'* ≠ {}
begin

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' 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 *agents'-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*
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* ∘ *π*) = *lift R* ∘ *π*
using *agents′-subset* **by** (*rule lift-pref-profile-permute-agents*)
hence *sds* (*lift* (*R* ∘ *π*)) = *sds* (*lift R* ∘ *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 ∘ π) = sds (lift R)
    by (simp add: lowered-def)
  finally show lowered (R ∘ 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

end

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 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-wf* **interpret** *liftR*: *pref-profile-wf agents alts lift R* **by** *simp*

define *lift-Ri′*
where *lift-Ri′ x y* ⟷ *x* ∈ *alts* ∧ *y* ∈ *alts* ∧ (*x* = *y* ∨ *x* ∉ *alts′* ∨ (*y* ∈ *alts′* ∧ *Ri′ x y*))
for *x y*
define *S* **where** *S* = (*lift R*)(*i* := *lift-Ri′*)
from *Ri′* **interpret** *Ri′*: *total-preorder-on alts′ Ri′* **.**
have *wf-lift-Ri′*: *total-preorder-on alts lift-Ri′* **using** *Ri′.total*
by *unfold-locales* (*auto simp*: *lift-Ri′-def intro*: *Ri′.trans*)
from *agents′-subset i* **have** *S-altdef*: *S* = *lift* (*R*(*i* := *Ri′*))
by (*auto simp*: *fun-eq-iff lift-pref-profile-def lift-Ri′-def S-def*)
have *lowered* (*R*(*i* := *Ri′*)) ∈ *lowered.lotteries*
by (*intro lowered.sds-wf R.wf-update i Ri′*)
hence *sds-S-wf*: *sds S* ∈ *lowered.lotteries* **by** (*simp add*: *S-altdef lowered-def*)

from *manipulable* **have** *lowered R* ≺[*SD* (*R i*)] *sds* (*lift* (*R*(*i* := *Ri′*)))
unfolding *lowered.manipulable-profile-def* **by** (*simp add*: *lowered-def*)
also note *S-altdef* [*symmetric*]
finally have *lowered R* ≺[*SD* (*lift R i*)] *sds S*
using *R-wf i lowered.sds-wf*[*OF R-wf*] *sds-S-wf*
by (*subst lift-SD-strict-iff*) (*simp-all add*: *agents′-subset alts′-subset*)
hence *manipulable-profile* (*lift R*) *i lift-Ri′*
by (*simp add*: *manipulable-profile-def lowered-def S-def*)
with *strategyproof*[*OF lift-wf*[*OF R-wf*] *i′ wf-lift-Ri′*] **show** *False* **by** *contradiction*
qed

end

locale *sds-lowering-anonymous-neutral-sdeff-stratproof* =
sds-lowering-anonymous + *sds-lowering-neutral* +
sds-lowering-sd-efficient + *sds-lowering-strategyproof*

end

7 Random Dictatorship

theory *Random-Dictatorship*
imports
Complex-Main
Social-Decision-Schemes
begin

We define Random Dictatorship as a social decision scheme on total preorders (i.e. agents are allowed to have ties in their rankings) by first selecting an agent uniformly at random and then selecting one of that agents' most preferred alternatives uniformly at random. Note that this definition also works for weak preferences.

definition *random-dictatorship* :: *'agent set* ⇒ *'alt set* ⇒ (*'agent*, *'alt*) *pref-profile* ⇒ *'alt lottery* **where**
random-dictatorship-auxdef:
random-dictatorship agents alts R =
do {
i ← *pmf-of-set agents*;
pmf-of-set (*Max-wrt-among* (*R i*) *alts*)
}

context *election*
begin

abbreviation *RD* :: (*'agent*, *'alt*) *pref-profile* ⇒ *'alt lottery* **where**
RD ≡ *random-dictatorship agents alts*

lemma *random-dictatorship-def*:
assumes *is-pref-profile R*
shows *RD R* =
do {
i ← *pmf-of-set agents*;
pmf-of-set (*favorites R i*)
}
proof −
from *assms* **interpret** *pref-profile-wf agents alts R* **.**
show *?thesis* **by** (*simp add*: *random-dictatorship-auxdef favorites-altdef*)
qed

lemma *random-dictatorship-unique-favorites*:
assumes *is-pref-profile R has-unique-favorites R*
shows *RD R* = *map-pmf* (*favorite R*) (*pmf-of-set agents*)
proof −
from *assms*(*1*) **interpret** *pref-profile-wf agents alts R* **.**

```
  from assms(2) interpret pref-profile-unique-favorites agents alts R by unfold-locales
  show ?thesis unfolding random-dictatorship-def[OF assms(1)] map-pmf-def
    by (intro bind-pmf-cong) (auto simp: unique-favorites pmf-of-set-singleton)
qed

lemma random-dictatorship-unique-favorites':
  assumes is-pref-profile R has-unique-favorites R
  shows   RD R = pmf-of-multiset (image-mset (favorite R) (mset-set agents))
  using assms by (simp add: random-dictatorship-unique-favorites map-pmf-of-set)

lemma pmf-random-dictatorship:
  assumes is-pref-profile R
  shows pmf (RD R) x =
          (∑ i∈agents. indicator (favorites R i) x /
            real (card (favorites R i))) / real (card agents)
proof −
  from assms(1) interpret pref-profile-wf agents alts R .
  from nonempty-dom have card agents > 0 by (auto simp del: nonempty-agents)
  hence ennreal (pmf (RD R) x) =
            ennreal ((∑ 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
          divide-ennreal [symmetric] ennreal-of-nat-eq-real-of-nat sum-nonneg)
  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** *wf*: *is-pref-profile R* **and** *perm*: π *permutes agents*
interpret *pref-profile-wf agents alts R* **by** *fact*
from *wf-permute-agents*[*OF perm*]
have *RD* $(R \circ \pi)$ = *map-pmf* π (*pmf-of-set agents*) $\gg\!=$ $(\lambda i.$ *pmf-of-set* (*favorites R i*))
by (*simp add*: *bind-map-pmf random-dictatorship-def o-def favorites-def*)
also from *perm wf* **have** ... = *RD R*
by (*simp add*: *map-pmf-of-set-inj permutes-inj-on permutes-image random-dictatorship-def*)
finally show *RD* $(R \circ \pi)$ = *RD R* **.**
qed

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 on 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*]
by (*auto intro*!: *card-mono nn-integral-mono-AE*
simp: *random-dictatorship-def R-wf R′-wf AE-measure-pmf-iff Max-wrt-prefs-finite*
emeasure-pmf-of-set Int-absorb2 favorites-def
Max-wrt-prefs-nonempty card-gt-0-iff)
thus *lottery-prob* (*RD* (*R*(*i* := *Ri′*))) (*preferred-alts* (*R i*) *x*)
≤ *lottery-prob* (*RD R*) (*preferred-alts* (*R i*) *x*)
by (*simp add*: *measure-pmf.emeasure-eq-measure*)
qed (*insert R-wf R′-wf*, *simp-all add*: *RD.sds-wf i*)
qed

end

end

8 Random Serial Dictatorship

theory *Random-Serial-Dictatorship*
imports
Complex-Main
Social-Decision-Schemes
Random-Dictatorship
begin

Random Serial Dictatorship is an anonymous, neutral, strongly strategyproof, and ex-post efficient Social Decision Scheme that extends Random Dictatorship to the domain of weak preferences.

We define RSD using a fold over a random permutation. Effectively, we choose a random order of the agents (in the form of a list) and then traverse that list from left to right, where each agent in turn removes all the

alternatives that are not top-ranked among the remaining ones.

definition *random-serial-dictatorship* ::
  'agent set ⇒ 'alt set ⇒ ('agent, 'alt) pref-profile ⇒ 'alt lottery **where**
 random-serial-dictatorship agents alts R =
   fold-bind-random-permutation (λi alts. Max-wrt-among (R i) alts) pmf-of-set alts agents

The following two facts correspond give an alternative recursive definition to the above definition, which uses random permutations and list folding.

lemma *random-serial-dictatorship-empty* [*simp*]:
 random-serial-dictatorship {} alts R = pmf-of-set alts
 by (*simp add*: *random-serial-dictatorship-def*)

lemma *random-serial-dictatorship-nonempty*:
 finite agents ⟹ agents ≠ {} ⟹
  random-serial-dictatorship agents alts R =
   do {
    i ← pmf-of-set agents;
    random-serial-dictatorship (agents − {i}) (Max-wrt-among (R i) alts) R
   }
 by (*simp add*: *random-serial-dictatorship-def*)

We define the RSD winners w.r.t. a given set of alternatives and a fixed permutation (i.e. list) of agents. In contrast to the above definition, the RSD winners are determined by traversing the list of agents from right to left. This may seem strange, but it makes induction much easier, since induction over *foldr* does not require generalisation over the set of alternatives and is therefore much easier than over *foldl*.

definition *rsd-winners* **where**
 rsd-winners R alts agents = foldr (λi alts. Max-wrt-among (R i) alts) agents alts

lemma *rsd-winners-empty* [*simp*]: *rsd-winners R alts [] = alts*
 by (*simp add*: *rsd-winners-def*)

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

lemma *rsd-winners-map* [*simp*]:
 rsd-winners R alts (map f agents) = rsd-winners (R ∘ f) alts agents
 by (*simp add*: *rsd-winners-def foldr-map o-def*)

There is now another alternative definition of RSD in terms of the RSD winners. This will mostly be used for induction.

lemma *random-serial-dictatorship-altdef*:
 assumes *finite agents*

shows *random-serial-dictatorship agents alts R* =
do {
agents′ ← *pmf-of-set* (*permutations-of-set agents*);
pmf-of-set (*rsd-winners R alts agents′*)
}
by (*simp add*: *random-serial-dictatorship-def*
fold-bind-random-permutation-foldr assms rsd-winners-def)

The following lemma shows that folding from left to right yields the same distribution. This is probably the most commonly used definition in the literature, along with the recursive one.

lemma *random-serial-dictatorship-foldl*:
assumes *finite agents*
shows *random-serial-dictatorship agents alts R* =
do {
agents′ ← *pmf-of-set* (*permutations-of-set agents*);
pmf-of-set (*foldl* (λ*alts i*. *Max-wrt-among* (*R i*) *alts*) *alts agents′*)
}
by (*simp add*: *random-serial-dictatorship-def fold-bind-random-permutation-foldl assms*)

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**
RSD-pareto-eqclass agents alts R A ⟷
A ≠ {} ∧ *A* ⊆ *alts* ∧ (∀ *x*∈*A*. ∀ *y*∈*alts*. *y* ∈ *A* ⟷ (∀ *i*∈*agents*. *R i x y*))

lemma *RSD-pareto-eqclassI*:
assumes *A* ≠ {} *A* ⊆ *alts* ⋀*x y*. *x* ∈ *A* ⟹ *y* ∈ *alts* ⟹ *y* ∈ *A* ⟷ (∀ *i*∈*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* ≠ {} *A* ⊆ *alts* ⋀*x y*. *x* ∈ *A* ⟹ *y* ∈ *alts* ⟹ *y* ∈ *A* ⟷ (∀ *i*∈*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* $i \in$ *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*!]:
$alts \neq \{\} \Longrightarrow$ *RSD-pareto-eqclass* $\{\}$ *alts R 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*
$i \in$ *agents agents′* $\subseteq$ *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-eqclassD*[*OF assms*(*1*)]
show *?case* **unfolding** *Max-wrt-among-total-preorder*
by (*blast intro*: *trans*)
qed (*insert RSD-pareto-eqclassD*[*OF assms*(*1*)] *assms*(*2*),
simp-all add: *Int-absorb1 Int-absorb2 finite-subset*)[*2*]
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′* $\subseteq$ *agents*
shows *rsd-winners R alts′ agents′* $\subseteq$ *alts′*
proof −
{
fix *i* **assume** $i \in$ *agents*
then interpret *total-preorder-on alts R i* **by** *simp*
have *Max-wrt-among* (*R i*) $A \subseteq A$ **for** *A*
using *Max-wrt-among-subset* **by** *blast*
} **note** A = *this*

from ‹*set agents′* $\subseteq$ *agents*› **show** *rsd-winners R alts′ agents′* $\subseteq$ *alts′*
using *A* **by** (*induction agents′*) *auto*
qed

There is always at least one RSD winner.

lemma *rsd-winners-nonempty*:

```
  assumes finite: finite alts  and alts' ≠ {}  set agents' ⊆ agents alts' ⊆ alts
  shows   rsd-winners R alts' agents' ≠ {}
proof −
  {
    fix i assume i ∈ agents
    then interpret total-preorder-on alts R i by simp
    have Max-wrt-among (R i) A ≠ {} if A ⊆ alts A ≠ {} for A
     using that assms by (intro Max-wrt-among-nonempty) (auto simp: Int-absorb)
  } note B = this

  with ‹set agents' ⊆ agents› ‹alts' ⊆ alts› ‹alts' ≠ {}›
    show rsd-winners R alts' agents' ≠ {}
  proof (induction agents')
    case (Cons i agents')
   with B[of i rsd-winners R alts' agents'] rsd-winners-subset[of agents' alts'] finite
wf
      show ?case by auto
  qed simp
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-wf =
  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 $\longleftrightarrow$ *A* $\neq$ {} $\wedge$ ($\forall$ *x*$\in$*A*. $\forall$ *y*$\in$*A*. *S x y*)

private lemma *indiff-set-mono*: *indiff-set S A* $\Longrightarrow$ *B* $\subseteq$ *A* $\Longrightarrow$ *B* $\neq$ {} $\Longrightarrow$ *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**
RSD-set-rel S A B $\longleftrightarrow$ *indiff-set S B* $\wedge$ ($\forall$ *x*$\in$*A*. $\forall$ *y*$\in$*B*. *S x y*)

The most-preferred alternatives (w.r.t. R) among any non-empty set of alternatives form an indifference set w.r.t. R.

private lemma *indiff-set-Max-wrt-among*:
assumes *finite carrier A* $\subseteq$ *carrier A* $\neq$ {} *total-preorder-on carrier S*
shows *indiff-set S* (*Max-wrt-among S A*)
unfolding *indiff-set-def*
proof
from *assms*(*4*) **interpret** *total-preorder-on carrier S* **.**
from *assms*(*1*−*3*)
show *Max-wrt-among S A* $\neq$ {} **by** (*intro Max-wrt-among-nonempty*) *auto*
from *assms*(*1*−*3*) **show** $\forall$ *x*$\in$*Max-wrt-among S A*. $\forall$ *y*$\in$*Max-wrt-among S A*. *S x y*
by (*auto simp*: *indiff-set-def Max-wrt-among-total-preorder*)
qed

We now consider the set of RSD winners in the setting of a preference profile R and a manipulated profile $R(i := Ri')$. 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 *rsd-winners-manipulation-aux*:
assumes *wf*: *total-preorder-on alts Ri′*
and *i*: *i* $\in$ *agents* **and** *set agents′* $\subseteq$ *agents finite agents*
and *finite*: *finite alts* **and** *alts* $\neq$ {}
defines [*simp*]: *w′* $\equiv$ *rsd-winners* (*R*(*i* := *Ri′*)) *alts* **and** [*simp*]: *w* $\equiv$ *rsd-winners R alts*
shows *w′ agents′* = *w agents′* $\vee$ *RSD-set-rel* (*R i*) (*w′ agents′*) (*w agents′*)
using ‹*set agents′* $\subseteq$ *agents*›
proof (*induction agents′*)

```
  case (Cons j agents')
  from wf i interpret Ri: total-preorder-on alts R i by simp
  from wf Cons.prems interpret Rj: total-preorder-on alts R j by simp
  from wf interpret Ri': total-preorder-on alts Ri' .
  from wf assms Cons.prems
    have indiff-set: indiff-set (R i) (Max-wrt-among (R i) (rsd-winners R alts
agents'))
   by (intro indiff-set-Max-wrt-among[OF finite] rsd-winners-wf) simp-all

  show ?case
  proof (cases j = i)
   assume j [simp]: j = i
  from indiff-set Cons have RSD-set-rel (R i) (w' (j # agents')) (w (j # agents'))
     unfolding RSD-set-rel-def
   by (auto simp: Ri.Max-wrt-among-total-preorder Ri'.Max-wrt-among-total-preorder)
   thus ?case ..
  next
   assume j [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'))
       by (intro indiff-set-mono[OF indiff-set] Rj.Max-wrt-among-subset)
          (simp-all 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
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 wf: total-preorder-on alts Ri'
     and i: i ∈ agents and set agents' = agents finite agents
```

and *finite*: *finite alts* **and** *alts* $\neq \{\}$
defines [*simp*]: $w' \equiv$ *rsd-winners* $(R(i := Ri'))$ *alts* **and** [*simp*]: $w \equiv$ *rsd-winners R alts*
shows $\forall x{\in}w'$ *agents'*. $\forall y{\in}w$ *agents'*. $x \preceq[R\ i]\ y$
proof −
have w' *agents'* $= w$ *agents'* $\vee$ *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*
qed (*auto simp*: *RSD-set-rel-def*)
qed

end

The lottery that RSD yields is well-defined.

lemma *random-serial-dictatorship-support*:
assumes *finite agents finite alts agents'* $\subseteq$ *agents alts'* $\neq \{\}$ *alts'* $\subseteq$ *alts*
shows *set-pmf* (*random-serial-dictatorship agents' alts' R*) $\subseteq$ *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''*)) $\subseteq$ *alts'*
if *agents''* $\in$ *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'* $\subseteq$ *agents*
shows *rsd-winners* (*permute-profile* σ *R*) *alts agents'* $= \sigma$ ‘ *rsd-winners R alts agents'*
using ‹*set agents'* $\subseteq$ *agents*›
proof (*induction agents'*)
case *Nil*
from *perm* **show** *?case* **by** (*simp add*: *permutes-image*)
next
case (*Cons i agents'*)
from *wf Cons* **interpret** *total-preorder-on alts R i* **by** *simp*
from *perm Cons* **show** *?case*
by (*simp add*: *permute-profile-map-relation Max-wrt-among-map-relation-bij*

permutes-bij)
qed

lemma *random-serial-dictatorship-singleton*:
assumes *finite agents finite alts agents'* ⊆ *agents x* ∈ *alts*
shows *random-serial-dictatorship agents'* {*x*} *R* = *return-pmf x* (**is** *?d* = -)
proof −
from *assms* **have** *set-pmf ?d* ⊆ {*x*}
by (*intro random-serial-dictatorship-support*) *simp-all*
thus *?thesis* **by** (*simp add*: *set-pmf-subset-singleton*)
qed

end

8.2 Proofs of properties

With all the facts that we have proven about the RSD winners, the hard work is mostly done. We can now simply fix some arbitrary order of the agents, apply the theorems about the RSD winners, and show the properties we want to show without doing much reasoning about probabilities.

context *election*
begin

abbreviation *RSD* ≡ *random-serial-dictatorship agents alts*

8.2.1 Well-definedness

sublocale *RSD*: *social-decision-scheme agents alts RSD*
using *pref-profile-wf.random-serial-dictatorship-support*[*of agents alts*]
by *unfold-locales* (*simp-all add*: *lotteries-on-def*)

8.2.2 RD extension

lemma *RSD-extends-RD*:
assumes *wf*: *is-pref-profile R* **and** *unique*: *has-unique-favorites R*
shows *RSD R* = *RD R*
proof −
from *wf* **interpret** *pref-profile-wf agents alts R* **.**
from *unique* **interpret** *pref-profile-unique-favorites* **by** *unfold-locales*
have *RSD R* = *pmf-of-set agents* ⤜
(λ*i*. *random-serial-dictatorship* (*agents* − {*i*}) (*favorites R i*) *R*)
by (*simp add*: *random-serial-dictatorship-nonempty favorites-altdef Max-wrt-def*)
also from *assms* **have** ... = *pmf-of-set agents* ⤜ (λ*i*. *return-pmf* (*favorite R i*))
by (*intro bind-pmf-cong refl*, *subst random-serial-dictatorship-singleton* [*symmetric*])
(*auto simp*: *unique-favorites favorite-in-alts*)
also from *assms* **have** ... = *RD R*
by (*simp add*: *random-dictatorship-unique-favorites map-pmf-def*)
finally show *?thesis* **.**

qed

8.2.3 Anonymity

Anonymity is a direct consequence of the fact that we randomise over all permutations in a uniform way.

sublocale *RSD*: *anonymous-sds agents alts RSD*
proof
fix π R **assume** *perm*: π *permutes agents* **and** *wf*: *is-pref-profile R*
let *?f* = λ*agents'*. *pmf-of-set* (*rsd-winners R alts agents'*)
from *perm wf* **have** *RSD* ($R \circ \pi$) = *map-pmf* (*map* π) (*pmf-of-set* (*permutations-of-set agents*)) $\gg\!=$ *?f*
by (*simp add*: *random-serial-dictatorship-altdef bind-map-pmf*)
also from *perm* **have** ... = *RSD R*
by (*simp add*: *map-pmf-of-set-inj permutes-inj-on inj-on-mapI*
permutations-of-set-image-permutes random-serial-dictatorship-altdef)
finally show *RSD* ($R \circ \pi$) = *RSD R* **.**
qed

8.2.4 Neutrality

Neutrality follows from the fact that the RSD winners of a permuted profile are simply the image of the original RSD winners under the permutation.

sublocale *RSD*: *neutral-sds agents alts RSD*
proof
fix σ R **assume** *perm*: σ *permutes alts* **and** *wf*: *is-pref-profile R*
from *wf* **interpret** *pref-profile-wf agents alts R* **.**
from *perm* **show** *RSD* (*permute-profile* σ R) = *map-pmf* σ (*RSD R*)
by (*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)
qed

8.2.5 Ex-post efficiency

Ex-post efficiency follows from the fact that the set of RSD winners is a Pareto-equivalence class.

sublocale *RSD*: *ex-post-efficient-sds agents alts RSD*
proof
fix R **assume** *wf*: *is-pref-profile R*
then interpret *pref-profile-wf agents alts R* **.**
{
fix x **assume** x: $x \in$ *set-pmf* (*RSD R*) $x \in$ *pareto-losers R*
from x(*2*) **obtain** y **where** [*simp*]: $y \in$ *alts* **and** *pareto*: $y \succ$[*Pareto*(*R*)] x
by (*cases rule*: *pareto-losersE*)
from x **have** [*simp*]: $x \in$ *alts* **using** *pareto-loser-in-alts* **by** *simp*

```
    from x(1) obtain agents' where agents': set agents' = agents and
        x ∈ set-pmf (pmf-of-set (rsd-winners R alts agents'))
      by (auto simp: random-serial-dictatorship-altdef dest: permutations-of-setD)
    with wf have x': x ∈ rsd-winners R alts agents'
      using rsd-winners-wf[where alts' = alts and agents' = agents']
      by (subst (asm) set-pmf-of-set) (auto simp: permutations-of-setD)

    from wf agents'
      have RSD-pareto-eqclass agents alts R (rsd-winners R alts agents')
      by (intro RSD-pareto-eqclass-rsd-winners) simp-all
    hence winner-iff: y ∈ rsd-winners R alts agents' ⟷ (∀ i∈agents. x ⪯[R i] y)
      if x ∈ rsd-winners R alts agents' y ∈ alts for x y
      using that unfolding RSD-pareto-eqclass-def by blast
    from x' pareto winner-iff[of x y] winner-iff[of y x] have False
      by (force simp: strongly-preferred-def Pareto-iff)
  }
  thus set-pmf (RSD R) ∩ pareto-losers R = {} by blast
qed
```

8.2.6 Strong strategy-proofness

Strong strategy-proofness is slightly more difficult to show. We have already shown that the set of RSD winners for the truthful profile is always set-preferred (by the manipulating agent) to the RSD winners for the manipulated profile. This can now be used to show strategy-proofness: We recall that the set of RSD winners is always an indifference class. Therefore, given any fixed alternative x and considering a fixed order of the agents, either all of the RSD winners in the original profile are at least as good as x or none of them are, and, since the original RSD winners are set-preferred to the manipulated ones, none of the RSD winners in the manipulated case are at least as good than x either in that case. This means that for a fixed order of agents, either the probability that the original outcome is at least as good as x is 1 or the probability that the manipulated outcome is at least as good as x is 0. Therefore, the original lottery is clearly SD-preferred to the manipulated one.

```
sublocale RSD: strongly-strategyproof-sds agents alts RSD
proof (unfold-locales, rule)
  fix R i Ri' x
  assume wf: is-pref-profile R and i [simp]: i ∈ agents and x: x ∈ alts and
      wf': total-preorder-on alts Ri'
  interpret R: pref-profile-wf 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-wf agents alts R' .
  note wf = wf wf'
  let ?A = preferred-alts (R i) x
  from wf interpret Ri: total-preorder-on alts R i by simp
```

```
  {
    fix agents' assume agents': agents' ∈ permutations-of-set agents
    from agents' have [simp]: set agents' = agents
      by (simp add: permutations-of-set-def)

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

    from ‹?W ≠ {}› obtain y where y: y ∈ ?W by blast
    with winners have [simp]: y ∈ alts by blast
    from wf' i have mono: ∀ x∈?W'. ∀ y∈?W. R i x y unfolding R'-def
      by (intro R.rsd-winners-manipulation) simp-all

    have lottery-prob (pmf-of-set ?W) ?A ≥ lottery-prob (pmf-of-set ?W') ?A
    proof (cases y ≻[R i] x)
      case True
      with y RSD-pareto-eqclass-indiff-set[OF indiff-set(1), of i]  winners
        have ?W ⊆ preferred-alts (R i) x
        by (auto intro: Ri.trans simp: preferred-alts-def)
      with winners show ?thesis
        by (subst (2) measure-pmf-of-set) (simp-all add: Int-absorb2)
    next
      case False
      with y mono have ?W' ∩ preferred-alts (R i) x = {}
        by (auto intro: Ri.trans simp: preferred-alts-def)
      with winners show ?thesis
        by (subst (1) measure-pmf-of-set)
           (simp-all add: Int-absorb2 one-ereal-def measure-nonneg)
    qed
    hence emeasure (measure-pmf (pmf-of-set ?W)) ?A ≥ emeasure (measure-pmf
(pmf-of-set ?W')) ?A
      by (simp add: measure-pmf.emeasure-eq-measure)
  }
  hence emeasure (measure-pmf (RSD R)) ?A ≥ emeasure (measure-pmf (RSD
R')) ?A
    by (auto simp: random-serial-dictatorship-altdef AE-measure-pmf-iff
          intro!: nn-integral-mono-AE)
  thus lottery-prob (RSD R) ?A ≥ lottery-prob (RSD R') ?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*
imports
Complex-Main
../Elections
keywords
preference-profile :: *thy-goal*
begin

ML-file ‹*preference-profiles.ML*›

context *election*
begin

lemma *preferred-alts-prefs-from-table*:
assumes *prefs-from-table-wf agents alts xs i* ∈ *set* (*map fst xs*)
shows *preferred-alts* (*prefs-from-table xs i*) *x* =
of-weak-ranking-Collect-ge (*rev* (*the* (*map-of xs i*))) *x*
proof −
interpret *pref-profile-wf agents alts prefs-from-table xs*
by (*intro pref-profile-from-tableI assms*)
from *assms* **have** [*simp*]: *i* ∈ *agents* **by** (*auto simp*: *prefs-from-table-wf-def*)
have *of-weak-ranking-Collect-ge* (*rev* (*the* (*map-of xs i*))) *x* =
Collect (*of-weak-ranking* (*the* (*map-of xs i*)) *x*)
by (*rule eval-Collect-of-weak-ranking* [*symmetric*])
also from *assms*(*2*) **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*) *this*]
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*: $i \in$ *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*
    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 (λ*z*. *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* $i \in$ *agents*)
assume *i*: $i \in$ *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** $i \in$ *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*: $i \notin$ *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* = (λ- -. *False*)
by (*intro ext*) (*auto simp*: *prefs-from-table-def*)
hence *favorites* (*prefs-from-table xs*) *i* = {}
by (*simp add*: *favorites-def Max-wrt-altdef*)
also from *i'* **have** ... = *favorites-prefs-from-table xs i*
by (*induction xs i rule*: *favorites-prefs-from-table.induct*) *simp-all*
finally show *?thesis* **..**
qed

function *weak-ranking-prefs-from-table* **where**
$i \neq j \Longrightarrow$ *weak-ranking-prefs-from-table* ((*i*,*x*)#*xs*) *j* = *weak-ranking-prefs-from-table xs j*
| $i = j \Longrightarrow$ *weak-ranking-prefs-from-table* ((*i*,*x*)#*xs*) *j* = *x*
| *weak-ranking-prefs-from-table* [] *j* = []
by (*metis list.exhaust old.prod.exhaust*) *auto*
termination by *lexicographic-order*

lemma *eval-weak-ranking-prefs-from-table*:
assumes *prefs-from-table-wf agents alts xs*
shows *weak-ranking-prefs-from-table xs i* = *weak-ranking* (*prefs-from-table xs i*)
proof (*cases* $i \in$ *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* = (λ- -. *False*)
by (*intro ext*) (*auto simp*: *prefs-from-table-def*)
hence *weak-ranking* (*prefs-from-table xs i*) = [] **by** *simp*
also from *i′* **have** ... = *weak-ranking-prefs-from-table xs i*
by (*induction xs i rule*: *weak-ranking-prefs-from-table.induct*) *simp-all*
finally show *?thesis* **..**
qed

lemma *eval-prefs-from-table-aux*:
assumes *R* ≡ *prefs-from-table xs prefs-from-table-wf agents alts xs*
shows *R i a b* ⟷ *prefs-from-table xs i a b*
a ≺[*R i*] *b* ⟷ *prefs-from-table xs i a b* ∧ ¬*prefs-from-table xs i b a*
anonymous-profile R = *mset* (*map snd xs*)
election agents alts ⟹ *i* ∈ *set* (*map fst xs*) ⟹
preferred-alts (*R i*) *x* =
of-weak-ranking-Collect-ge (*rev* (*the* (*map-of xs i*))) *x*
election agents alts ⟹ *i* ∈ *set* (*map fst xs*) ⟹
favorites R i = *favorites-prefs-from-table xs i*
election agents alts ⟹ *i* ∈ *set* (*map fst xs*) ⟹
weak-ranking (*R i*) = *weak-ranking-prefs-from-table xs i*
election agents alts ⟹ *i* ∈ *set* (*map fst xs*) ⟹
favorite R i = *the-elem* (*favorites-prefs-from-table xs i*)
election agents alts ⟹
has-unique-favorites R ⟷ *list-all* (λ*z*. *is-singleton* (*hd* (*snd z*))) *xs*
using *assms prefs-from-table-wfD*[*OF assms*(*2*)]
by (*simp-all add*: *strongly-preferred-def favorite-def anonymise-prefs-from-table election.preferred-alts-prefs-from-table election.eval-favorites-prefs-from-table election.has-unique-favorites-prefs-from-table eval-weak-ranking-prefs-from-table*)

lemma *pref-profile-from-tableI′*:
assumes *R1* ≡ *prefs-from-table xss prefs-from-table-wf agents alts xss*
shows *pref-profile-wf agents alts R1*
using *assms* **by** (*simp add*: *pref-profile-from-tableI*)

ML ‹

signature PREFERENCE-PROFILES-CMD =
sig

type info

val preference-profile :
(*term* ∗ *term*) ∗ ((*binding* ∗ (*term* ∗ *term list list*) *list*) *list*) −> *Proof.context*
−> *Proof.state*

```
val preference-profile-cmd :
  (string * string) * ((binding * (string * string list list) list) list) ->
    Proof.context -> Proof.state

val get-info : term -> Proof.context -> info
val add-info : term -> info -> Context.generic -> Context.generic
val transform-info : info -> morphism -> info

end

structure Preference-Profiles-Cmd : PREFERENCE-PROFILES-CMD =
struct

open Preference-Profiles


type info =
  { term : term, def-thm : thm, wf-thm : thm, wf-raw-thm : thm, binding : binding,
    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 aconv o apply2 fst) (single o fst)
  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, [Code.singleton-default-equation-attrib]),
t)) lthy
    val ty = HOLogic.mk-prodT (agentT, HOLogic.listT (HOLogic.mk-setT altT))
    val args' =
      args |> map (fn x => x ||> map (HOLogic.mk-set altT) ||> HOLogic.mk-list
alt-setT)
    val t-raw =
      args'
        |> map HOLogic.mk-prod
        |> HOLogic.mk-list ty
    val t = Const (@{const-name prefs-from-table},
            HOLogic.listT ty --> pref-profileT agentT altT) $ t-raw
    val ((prefs, prefs-def), lthy) = define t
    val prefs-from-table-wf-const =
       Const (@{const-name prefs-from-table-wf}, HOLogic.mk-setT agentT -->
HOLogic.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) |> HO-
Logic.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' = lthy |> put-simpset HOL-ss |> Simplifier.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}
      in
       Local-Defs.unfold-tac lthy defs
        THEN ALLGOALS (resolve-tac lthy [@{thm prefs-from-table-wfI}])
        THEN Local-Defs.unfold-tac lthy @{thms is-finite-weak-ranking-def list.set
insert-iff
            empty-iff simp-thms list.map snd-conv fst-conv}
            THEN ALLGOALS (TRY o REPEAT-ALL-NEW (eresolve-tac lthy
@{thms disjE}))
          THEN ALLGOALS (TRY o Hypsubst.hyp-subst-tac lthy)
          THEN ALLGOALS (Simplifier.asm-full-simp-tac lthy')
            THEN ALLGOALS (TRY o REPEAT-ALL-NEW (resolve-tac lthy
@{thms conjI}))
          THEN distinct-subgoals-tac
      end

  fun after-qed [wf-thms-raw] lthy =
    let
      fun prep-thms attrs suffix (thms : thm list) binding =
        (((qualify binding suffix, attrs), [(thms,[])]))
       fun prep-thmss simp suffix thmss = map2 (prep-thms simp suffix) thmss
bindings
      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
      val eval-thmss = map2 (fn def => fn wf =>
        map (fn thm => thm OF [def, wf]) @{thms eval-prefs-from-table-aux})
        defs wf-thms-raw
      val wf-thms = map2 (fn def => fn wf =>
        @{thm pref-profile-from-tableI'} OF [def, wf]) defs wf-thms-raw
      val mk-infos =
        let
          fun aux acc (bdg::bdgs) (t::ts) (r::raws) (def::def-thms) (wf::wf-thms)
                (wf-raw::wf-raw-thms) (evals::eval-thmss) =
            aux ((t, {binding = bdg, term = t, raw = r, def-thm = def, wf-thm =
wf,
```

```
                wf-raw-thm = wf-raw, eval-thms = evals}) :: acc)
                bdgs ts raws def-thms wf-thms wf-raw-thms eval-thmss
            | aux acc [] - - - - - - - = (acc : (term * info) list)
            | aux - - - - - - - - - = raise Match
          in
            aux []
          end
          val infos = mk-infos bindings prefs-terms raws defs wf-thms wf-thms-raw
eval-thmss
      in
        lthy
        |> note wf-thms-raw wf-raw []
        |> note wf-thms wf @{attributes [simp]}
        |> notes eval-thmss eval []
       |> Local-Theory.declaration {syntax = false, pervasive = false, pos = here}
          (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);

end
›

end
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
    Const (@{const-name Code-Numeral.int-of-integer}, -) $ x => dest-num x
  | - => HOLogic.dest-number x

fun dest-rat-number t =
  case t of
    (Const (@{const-name Rings.divide-class.divide},-)) $ a $ b
      => Rat.make (snd (dest-num a), snd (dest-num b))
  | (Const (@{const-name Groups.uminus-class.uminus},-)) $ a
      => ~ (dest-rat-number a)
  | (Const (@{const-name Rat.field-char-0-class.of-rat},-)) $ a => dest-rat-number
a
  | (Const (@{const-name Rat.Frct}, -) $ (Const (@{const-name Product-Type.Pair},
-) $ a $ b))
      => Rat.make (snd (dest-num a), snd (dest-num b))
  | - => Rat.of-int (snd (dest-num t));

fun mk-rat-number ty r =
  case Rat.dest r of
    (a, 1) => HOLogic.mk-number ty a
  | (a, b) =>
      Const (@{const-name Rings.divide-class.divide}, ty --> ty --> ty) $
        HOLogic.mk-number ty a $ HOLogic.mk-number ty b

end

›
```

ML ‹

```
signature LP-PARAMS =
sig

type T
val print : T -> string
val read : string -> T option
```

```
val compare : (T * T) -> General.order
val negate : T -> T
val from-int : int -> T

end;

signature LINEAR-PROGRAM-COMMON =
sig
  exception QSOpt-Parse
  datatype 'a infty = Finite of 'a | Pos-Infty | Neg-Infty;
  datatype comparison = LEQ | EQ | GEQ
  datatype optimization-mode = MAXIMIZE | MINIMIZE
  datatype 'a result = Optimal of 'a * (string * 'a) list | Unbounded | Infeasible |
Unknown
  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

  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 gen-print-bound : ('a -> string) -> 'a bound -> string
  val gen-print-linterm :
    (('a * 'a -> General.order) * (int -> 'a) * ('a -> string) * ('a -> 'a)) ->
    'a linterm -> string
  val gen-print-constraint :
    (('a * 'a -> General.order) * (int -> 'a) * ('a -> string) * ('a -> 'a)) ->
    'a constraint -> string
  val gen-print-program :
    (('a * 'a -> General.order) * (int -> 'a) * ('a -> string) * ('a -> 'a)) ->
    'a prog -> string
  val gen-read-result : (string -> 'a option) -> 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 : Path.T -> T prog -> unit
val solve-program : T prog -> T result
val read-result : string -> T result

end;


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

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
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-Infty = INF
  | print-infty f (Finite x) = f x

fun print-comparison LEQ = <=
  | print-comparison EQ = =
  | print-comparison GEQ = >=

fun print-optimization-mode MINIMIZE = MINIMIZE
  | print-optimization-mode MAXIMIZE = MAXIMIZE

fun gen-print-bound - (Neg-Infty, v, Pos-Infty) = v ^ 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)
    fun eq x = (cmp (c, x) = EQUAL)
    val one = eq (from-int 1)
    val mone = eq (from-int (~ 1))
    val c' =
      if first andalso one then
      else if first andalso mone then -
      else if first then print c ^
      else if mone then  -
      else if one then  +
      else if neg then  -  ^ print (negate c) ^
      else  +  ^ print c ^
  in
    c' ^ v
  end

fun gen-print-linterm ops t =
  let
    val n = length t
    val print-summand = gen-print-summand ops
    fun go (c, v) (i, acc) = (i+1, print-summand (i = n) c v ^ acc)
  in
    snd (fold go (rev t) (1, ))
  end

fun gen-print-constraint (ops as (-, -, print, -)) (lhs, cmp, rhs) =
   gen-print-linterm ops lhs ^   ^ print-comparison cmp ^   ^ print rhs

fun gen-print-program (ops as (-, -, print, -)) (mode, obj, constrs, bnds) =
  let
    val padding = replicate-string 4
    fun mk-block s f xs = (s :: map (prefix padding o f) xs)
    fun mk-block' s f xs = if null xs then [] else mk-block s f xs
    val lines =
      mk-block (print-optimization-mode mode) (gen-print-linterm ops) [obj] @
      mk-block' ST (gen-print-constraint ops) constrs @
      mk-block' BOUNDS (gen-print-bound print) bnds @ [END, ]
  in
    cat-lines lines
  end



exception QSOpt-Parse

fun read-status x =
  if String.isPrefix status  x andalso not (String.isPrefix status = x) then
    let
```

```
      val statuses = [OPTIMAL, INFEASIBLE, UNBOUNDED]
    in
      case find-first (fn s => String.isPrefix (status ^ s) x) statuses of
        NONE => SOME UNKNOWN
      | SOME y => SOME y
    end
  else
    NONE

fun apply - - [] = NONE
  | apply abort f (x :: xs) =
      if abort x then
        NONE
      else case f x of
        NONE => apply abort f xs
      | SOME y => SOME (y, xs)

fun apply-repeat abort (f : string -> 'a option) : string list -> 'a list * string list
=
  let
    fun go acc xs =
      case apply abort f xs of
        NONE => (rev acc, xs)
      | SOME (y,xs) => go (y :: acc) xs
  in
    go []
  end

fun the-apply f xs =
  case apply (K false) f xs of
    NONE => raise QSOpt-Parse
  | SOME y => y

fun apply-unit p xs =
  case apply (not o p) (K (SOME ())) xs of
    NONE => raise QSOpt-Parse
  | SOME (-, xs) => xs

fun gen-read-value read x =
  let
    val x = unprefix Value =  x
  in
    read x
  end
    handle Fail - => NONE

val trim =
  let
    fun chop [] = []
```

```
      | chop (l as (x::xs)) = if Char.isSpace x then chop xs else l
  in
    String.implode o chop o rev o chop o rev o String.explode
  end

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

fun gen-read-result read s =
  let
    val s = s |> split-lines |> map trim
    val (status, s) = the-apply read-status s
    val (result, -) =
      if status = OPTIMAL then
        let
          val (value, s) = the-apply (gen-read-value read) s
          val s = apply-unit (fn x => x = VARS:) s
          val (vars, s) = apply-repeat (String.isSuffix :) (gen-read-assignment read) 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 =
  File.write filename (print-program prog)


val read-result = gen-read-result LP-Params.read

fun solve-program prog =
  Isabelle-System.with-tmp-file prog lp (fn lpname =>
    Isabelle-System.with-tmp-file prog sol (fn resultname =>
      let
        val - = save-program lpname prog
        val esolver-path = getenv QSOPT-EXACT-PATH
        val esolver = if esolver-path =  then esolver else esolver-path
        val command =  Bash.string esolver ^ -O  ^ File.bash-path resultname ^
^ File.bash-path lpname
        val res = Isabelle-System.bash-process (Bash.script command)
        in
          if not (Process-Result.ok res) then
            raise Fail (QSopt-exact returned with an error (return code  ^
              Int.toString (Process-Result.rc res) ^ ):\n ^ Process-Result.err res)
          else read-result (File.read resultname)
        end))

end

structure Rat-Linear-Program = Linear-Program(
struct

type T = 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 *ex-post*- or *SD*-Efficiency. For conditions arising from *SD*-Efficiency, a witness lottery must be specified (e. g. as computed by **derive-orbit-equations**).

derive-support-conditions to automatically find and prove all support conditions arising from *ex-post*- and *SD*-Efficiency

derive-strategyproofness-conditions to automatically derive all conditions arising from weak Strategy-Proofness and any manipulations between the given preference profiles. An optional maximum manipulation size can be specified.

All commands except **find-inefficient-supports** open a proof state and leave behind proof obligations for the user to discharge. This should always be possible using the Simplifier, possibly with a few additional rules, depending on the context.

lemma *disj-False-right*: $P \vee \mathit{False} \longleftrightarrow P$ **by** *simp*

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

lemma *less-or-eq-real*:
(*x*::*real*) < *y* ∨ *x* = *y* ⟷ *x* ≤ *y* *x* < *y* ∨ *y* = *x* ⟷ *x* ≤ *y* **by** *linarith*+

lemma *multiset-Diff-single-normalize*:
fixes *a c* **assumes** *a* ≠ *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* ≡ *prefs-from-table xss*
assumes *i* ∈ *agents* ∀ *i*∈*agents*. *y* ≽[*prefs-from-table xss i*] *x* ¬*y* ≼[*prefs-from-table xss i*] *x*
shows *ex-post-efficient-sds agents alts sds* ⟶ *pmf* (*sds R*) *x* = *0*
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 y*]) *simp-all*
qed

lemma *SD-inefficient-support-aux*:
assumes *R*: *prefs-from-table-wf agents alts xss R* ≡ *prefs-from-table xss*
assumes *as*: *as* ≠ [] *set as* ⊆ *alts distinct as A* = *set as*
assumes *ys*: ∀ *x*∈*set* (*map snd ys*). *0* ≤ *x sum-list* (*map snd ys*) = *1 set* (*map fst ys*) ⊆ *alts*
assumes *i*: *i* ∈ *agents*
assumes *SD1*: ∀ *i*∈*agents*. ∀ *x*∈*alts*.
sum-list (*map snd* (*filter* (λ*y*. *prefs-from-table xss i x* (*fst y*)) *ys*)) ≥
real (*length* (*filter* (*prefs-from-table xss i x*) *as*)) / *real* (*length as*)
assumes *SD2*: ∃ *x*∈*alts*. *sum-list* (*map snd* (*filter* (λ*y*. *prefs-from-table xss i x* (*fst y*)) *ys*)) >
real (*length* (*filter* (*prefs-from-table xss i x*) *as*)) / *real* (*length as*)
shows *sd-efficient-sds agents alts sds* ⟶ (∃ *x*∈*A*. *pmf* (*sds R*) *x* = *0*)
proof
assume *sd-efficient-sds agents alts sds*
from *R* **have** *wf*: *pref-profile-wf agents alts R*
by (*simp add*: *pref-profile-from-tableI'*)
then interpret *pref-profile-wf agents alts R* **.**

```
  interpret sd-efficient-sds agents alts sds by fact
  from ys have ys': pmf-of-list-wf ys by (intro pmf-of-list-wfI) auto

  {
    fix i x assume x ∈ alts i ∈ agents
    with ys' have lottery-prob (pmf-of-list ys) (preferred-alts (R i) x) =
      sum-list (map snd (filter (λy. prefs-from-table xss i x (fst y)) ys))
      by (subst measure-pmf-of-list) (simp-all add: preferred-alts-def R)
  } note A = this
  {
    fix i x assume x ∈ alts i ∈ agents
    with as have lottery-prob (pmf-of-set (set as)) (preferred-alts (R i) x) =
      real (card (set as ∩ preferred-alts (R i) x)) / real (card (set as))
      by (subst measure-pmf-of-set) simp-all
    also have set as ∩ preferred-alts (R i) x = set (filter (λy. R i x y) as)
      by (auto simp add: preferred-alts-def)
    also have card ... = length (filter (λy. R i x y) as)
      by (intro distinct-card distinct-filter assms)
    also have card (set as) = length as by (intro distinct-card assms)
    finally have lottery-prob (pmf-of-set (set as)) (preferred-alts (R i) x) =
      real (length (filter (prefs-from-table xss i x) as)) / real (length as)
      by (simp add: R)
  } note B = this

  from wf show ∃ x∈A. pmf (sds R) x = 0
  proof (rule SD-inefficient-support')
   from ys ys' show lottery1: pmf-of-list ys ∈ lotteries by (intro pmf-of-list-lottery)
    show i: i ∈ agents by fact
    from as have lottery2: pmf-of-set (set as) ∈ lotteries
      by (intro pmf-of-set-lottery) simp-all
    from i as SD2 lottery1 lottery2 show ¬SD (R i) (pmf-of-list ys) (pmf-of-set
A)
      by (subst preorder-on.SD-preorder[of alts]) (auto simp: A B not-le)
    from as SD1 lottery1 lottery2
      show ∀ i∈agents. SD (R i) (pmf-of-set A) (pmf-of-list ys)
        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 (⋃(set (xs#xss))) (pref-classes-lists xss)

fun pref-classes-lists-aux where
```

```
  pref-classes-lists-aux acc [] = {}
| 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) = (∪) (⋃(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 ∩ (⋃(set xss)) = {}
  shows  pref-classes-lists-aux acc xss =
           (insert acc ((λA. A ∪ acc) ‘ pref-classes-lists (rev xss)) − {acc ∪ ⋃(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)) = {} xs ≠ {}
    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) ((λx. x ∪ (acc ∪ xs)) ‘
                          pref-classes-lists (rev xss))) − {acc ∪ (xs ∪ ⋃(set xss))}
    by blast
  finally show ?case
    by (simp-all add: pref-classes-lists-append image-image Un-ac)
qed simp-all

lemma pref-classes-list-aux-hd-tl:
  assumes is-weak-ranking xss xss ≠ []
  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 xs 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 ‘
                        ((xs ∪ ⋃(set xss)) ∩ {x. x ∉ xs}) by blast
      thus ?thesis by (simp add: of-weak-ranking-Collect-ge-Cons′)
    qed
  qed simp-all
qed

lemma eval-pref-classes-of-weak-ranking:
  assumes ⋃(set xss) = alts is-weak-ranking xss alts ≠ {}
  shows   pref-classes alts (of-weak-ranking xss) = pref-classes-lists-aux (hd xss)
(tl xss)
proof −
  have pref-classes alts (of-weak-ranking xss) =
            preferred-alts (of-weak-ranking xss) ‘ (⋃(set (rev xss))) − {⋃(set xss)}
    by (simp add: pref-classes-def assms)
```

```
  also {
   have of-weak-ranking-Collect-ge (rev xss) ‘ (⋃(set (rev xss))) = pref-classes-lists
(rev xss)
     using assms by (intro pref-classes-of-weak-ranking-aux) simp-all
   also have of-weak-ranking-Collect-ge (rev xss) = preferred-alts (of-weak-ranking
xss)
     by (intro ext) (simp-all add: of-weak-ranking-Collect-ge-def preferred-alts-def)
    finally have preferred-alts (of-weak-ranking xss) ‘ (⋃(set (rev xss))) =
                  pref-classes-lists (rev xss) .
  }
  also from assms have pref-classes-lists (rev xss) − {⋃(set xss)} =
                       pref-classes-lists-aux (hd xss) (tl xss)
    by (intro pref-classes-list-aux-hd-tl [symmetric]) auto
  finally show ?thesis by simp
qed


context preorder-on
begin

lemma SD-iff-pref-classes:
  assumes p ∈ lotteries-on carrier q ∈ lotteries-on carrier
  shows   p ⪯[SD(le)] q ⟷
          (∀ A∈pref-classes carrier le. measure-pmf.prob p A ≤ measure-pmf.prob
q A)
proof safe
  fix A assume p ⪯[SD(le)] q A ∈ pref-classes carrier le
  thus measure-pmf.prob p A ≤ measure-pmf.prob q A
    by (auto simp: SD-preorder pref-classes-def)
next
 assume A: ∀ A∈pref-classes carrier le. measure-pmf.prob p A ≤ measure-pmf.prob
q A
  show p ⪯[SD(le)] q
  proof (rule SD-preorderI)
    fix x assume x: x ∈ carrier
    show measure-pmf.prob p (preferred-alts le x)
           ≤ measure-pmf.prob q (preferred-alts le x)
    proof (cases preferred-alts le x = carrier)
      case False
      with x have preferred-alts le x ∈ pref-classes carrier le
        unfolding pref-classes-def by (intro DiffI imageI) simp-all
      with A show ?thesis by simp
    next
      case True
     from assms have measure-pmf.prob p carrier = 1 measure-pmf.prob q carrier
= 1
       by (auto simp: measure-pmf.prob-eq-1 lotteries-on-def AE-measure-pmf-iff)
      with True show ?thesis by simp
    qed
```

qed (*insert assms*, *simp-all*)
qed

end

lemma (**in** *strategyproof-an-sds*) *strategyproof'*:
assumes *wf*: *is-pref-profile R total-preorder-on alts Ri'* **and** *i*: *i* ∈ *agents*
shows (∃ *A*∈*pref-classes alts* (*R i*). *lottery-prob* (*sds* (*R*(*i* := *Ri'*))) *A* <
lottery-prob (*sds R*) *A*) ∨
(∀ *A*∈*pref-classes alts* (*R i*). *lottery-prob* (*sds* (*R*(*i* := *Ri'*))) *A* =
lottery-prob (*sds R*) *A*)
proof −
from *wf*(*1*) **interpret** *R*: *pref-profile-wf agents alts R* **.**
from *i* **interpret** *total-preorder-on alts R i* **by** *simp*
from *assms* **have** ¬ *manipulable-profile R i Ri'* **by** (*intro strategyproof*)
moreover from *wf i* **have** *sds R* ∈ *lotteries sds* (*R*(*i* := *Ri'*)) ∈ *lotteries*
by (*simp-all add*: *sds-wf*)
ultimately show *?thesis*
by (*fastforce simp*: *manipulable-profile-def strongly-preferred-def*
SD-iff-pref-classes not-le not-less)
qed

lemma *pref-classes-lists-aux-finite*:
A ∈ *pref-classes-lists-aux acc xss* ⟹ *finite acc* ⟹ (⋀*A*. *A* ∈ *set xss* ⟹ *finite A*)
⟹ *finite A*
by (*induction acc xss rule*: *pref-classes-lists-aux.induct*) *auto*

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 *eq*: *R1*(*i* := *R2 j*) = *R2 the* (*map-of xss1 i*) = *xs*
pref-classes-lists-aux (*hd xs*) (*tl xs*) = *ps*
shows (∃ *A*∈*ps*. (∑ *x*∈*A*. *pmf* (*sds R2*) *x*) < (∑ *x*∈*A*. *pmf* (*sds R1*) *x*)) ∨
(∀ *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'*(*1*))

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*)] *prefs-from-table-wfD*(*5*,*6*)[*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.set-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: ∀ A∈ps. distinct A
  assumes eq:  mset (map snd xss1) − {#the (map-of xss1 i)#} + {#xs'#} =
                 mset (map (map ((‘) σ) ∘ snd) xss2)
               pref-classes-lists-aux (hd xs) (tl xs) = set ‘ ps
  shows   list-permutes ys alts ∧
          ((∃ A∈ps. (∑ x←A. pmf (sds R2) (σ' x)) < (∑ x←A. pmf (sds R1) x))
∨
             (∀ A∈ps. (∑ x←A. pmf (sds R2) (σ' x)) = (∑ x←A. pmf (sds R1)
x)))
          (is - ∧ ?th)
proof
  from perm have perm': σ permutes alts by (simp add: σ-def)
  from sds interpret strategyproof-an-sds agents alts sds .
  from wf(3) j have j ∈ set (map fst xss2) by (simp add: prefs-from-table-wf-def)
```

```
  with prefs-from-table-wfD(3)[OF wf(3)]
    have xs'-aux: the (map-of xss2 j) ∈ set (map snd xss2) by force
  with wf(3) have xs'-aux': is-finite-weak-ranking (the (map-of xss2 j))
    by (auto simp: prefs-from-table-wf-def)
  hence ∗: is-weak-ranking xs' unfolding xs'
    by (intro is-weak-ranking-map-inj permutes-inj-on[OF perm'])
       (auto simp add: is-finite-weak-ranking-def)
  moreover from ∗ xs'-aux' have is-finite-weak-ranking xs'
    by (auto simp: xs' is-finite-weak-ranking-def)
  moreover from prefs-from-table-wfD(5)[OF wf(3) xs'-aux]
    have ⋃(set xs') = alts unfolding xs'
    by (simp add: image-Union [symmetric] permutes-image[OF perm'])
  ultimately have wf-xs': is-weak-ranking xs' is-finite-weak-ranking xs' ⋃(set xs')
= alts
    by (simp-all add: is-finite-weak-ranking-def)
  from this wf j have wf': is-pref-profile R1 total-preorder-on alts Ri'
                        is-pref-profile R2 finite-total-preorder-on alts Ri'
   unfolding Ri'-def by (auto intro: pref-profile-from-tableI pref-profile-wf.prefs-wf'(1)
                                     total-preorder-of-weak-ranking)

 interpret R1: pref-profile-wf agents alts R1 by fact
 interpret R2: pref-profile-wf agents alts R2 by fact

  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(2)
    have xs ∈ set (map snd xss1) unfolding xs-def by force
  note xs = prefs-from-table-wfD(2)[OF wf(1)] prefs-from-table-wfD(5,6)[OF wf(1)
this]

  from wf i wf' wf-xs' xs eq
     have eq': anonymous-profile (R1(i := Ri')) = image-mset (map ((') σ))
(anonymous-profile R2)
    by (subst R1.anonymous-profile-update)
     (simp-all add: Ri'-def weak-ranking-of-weak-ranking mset-map multiset.map-comp
xs-def
          anonymise-prefs-from-table prefs-from-table-map-of)

  {
    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.set-sel)
    hence lottery-prob p A = (∑x∈A. pmf p x)
      by (rule measure-measure-pmf-finite)
  } note A = this

  from strategyproof'[OF wf'(1,2) i] eq' have
   (∃A∈pref-classes alts (R1 i). lottery-prob (sds (R1(i := Ri'))) A < lottery-prob
```

(*sds R1*) *A*) ∨
(∀ *A*∈*pref-classes alts* (*R1 i*). *lottery-prob* (*sds* (*R1*(*i* := *Ri'*))) *A* = *lottery-prob* (*sds R1*) *A*)
by *simp*
also from *eq' i* **have** *sds* (*R1*(*i* := *Ri'*)) = *map-pmf σ* (*sds R2*)
unfolding *σ-def* **by** (*intro sds-anonymous-neutral permutation-of-list-permutes perm wf'*
pref-profile-wf.wf-update eq)
also from *wf eq i* **have** *R1 i* = *of-weak-ranking xs*
by (*simp add*: *prefs-from-table-map-of xs-def*)
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 have (∃ *A*∈*ps*. (∑ *x*←*A*. *pmf* (*map-pmf σ* (*sds R2*)) *x*) < (∑ *x*←*A*. *pmf* (*sds R1*) *x*)) ∨
(∀ *A*∈*ps*. (∑ *x*←*A*. *pmf* (*map-pmf σ* (*sds R2*)) *x*) = (∑ *x*←*A*. *pmf* (*sds R1*) *x*))
using *distinct-ps*
by (*simp add*: *A eq sum.distinct-set-conv-list del*: *measure-map-pmf*)
also from *perm'* **have** *pmf* (*map-pmf σ* (*sds R2*)) = (λ*x*. *pmf* (*sds R2*) (*inv σ x*))
using *pmf-map-inj'*[*of σ* - *inv σ x* **for** *x*]
by (*simp add*: *fun-eq-iff permutes-inj permutes-inverses*)
also from *perm* **have** *inv σ* = *σ'* **unfolding** *σ-def σ'-def*
by (*rule inverse-permutation-of-list-correct* [*symmetric*])
finally show *?th* **.**
qed *fact+*

ML-file ‹*randomised-social-choice.ML*›
ML-file ‹*sds-automation.ML*›

end

References

[1] F. Brandl, F. Brandt, and C. Geist. Proving the incompatibility of Efficiency and Strategyproofness via SMT solving. *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI)*, 2016. Forthcoming.