Arrow and Gibbard-Satterthwaite

Tobias Nipkow

March 17, 2025

Abstract

This article formalizes two proofs of Arrow's impossibility theorem due to Geanakoplos and derives the Gibbard-Satterthwaite theorem as a corollary. One formalization is based on utility functions, the other one on strict partial orders.

For an article about these proofs see http://www.in.tum.de/~nipkow/pubs/arrow.pdf.

1 Arrow's Theorem for Utility Functions

theory *Arrow-Utility* **imports** *Complex-Main*
begin

This theory formalizes the first proof due to Geanakoplos [1]. In contrast to the standard model of preferences as linear orders, we model preferences as *utility functions* mapping each alternative to a real number. The type of alternatives and voters is assumed to be finite.

typedecl *alt*
typedecl *indi*

axiomatization where
alt3: $\exists\, a\ b\ c{::}alt.\ distinct[a,b,c]$ **and**
finite-alt: $finite(UNIV :: alt\ set)$ **and**

finite-indi: $finite(UNIV :: indi\ set)$

lemma *third-alt*: $a \neq b \Longrightarrow \exists\, c{::}alt.\ distinct[a,b,c]$
using *alt3* **by** *simp metis*

lemma *alt2*: $\exists\, b{::}alt.\ b \neq a$
using *alt3* **by** *simp metis*

type-synonym $pref = alt \Rightarrow real$
type-synonym $prof = indi \Rightarrow pref$

definition
top :: *pref* ⇒ *alt* ⇒ *bool* (**infixr** ‹<·› *60*) **where**
$p <\cdot\ b \ \equiv\ \forall a.\ a \neq b \longrightarrow p\ a < p\ b$

definition
bot :: *alt* ⇒ *pref* ⇒ *bool* (**infixr** ‹·<› *60*) **where**
$b \cdot\!< p \ \equiv\ \forall a.\ a \neq b \longrightarrow p\ b < p\ a$

definition
extreme :: *pref* ⇒ *alt* ⇒ *bool* **where**
$extreme\ p\ b \ \equiv\ b \cdot\!< p \lor p <\cdot\ b$

abbreviation
$Extreme\ P\ b == \forall i.\ extreme\ (P\ i)\ b$

lemma [*simp*]: $r <= s \Longrightarrow r < s+(1::real)$
by *arith*
lemma [*simp*]: $r < s \Longrightarrow r < s+(1::real)$
by *arith*
lemma [*simp*]: $r <= s \Longrightarrow \neg\ s+(1::real) < r$
by *arith*
lemma [*simp*]: $(r < s-(1::real)) = (r+1 < s)$
by *arith*
lemma [*simp*]: $(s-(1::real) < r) = (s < r+1)$
by *arith*

lemma *less-if-bot*[*simp*]: $[\![\ b \cdot\!< p;\ x \neq b\ ]\!] \Longrightarrow p\ b < p\ x$
by(*simp add*:*bot-def*)

lemma [*simp*]: $[\![\ p <\cdot\ b;\ x \neq b\ ]\!] \Longrightarrow p\ x < p\ b$
by(*simp add*:*top-def*)

lemma [*simp*]: **assumes** *top*: $p <\cdot\ b$ **shows** $\neg\ p\ b < p\ c$
proof (*cases*)
  assume $b = c$ **thus** *?thesis* **by** *simp*
next
  assume $b \neq c$
  with *top* **have** $p\ c < p\ b$ **by** (*simp add*:*eq-sym-conv*)
  thus *?thesis* **by** *simp*
qed

lemma *not-less-if-bot*[*simp*]:
  assumes *bot*: $b \cdot\!< p$ **shows** $\neg\ p\ c < p\ b$
proof (*cases*)
  assume $b = c$ **thus** *?thesis* **by** *simp*
next
  assume $b \neq c$
  with *bot* **have** $p\ b < p\ c$ **by** (*simp add*:*eq-sym-conv*)
  thus *?thesis* **by** *simp*

qed

lemma *top-impl-not-bot*[*simp*]: $p <\cdot\ b \Longrightarrow \neg\ b \cdot< p$
by(*unfold bot-def*, *simp add*:*alt2*)

lemma [*simp*]: $extreme\ p\ b \Longrightarrow (\neg\ p <\cdot\ b) = (b \cdot< p)$
apply(*unfold extreme-def*)
apply(*fastforce dest*:*top-impl-not-bot*)
done

lemma [*simp*]: $extreme\ p\ b \Longrightarrow (\neg\ b \cdot< p) = (p <\cdot\ b)$
apply(*unfold extreme-def*)
apply(*fastforce dest*:*top-impl-not-bot*)
done

Auxiliary construction to hide details of preference model.

definition
$mktop :: pref \Rightarrow alt \Rightarrow pref$ **where**
$mktop\ p\ b \equiv p(b := Max(range\ p) + 1)$

definition
$mkbot :: pref \Rightarrow alt \Rightarrow pref$ **where**
$mkbot\ p\ b \equiv p(b := Min(range\ p) - 1)$

definition
$between :: pref \Rightarrow alt \Rightarrow alt \Rightarrow alt \Rightarrow pref$ **where**
$between\ p\ a\ b\ c \equiv p(b := (p\ a + p\ c)/2)$

To make things simpler:

declare *between-def*[*simp*]

lemma [*simp*]: $a \neq b \Longrightarrow mktop\ p\ b\ a = p\ a$
by(*simp add*:*mktop-def*)

lemma [*simp*]: $a \neq b \Longrightarrow mkbot\ p\ b\ a = p\ a$
by(*simp add*:*mkbot-def*)

lemma [*simp*]: $a \neq b \Longrightarrow p\ a < mktop\ p\ b\ b$
by(*simp add*:*mktop-def finite-alt*)

lemma [*simp*]: $a \neq b \Longrightarrow mkbot\ p\ b\ b < p\ a$
by(*simp add*:*mkbot-def finite-alt*)

lemma [*simp*]: $mktop\ p\ b <\cdot\ b$
by(*simp add*:*mktop-def top-def finite-alt*)

lemma [*simp*]: $\neg\ b \cdot< mktop\ p\ b$
by(*simp add*:*mktop-def bot-def alt2 finite-alt*)

lemma [*simp*]: $a \neq b \Longrightarrow \neg\ P\ p\ a < mkbot\ (P\ p)\ b\ b$
proof (*simp add*:*mkbot-def finite-alt*)
  have $\neg\ P\ p\ a + 1 < P\ p\ a$ **by** *simp*
  thus $\exists x.\ \neg\ P\ p\ a + 1 < P\ p\ x$ **..**
qed

The proof starts here.

locale *arrow* =
fixes F :: *prof* ⇒ *pref*
assumes *unanimity*: $(\bigwedge i.\ P\ i\ a < P\ i\ b) \Longrightarrow F\ P\ a < F\ P\ b$
and *IIA*:
$(\bigwedge i.\ (P\ i\ a < P\ i\ b) = (P'\ i\ a < P'\ i\ b)) \Longrightarrow$
$(F\ P\ a < F\ P\ b) = (F\ P'\ a < F\ P'\ b)$
begin

lemmas $IIA' = IIA$[*THEN iffD1*]

definition
dictates :: *indi* ⇒ *alt* ⇒ *alt* ⇒ *bool* (‹- *dictates* - < -›) **where**
$(i\ dictates\ a < b) \ \equiv\ \forall P.\ P\ i\ a < P\ i\ b \longrightarrow F\ P\ a < F\ P\ b$
definition
dictates2 :: *indi* ⇒ *alt* ⇒ *alt* ⇒ *bool* (‹- *dictates* -,-›) **where**
$(i\ dictates\ a,b) \ \equiv\ (i\ dictates\ a < b) \wedge (i\ dictates\ b < a)$
definition
dictatesx:: *indi* ⇒ *alt* ⇒ *bool* (‹- *dictates*′*-except* -›) **where**
$(i\ dictates\text{-}except\ c) \ \equiv\ \forall a\ b.\ c \notin \{a,b\} \longrightarrow (i\ dictates\ a{<}b)$
definition
dictator :: *indi* ⇒ *bool* **where**
$dictator\ i \ \equiv\ \forall a\ b.\ (i\ dictates\ a{<}b)$

definition
pivotal :: *indi* ⇒ *alt* ⇒ *bool* **where**
$pivotal\ i\ b \equiv$
$\exists P.\ Extreme\ P\ b \ \wedge\ b \cdot\!< P\ i \ \wedge\ b \cdot\!< F\ P \ \wedge$
$F\ (P(i := mktop\ (P\ i)\ b)) <\!\cdot\ b$

lemma *all-top*[*simp*]: $\forall i.\ P\ i <\!\cdot\ b \Longrightarrow F\ P <\!\cdot\ b$
by (*unfold top-def*) (*simp add*: *unanimity*)

lemma *not-extreme*:
  assumes *nex*: $\neg\ extreme\ p\ b$
  shows $\exists a\ c.\ distinct[a,b,c] \wedge \neg\ p\ a < p\ b \wedge \neg\ p\ b < p\ c$
proof −
  obtain *a c* **where** *abc*: $a \neq b \wedge \neg\ p\ a < p\ b\ b \neq c \wedge \neg\ p\ b < p\ c$
    using *nex* **by** (*unfold extreme-def top-def bot-def*) *fastforce*
  show *?thesis*
  proof (*cases* $a = c$)
    assume $a \neq c$ **thus** *?thesis* **using** *abc* **by** *simp blast*
  next

```
    assume ac: a = c
    obtain d where d: distinct[a,b,d] using abc third-alt by blast
    show ?thesis
    proof (cases p b < p d)
      case False thus ?thesis using abc d by blast
    next
      case True
      hence db: ¬ p d < p b by arith
      from d have distinct[d,b,c] by(simp add:ac eq-sym-conv)
      thus ?thesis using abc db by blast
    qed
  qed
qed

lemma extremal:
  assumes extremes: Extreme P b shows extreme (F P) b
proof (rule ccontr)
  assume nec: ¬ extreme (F P) b
  hence ∃ a c. distinct[a,b,c] ∧ ¬ F P a < F P b ∧ ¬ F P b < F P c
    by(rule not-extreme)
  then obtain a c where d: distinct[a,b,c] and
    ab: ¬ F P a < F P b and bc: ¬ F P b < F P c by blast
  let ?P = λi. if P i <· b then between (P i) a c b
               else (P i)(c := P i a + 1)
  have ¬ F ?P a < F ?P b
    using extremes d by(simp add:IIA[of - - - P] ab)
  moreover have ¬ F ?P b < F ?P c
    using extremes d by(simp add:IIA[of - - - P] bc eq-sym-conv)
  moreover have F ?P a < F ?P c by(rule unanimity)(insert d, simp)
  ultimately show False by arith
qed


lemma pivotal-ind: assumes fin: finite D
  shows ⋀P. ⟦ D = {i. b ·< P i}; Extreme P b; b ·< F P ⟧
  ⟹ ∃ i. pivotal i b (is ⋀P. ?D D P ⟹ ?E P ⟹ ?B P ⟹ -)
using fin
proof (induct)
  case (empty P)
  from empty(1,2) have ∀ i. P i <· b by simp
  hence F P <· b by simp
  hence False using empty by(blast dest:top-impl-not-bot)
  thus ?case ..
next
  fix D i P
  assume IH: ⋀P. ?D D P ⟹ ?E P ⟹ ?B P ⟹ ∃ i. pivotal i b
    and ?E P and ?B P and insert: insert i D = {i. b ·< P i} and i ∉ D
  from insert have b ·< P i by blast
  let ?P = P(i := mktop (P i) b)
```

```
  show ∃ i. pivotal i b
  proof (cases F ?P <· b)
    case True
    have pivotal i b
    proof −
      from ‹?E P› ‹?B P› ‹b ·< P i› True
      show ?thesis by(unfold pivotal-def, blast)
    qed
    thus ?thesis ..
  next
    case False
    have D = {i. b ·< ?P i}
      by (rule set-eqI) (simp add:‹i ∉ D›, insert insert, blast)
    moreover have Extreme ?P b
      using ‹?E P› by (simp add:extreme-def)
    moreover have b ·< F ?P
      using extremal[OF ‹Extreme ?P b›] False by(simp del:fun-upd-apply)
    ultimately show ?thesis by(rule IH)
  qed
qed

lemma pivotal-exists: ∃ i. pivotal i b
proof −
  let ?P = (λ- a. if a=b then 0 else 1)::prof
  have Extreme ?P b by(simp add:extreme-def bot-def)
  moreover have b ·< F ?P
    by(simp add:bot-def unanimity del: less-if-bot not-less-if-bot)
  ultimately show ∃ i. pivotal i b
    by (rule pivotal-ind[OF finite-subset[OF subset-UNIV finite-indi] refl])
qed


lemma pivotal-xdictates: assumes pivo: pivotal i b
  shows i dictates-except b
proof −
  have ⋀a c. ⟦ a ≠ b; b ≠ c ⟧ ⟹ i dictates a < c
  proof (unfold dictates-def, intro allI impI)
    fix a c and P::prof
    assume abc: a ≠ b b ≠ c and
           ac: P i a < P i c
    show F P a < F P c
    proof −
      obtain P1 P2 where
        Extreme P1 b and b ·< F P1 and b ·< P1 i and F P2 <· b and
        [simp]: P2 = P1(i := mktop (P1 i) b)
        using pivo by (unfold pivotal-def) fast
      let ?P = λj. if j=i then between (P j) a b c
                     else if P1 j <· b then mktop (P j) b else mkbot (P j) b
      have eq: (F P a < F P c) = (F ?P a < F ?P c)
```

```
      using abc by − (rule IIA, auto)
    have F ?P a < F ?P b
    proof (rule IIA′)
      fix j show (P2 j a < P2 j b) = (?P j a < ?P j b)
        using ‹Extreme P1 b› by(simp add: ac)
    next
      show F P2 a < F P2 b
        using ‹F P2 <· b› abc by(simp add: eq-sym-conv)
    qed
    also have ... < F ?P c
    proof (rule IIA′)
      fix j show (P1 j b < P1 j c) = (?P j b < ?P j c)
        using ‹Extreme P1 b› ‹b ·< P1 i› by(simp add: ac)
    next
      show F P1 b < F P1 c
        using ‹b ·< F P1› abc by(simp add: eq-sym-conv)
    qed
    finally show ?thesis by(simp add:eq)
  qed
 qed
 thus ?thesis by(unfold dictatesx-def) fast
qed

lemma pivotal-is-dictator:
 assumes pivo: pivotal i b and ab: a ≠ b and d: j dictates a,b
 shows i = j
proof (rule ccontr)
 assume pd: i ≠ j
 obtain P1 P2 where Extreme P1 b and b ·< F P1 and F P2 <· b and
   P2: P2 = P1(i := mktop (P1 i) b)
   using pivo by (unfold pivotal-def) fast
 have ~(P1 j a < P1 j b) (is ~ ?ab)
 proof
   assume ?ab
   hence F P1 a < F P1 b using d by(simp add: dictates-def dictates2-def)
   with ‹b ·< F P1› show False by simp
 qed
 hence P1 j b < P1 j a using ‹Extreme P1 b›[THEN spec, of j] ab
   unfolding extreme-def top-def bot-def by metis
 hence P2 j b < P2 j a using pd by (simp add:P2)
 hence F P2 b < F P2 a using d by(simp add: dictates-def dictates2-def)
 with ‹F P2 <· b› show False by simp
qed


theorem dictator: ∃ i. dictator i
proof−
 from pivotal-exists[of b] obtain i where pivo: pivotal i b ..
 { fix a assume neq: a ≠ b have i dictates a,b
```

```
    proof –
      obtain c where dist: distinct[a,b,c]
        using neq third-alt by blast
      obtain j where pivotal j c using pivotal-exists by fast
      hence j dictates-except c by(rule pivotal-xdictates)
      hence b: j dictates a,b
        using dist by(simp add:dictatesx-def dictates2-def eq-sym-conv)
      with pivo neq have i = j by(rule pivotal-is-dictator)
      thus ?thesis using b by simp
    qed
  }
  with pivotal-xdictates[OF pivo] have dictator i
    by(simp add: dictates-def dictatesx-def dictates2-def dictator-def)
      (metis less-le)
  thus ?thesis ..
qed

end

end
```

2 Arrow's Theorem for Strict Linear Orders

```
theory Arrow-Order imports Main HOL−Library.FuncSet
begin
```

This theory formalizes the third proof due to Geanakoplos [1]. Preferences are modeled as strict linear orderings. The set of alternatives need not be finite.

Individuals are assumed to be finite but are not a priori identified with an initial segment of the naturals. In retrospect this generality appears gratuitous and complicates some of the low-level reasoning where we use a bijection with such an initial segment.

```
typedecl alt
typedecl indi

abbreviation I == (UNIV::indi set)

axiomatization where
  alt3: ∃ a b c::alt. distinct[a,b,c] and
  finite-indi: finite I

abbreviation N == card I

lemma third-alt: a ≠ b ⟹ ∃ c::alt. distinct[a,b,c]
using alt3 by simp metis

lemma alt2: ∃ b::alt. b ≠ a
```

using *alt3* **by** *simp metis*

type-synonym *pref* = (*alt* ∗ *alt*)*set*

definition *Lin* == {*L*::*pref*. *strict-linear-order L*}

lemmas *slo-defs* = *Lin-def strict-linear-order-on-def total-on-def irrefl-def*

lemma *notin-Lin-iff*: $L : Lin \Longrightarrow x{\neq}y \Longrightarrow (x,y) \notin L \longleftrightarrow (y,x) : L$
apply(*auto simp add*: *slo-defs*)
apply(*metis trans-def*)
done

lemma *converse-in-Lin*[*simp*]: $L^{-1} : Lin \longleftrightarrow L : Lin$
apply (*simp add*: *slo-defs*)
apply *blast*
done

lemma *Lin-irrefl*: $L{:}Lin \Longrightarrow (a,b){:}L \Longrightarrow (b,a){:}L \Longrightarrow False$
by(*simp add*:*slo-defs*)(*metis trans-def*)

corollary *linear-alt*: ∃ *L*::*pref*. *L* : *Lin*
using *well-order-on*[**where** ′*a* = *alt*, *of UNIV*]
apply (*auto simp*:*well-order-on-def Lin-def*)
apply (*metis strict-linear-order-on-diff-Id*)
done

abbreviation
rem :: *pref* ⇒ *alt* ⇒ *pref* **where**
rem L a ≡ {(*x*,*y*). (*x*,*y*) ∈ *L* ∧ *x*≠*a* ∧ *y*≠*a*}
definition
mktop :: *pref* ⇒ *alt* ⇒ *pref* **where**
mktop L b ≡ *rem L b* ∪ {(*x*,*b*)|*x*. *x*≠*b*}
definition
mkbot :: *pref* ⇒ *alt* ⇒ *pref* **where**
mkbot L b ≡ *rem L b* ∪ {(*b*,*y*)|*y*. *y*≠*b*}
definition
below :: *pref* ⇒ *alt* ⇒ *alt* ⇒ *pref* **where**
below L a b ≡ *rem L a* ∪
{(*a*,*b*)} ∪ {(*x*,*a*)|*x*. (*x*,*b*) : *L* ∧ *x*≠*a*} ∪ {(*a*,*y*)|*y*. (*b*,*y*) : *L* ∧ *y*≠*a*}
definition
above :: *pref* ⇒ *alt* ⇒ *alt* ⇒ *pref* **where**
above L a b ≡ *rem L b* ∪
{(*a*,*b*)} ∪ {(*x*,*b*)|*x*. (*x*,*a*) : *L* ∧ *x*≠*b*} ∪ {(*b*,*y*)|*y*. (*a*,*y*) : *L* ∧ *y*≠*b*}

lemma *in-mktop*: (*x*,*y*) ∈ *mktop L z* ⟷ *x*≠*z* ∧ (*if y*=*z then x*≠*y else* (*x*,*y*) ∈ *L*)
by(*auto simp*:*mktop-def*)

lemma *in-mkbot*: (*x*,*y*) ∈ *mkbot L z* ⟷ *y*≠*z* ∧ (*if x*=*z then x*≠*y else* (*x*,*y*) ∈ *L*)

```
by(auto simp:mkbot-def)

lemma in-above: a≠b ⟹ L:Lin ⟹
  (x,y) : above L a b ⟷ x≠y ∧
  (if x=b then (a,y) : L else
   if y=b then x=a | (x,a) : L else (x,y) : L)
by(auto simp:above-def slo-defs)

lemma in-below: a≠b ⟹ L:Lin ⟹
  (x,y) : below L a b ⟷ x≠y ∧
  (if y=a then (x,b) : L else
   if x=a then y=b | (b,y) : L else (x,y) : L)
by(auto simp:below-def slo-defs)

declare [[simp-depth-limit = 2]]

lemma mktop-Lin: L : Lin ⟹ mktop L x : Lin
by(auto simp add:slo-defs mktop-def trans-def)
lemma mkbot-Lin: L : Lin ⟹ mkbot L x : Lin
by(auto simp add:slo-defs trans-def mkbot-def)

lemma below-Lin: x≠y ⟹ L : Lin ⟹ below L x y : Lin
unfolding slo-defs below-def trans-def
apply(simp)
apply blast
done

lemma above-Lin: x≠y ⟹ L : Lin ⟹ above L x y : Lin
unfolding slo-defs above-def trans-def
apply(simp)
apply blast
done

declare [[simp-depth-limit = 50]]

abbreviation lessLin :: alt ⇒ pref ⇒ alt ⇒ bool (‹(- <- -)› [51, 51] 50)
where a <L b == (a,b) : L

definition Prof = I → Lin

abbreviation SWF == Prof → Lin

definition unanimity F == ∀ P∈Prof.∀ a b. (∀ i. a <P i b) ⟶  a <F P b

definition IIA F == ∀ P∈Prof.∀ P'∈Prof.∀ a b.
  (∀ i. a <P i b ⟷ a <P' i b) ⟶ (a <F P b ⟷ a <F P' b)

definition dictator F i == ∀ P∈Prof. F P = P i
```

```
lemma dictatorI: F : SWF ⟹
  ∀P∈Prof. ∀a b. a ≠ b ⟶ (a,b) : P i ⟶ (a,b) : F P ⟹ dictator F i
apply(simp add:dictator-def Prof-def Pi-def Lin-def strict-linear-order-on-def)
apply safe
 apply(erule-tac x=P in allE)
 apply(erule-tac x=P in allE)
 apply(simp add:total-on-def irrefl-def)
 apply (metis trans-def)
apply (metis irrefl-def)
done

lemma const-Lin-Prof: L:Lin ⟹ (%p. L) : Prof
by(simp add:Prof-def Pi-def)

lemma complete-Lin: assumes a≠b shows ∃L∈Lin. (a,b) : L
proof −
  from linear-alt obtain R where R:Lin by auto
  thus ?thesis by (metis assms in-mkbot mkbot-Lin)
qed

declare Let-def[simp]

theorem Arrow: assumes F : SWF and u: unanimity F and IIA F
shows ∃i. dictator F i
proof −
  { fix a b a′ b′ and P P′
    assume d1: a≠b a′≠b′ and d2: a≠b′ b≠a′ and
      P : Prof P′ : Prof and 1: ∀i. (a,b) : P i ⟷ (a′,b′) : P′ i
    assume (a,b) : F P
    define Q where
      Q i = (let L = (if a=a′ then P i else below (P i) a′ a)
             in if b=b′ then L else above L b b′) for i
    have Q : Prof using ‹P : Prof›
      by(simp add:Q-def Prof-def Pi-def above-Lin below-Lin)
    hence F Q : Lin using ‹F : SWF› by(simp add:Pi-def)
    have ∀i. (a,b) : P i ⟷ (a,b) : Q i using d1 d2 ‹P : Prof›
      by(simp add:in-above in-below Q-def Prof-def Pi-def below-Lin)
    hence (a,b) : F Q using ‹(a,b) : F P› ‹IIA F› ‹P:Prof› ‹Q : Prof›
      unfolding IIA-def by blast
    moreover
    { assume a≠a′
      hence !!i. (a′,a) : Q i using d1 d2 ‹P : Prof›
        by(simp add:in-above in-below Q-def Prof-def Pi-def below-Lin)
      hence (a′,a) : F Q using u ‹Q : Prof› by(simp add:unanimity-def)
    } moreover
    { assume b≠b′
      hence !!i. (b,b′) : Q i using d1 d2 ‹P : Prof›
        by(simp add:in-above in-below Q-def Prof-def Pi-def below-Lin)
      hence (b,b′) : F Q using u ‹Q : Prof› by(simp add:unanimity-def)
```

```
  }
  ultimately have (a′,b′) : F Q using ‹F Q : Lin›
    unfolding slo-defs trans-def
    by safe metis
  moreover
  have ∀ i. (a′,b′) : Q i ⟷ (a′,b′) : P′ i using d1 d2 ‹P : Prof› 1
    by(simp add:Q-def in-below in-above Prof-def Pi-def below-Lin) blast
  ultimately have (a′,b′) : F P′
    using ‹IIA F› ‹P′ : Prof› ‹Q : Prof› unfolding IIA-def by blast
} note 1 = this
{ fix a b a′ b′ and P P′
  assume a≠b a′≠b′ a≠b′ b≠a′ P : Prof P′ : Prof
       ∀ i. (a,b) : P i ⟷ (a′,b′) : P′ i
  hence (a,b) : F P ⟷ (a′,b′) : F P′ using 1 by blast
} note 2 = this
{ fix a b P P′
  assume a≠b P : Prof P′ : Prof and
    iff: ∀ i. (a,b) : P i ⟷ (b,a) : P′ i
  from ‹a≠b› obtain c where dist: distinct[a,b,c] using third-alt by metis
  let ?Q = %p. below (P p) c b
  let ?R = %p. below (?Q p) b a
  let ?S = %p. below (?R p) a c
  have ?Q : Prof using ‹P : Prof› dist
    by(auto simp add:Prof-def Pi-def below-Lin)
  hence ?R : Prof using dist by(auto simp add:Prof-def Pi-def below-Lin)
  hence ?S : Prof using dist by(auto simp add:Prof-def Pi-def below-Lin)
  have ∀ i. (a,b) : P i ⟷ (a,c) : ?Q i using ‹P : Prof› dist
    by(auto simp add:in-below Prof-def Pi-def)
  hence ab: (a,b) : F P ⟷ (a,c) : F ?Q
    using 2 ‹P : Prof› ‹?Q : Prof› dist[simplified] by (blast)
  have ∀ i. (a,c) : ?Q i ⟷ (b,c) : ?R i using ‹P : Prof› dist
    by(auto simp add:in-below Prof-def Pi-def below-Lin)
  hence ac: (a,c) : F ?Q ⟷ (b,c) : F ?R
    using 2 ‹?Q : Prof› ‹?R : Prof› dist[simplified] by (blast)
  have ∀ i. (b,c) : ?R i ⟷ (b,a) : ?S i using ‹P : Prof› dist
    by(auto simp add:in-below Prof-def Pi-def below-Lin)
  hence bc: (b,c) : F ?R ⟷ (b,a) : F ?S
    using ‹?R : Prof› ‹?S : Prof› dist[simplified] 2
    apply −
    apply(rule 2)
    by fast+
  have ∀ i. (b,a) : ?S i ⟷ (a,b) : P i using ‹P : Prof› dist
    by(auto simp add:in-below Prof-def Pi-def below-Lin)
  hence ∀ i. (b,a) : ?S i ⟷ (b,a) : P′ i using iff by blast
  hence ba: (b,a) : F ?S ⟷ (b,a) : F P′
    using ‹IIA F› ‹P′ : Prof› ‹?S : Prof› unfolding IIA-def by fast
  from ab ac bc ba have (a,b) : F P ⟷ (b,a) : F P′ by simp
} note 3 = this
{ fix a b c P P′
```

```
  assume A: a≠b b≠c a≠c P : Prof P′ : Prof and
    iff: ∀ i. (a,b) : P i ⟷ (b,c) : P′ i
  hence ∀ i. (b,a) : (converse o P)i ⟷ (b,c) : P′ i by simp
  moreover have cP: converse o P : Prof
    using ‹P:Prof› by(simp add:Prof-def Pi-def)
  ultimately have (b,a) : F(converse o P) ⟷ (b,c) : F P′ using A 2
    by metis
  moreover have (a,b) : F P ⟷ (b,a) : F(converse o P)
    by (rule 3[OF ‹a≠b› ‹P:Prof› cP]) simp
  ultimately have (a,b) : F P ⟷ (b,c) : F P′ by blast
} note 4 = this
{ fix a b a′ b′ :: alt and P P′
  assume A: a≠b a′≠b′ P : Prof P′ : Prof
    ∀ i. (a,b) : P i ⟷ (a′,b′) : P′ i
  have (a,b) : F P ⟷ (a′,b′) : F P′
  proof−
    { assume a≠b′ & b≠a′ hence ?thesis using 2 A by blast }
    moreover
    { assume a=b′ & b≠a′ hence ?thesis using 4 A by blast }
    moreover
    { assume a≠b′ & b=a′ hence ?thesis using 4 A by blast }
    moreover
    { assume a=b′ & b=a′ hence ?thesis using 3 A by blast }
    ultimately show ?thesis by blast
  qed
} note pairwise-neutrality = this
obtain h :: indi ⇒ nat where
  injh: inj h and surjh: h ‘ I = {0..<N}
  by(metis ex-bij-betw-finite-nat[OF finite-indi] bij-betw-def)
obtain a b :: alt where a ≠ b using alt3 by auto
obtain Lab where (a,b) : Lab Lab:Lin using ‹a≠b› by (metis complete-Lin)
hence (b,a) ∉ Lab by(simp add:slo-defs trans-def) metis
obtain Lba where (b,a) : Lba Lba:Lin using ‹a≠b› by (metis complete-Lin)
hence (a,b) ∉ Lba by(simp add:slo-defs trans-def) metis
let ?Pi = %n. %i. if h i < n then Lab else Lba
have PiProf: !!n. ?Pi n : Prof using ‹Lab:Lin› ‹Lba:Lin›
  unfolding Prof-def Pi-def by simp
have ∃ n<N. (∀ m≤n. (b,a) : F(?Pi m)) & (a,b) : F(?Pi(n+1))
proof −
  have 0: !!n. F(?Pi n) : Lin using ‹F : SWF› PiProf by(simp add:Pi-def)
  have F(%i. Lba):Lin using ‹F:SWF› ‹Lba:Lin› by(simp add:Prof-def Pi-def)
  hence 1: (a,b) ∉ F(?Pi 0) using u ‹(a,b) ∉ Lba› ‹Lba:Lin› ‹Lba:Lin› ‹a≠b›
    by(simp add:unanimity-def notin-Lin-iff const-Lin-Prof)
  have ?Pi N = (%p. Lab) using surjh [THEN equalityD1]
    by(auto simp: subset-eq)
  moreover
  have F(%i. Lab):Lin using ‹F:SWF› ‹Lab:Lin› by(simp add:Prof-def Pi-def)
  ultimately have 2: (a,b) ∈ F(?Pi N) using u ‹(a,b) : Lab› ‹Lab:Lin›
    by(simp add:unanimity-def const-Lin-Prof)
```

```
    with ex-least-nat-less[of %n. (a,b) : F(?Pi n)] 1 2 notin-Lin-iff[OF 0 ‹a≠b›]
    show ?thesis by simp
  qed
  then obtain n where n: n<N ∀m≤n. (b,a) : F(?Pi m) (a,b) : F(?Pi(n+1))
    by blast
  have dictator F (inv h n)
  proof (rule dictatorI [OF ‹F : SWF›], auto)
    fix P c d assume P ∈ Prof c≠d (c,d) ∈ P(inv h n)
    then obtain e where dist: distinct[c,d,e] using third-alt by metis
    let ?W = %i. if h i < n then mktop (P i) e else
                 if h i = n then above (P i) c e else mkbot (P i) e
    have ?W : Prof using ‹P : Prof› dist
      by(simp add:Pi-def Prof-def mkbot-Lin mktop-Lin above-Lin)
    have ∀i. (c,d) : P i ⟷ (c,d) : ?W i using dist ‹P : Prof›
      by(auto simp: in-above in-mkbot in-mktop Prof-def Pi-def)
    hence PW: (c,d) : F P ⟷ (c,d) : F ?W
      using ‹IIA F›[unfolded IIA-def] ‹P:Prof› ‹?W:Prof› by fast
    have ∀i. (c,e) : ?W i ⟷ (a,b) : ?Pi (n+1) i using dist ‹P : Prof›
      by (auto simp: ‹(a,b):Lab› ‹(a,b)∉Lba›
        in-mkbot in-mktop in-above Prof-def Pi-def)
    hence (c,e) : F ?W ⟷ (a,b) : F(?Pi(n+1))
      using pairwise-neutrality[of c e a b ?W ?Pi(n+1)]
        ‹a≠b› dist ‹?W : Prof› PiProf by simp
    hence (c,e) : F ?W using n(3) by blast
    have ∀i. (e,d) : ?W i ⟷ (b,a) : ?Pi n i
      using dist ‹P : Prof› ‹(c,d) ∈ P(inv h n)› ‹inj h›
      by(auto simp: ‹(b,a):Lba› ‹(b,a)∉Lab›
        in-mkbot in-mktop in-above Prof-def Pi-def)
    hence (e,d) : F ?W ⟷ (b,a) : F(?Pi n)
      using pairwise-neutrality[of e d b a ?W ?Pi n]
        ‹a≠b› dist ‹?W : Prof› PiProf by simp blast
    hence (e,d) : F ?W using n(2) by auto
    with ‹(c,e) : F ?W› ‹?W : Prof› ‹F:SWF›
    have (c,d) ∈ F ?W unfolding Pi-def slo-defs trans-def by blast
    thus (c,d) ∈ F P using PW by blast
  qed
  thus ?thesis ..
qed

end
```

3 The Gibbard-Satterthwaite Theorem

```
theory GS imports Arrow-Order
begin
```

The Gibbard-Satterthwaite theorem as a corollary to Arrow's theorem. The proof follows Nisan [2].

```
definition manipulable f == ∃P∈Prof. ∃i. ∃L∈Lin. (f P, f(P(i:=L))) : P i
```

```
definition dict f i == ∀ P∈Prof.∀ a. a ≠ f P ⟶ (a,f P) : P i

definition
 Top :: alt set ⇒ pref ⇒ pref where
Top S L ≡ {(a,b). (a,b) ∈ L ∧ (a ∈ S ∧ b ∈ S ∨ a ∉ S ∧ b ∉ S)} ∪
             {(a,b). a ∉ S ∧ b ∈ S}

lemma Top-in-Lin: L:Lin ⟹ Top S L : Lin
apply(simp add:Top-def slo-defs Sigma-def)
unfolding trans-def
apply blast
done

lemma Top-in-Prof: P:Prof ⟹ Top S o P : Prof
by(simp add:Prof-def Pi-def Top-in-Lin)

lemma not-manipulable: ¬ manipulable f ⟷
 (∀ P∈Prof.∀ i.∀ L∈Lin. f P ≠ f(P(i:=L)) ⟶
   (f(P(i := L)), f P) : P i ∧ (f P, f(P(i := L))) : L) (is ?A = ?B)
proof
  assume ?A
  show ?B
  proof(clarsimp)
    fix P i L assume 0: P ∈ Prof L ∈ Lin f P ≠ f (P(i := L))
    moreover hence 1: P i: Lin P(i:=L): Prof by(simp add:Prof-def Pi-def)+
    ultimately have (f (P(i := L)), f P) ∈ P i (is ?L)
      using ‹?A› unfolding manipulable-def by (metis notin-Lin-iff)
    moreover have (f P, f (P(i := L))) ∈ L (is ?R)
      using 0 1 fun-upd-upd[of P] fun-upd-triv[of P] fun-upd-same[of P]
      using ‹?A› unfolding manipulable-def by (metis notin-Lin-iff)
    ultimately show ?L ∧ ?R ..
  qed
next
  assume ?B
  show ?A
  proof(clarsimp simp:manipulable-def)
    fix P i L assume P ∈ Prof L ∈ Lin (f P, f (P(i := L))) ∈ P i
    moreover hence P i: Lin by(simp add:Prof-def Pi-def)
    ultimately show False
      using ‹?B› by(metis Lin-irrefl)
  qed
qed

definition swf(f) ≡ λP. {(a,b). a≠b ∧ f(Top {a,b} o P) = b}

locale GS =
fixes f
assumes not-manip: ¬ manipulable f
```

```
and onto: f ‘ Prof = UNIV
begin

lemma nonmanip:
  P:Prof ⟹ L:Lin ⟹ f(P(i := L)) ≠ f P ⟹
  (f(P(i := L)), f P) : P i ∧ (f P, f(P(i := L))) : L
using not-manip by(metis not-manipulable)

lemma mono:
assumes P∈Prof P′∈Prof ∀ i a. (a, f P) : P i ⟶ (a, f P) : P′ i
shows f P′ = f P
proof−
  obtain h :: indi ⇒ nat where
    injh: inj h and surjh: h ‘ I = {0..<N}
    by(metis ex-bij-betw-finite-nat[OF finite-indi] bij-betw-def)
  let ?M = %n i. if h i < n then P′ i else P i
  have N: !!i. h i < N using surjh by auto
  have MProf: !!n. ?M n : Prof and P′Lin: !!i. P′ i : Lin
    using ‹P:Prof› ‹P′:Prof› by(simp add:Prof-def Pi-def)+
  { fix n have n<=N ⟹ f(?M n) = f P
    proof(induct n)
      case 0 show ?case by simp
    next
      case (Suc n)
      let ?up = (?M n)(inv h n := P′ (inv h n))
      have 1: ?M(Suc n) = ?up using surjh Suc(2)
        by(simp (no-asm-simp) add:fun-eq-iff f-inv-into-f)
          (metis injh inv-f-f less-antisym)
      show ?case
      proof(rule ccontr)
        assume ¬ ?case
        with ‹?M(Suc n) = ?up› Suc have 0: f ?up ≠ f(?M n) by simp
        from nonmanip[OF MProf P′Lin 0] assms(3) show False
          using N surjh Suc Lin-irrefl[OF P′Lin]
          by(fastforce simp: f-inv-into-f)
      qed
    qed
  }
  from this[of N] N show f P′ = f P by simp
qed

lemma una-Top: assumes P:Prof S ≠ {} shows f(Top S o P) : S
proof −
  obtain h :: indi ⇒ nat where
    injh: inj h and surjh: h ‘ I = {0..<N}
    by(metis ex-bij-betw-finite-nat[OF finite-indi] bij-betw-def)
  from assms obtain a where a:S by blast
  from onto obtain Pa where Pa:Prof f Pa = a
    by(metis inv-into-into UNIV-I f-inv-into-f)
```

```
  let ?M = %n i. if h i < n then Top S (P i) else Pa i
  have N: !!i. h i < N using surjh by auto
  have MProf: !!n. ?M n : Prof using ‹P:Prof› ‹Pa:Prof›
    by(simp add:Prof-def Pi-def Top-in-Lin mktop-Lin)
  { fix n have n<=N ⟹ f(?M n) : S
    proof(induct n)
      case 0 thus ?case using ‹f Pa = a› ‹a:S› by simp
    next
      case (Suc n)
      show ?case
      proof cases
        assume f(?M n) = f(?M(Suc n))
        thus ?case using Suc by simp
      next
        let ?up = (?M n)(inv h n := Top S (P(inv h n)))
        assume f(?M n) ≠ f(?M(Suc n))
        also have eq: ?M(Suc n) = ?up using surjh Suc
          by(simp (no-asm-simp) add:fun-eq-iff f-inv-into-f)
            (metis injh inv-f-eq less-antisym)
        finally have n: f(?M n) ≠ f(?up) .
        with nonmanip[OF MProf Top-in-Lin n[symmetric]] Suc eq ‹P:Prof›
        show ?case by (simp add:Top-def Prof-def Pi-def)
      qed
    qed
  }
  from this[of N] N show ?thesis by(simp add:comp-def)
qed

lemma SWF-swf: swf f : SWF
proof (rule Pi-I)
  fix P assume P: Prof
  show swf f P : Lin
  proof(unfold Lin-def strict-linear-order-on-def, auto)
    show total(swf f P)
    proof(simp add: total-on-def, intro allI impI)
      fix a b :: alt assume a≠b
      thus (a,b) ∈ swf f P ∨ (b,a) ∈ swf f P
        unfolding swf-def using una-Top[of P {a,b}] ‹P:Prof›
        by simp(metis insert-commute)
    qed
    show irrefl(swf f P) by(simp add: irrefl-def swf-def)
    show trans(swf f P)
    proof (clarsimp simp:trans-def swf-def insert-commute)
      fix a b c assume a≠b b≠c f(Top{a,b} ∘ P) = b f(Top{b,c} ∘ P) = c
      hence a≠c by(auto simp: insert-commute)
      note 3 = Top-in-Prof[OF ‹P:Prof›, of {a,b,c}]
      { assume f (Top {a, b, c} ∘ P) = a
        hence f(Top{a,b} ∘ P) = a
          using mono[OF 3 Top-in-Prof[OF ‹P:Prof›], of {a,b}]
```

```
        by(auto simp:Top-def)
      with ‹f(Top{a,b} ∘ P) = b› ‹a≠b› have False by simp
    } moreover
    { assume f (Top {a, b, c} ∘ P) = b
      hence f(Top{b,c} ∘ P) = b
        using mono[OF 3 Top-in-Prof[OF ‹P:Prof›], of {b,c}]
        by(auto simp:Top-def)
      with ‹f(Top{b,c} ∘ P) = c› ‹b≠c› have False by simp
    }
    ultimately have f (Top {a, b, c} ∘ P) = c
      using una-Top[OF ‹P:Prof›, of {a,b,c}, simplified] by blast
    hence f(Top{a,c} ∘ P) = c (is ?R)
      using mono[OF 3 Top-in-Prof[OF ‹P:Prof›], of {a,c}]
      by (auto simp:Top-def)
    thus a≠c ∧ ?R using ‹a≠c› by blast
  qed
 qed
qed

lemma Top-top: L:Lin ⟹ (!!a. a≠b ⟹ (a,b) : L) ⟹ Top {b} L = L
apply(auto simp:Top-def slo-defs)
apply (metis trans-def)
apply (metis trans-def)
done

lemma una-swf: unanimity(swf f)
proof(clarsimp simp:swf-def unanimity-def)
 fix P a b
 assume P:Prof and abP: ∀ i. (a, b) ∈ P i
 hence a ≠ b by(fastforce simp:Prof-def Pi-def slo-defs)
 let ?abP = Top {a,b} ∘ P
 have ?abP : Prof using ‹P:Prof› by(simp add:Prof-def Pi-def Top-in-Lin)
 have top: !!i c. b≠c ⟹ (c,b) : Top {a,b} (P i)
  using abP by(auto simp:Top-def)
 have Top {b} o ?abP = ?abP using ‹P:Prof›
  by (simp add:fun-eq-iff top Top-top Prof-def Pi-def Top-in-Lin)
 moreover have f(Top {b} o ?abP) = b
  by (metis una-Top[OF ‹?abP : Prof›] empty-not-insert singletonE)
 ultimately have f ?abP = b by simp
 thus a≠b ∧ f ?abP = b using ‹a≠b› by blast
qed

lemma IIA-swf: IIA(swf f)
proof(clarsimp simp: IIA-def)
 fix P P′ a b
 assume P:Prof P′:Prof and iff: ∀ i. ((a,b) ∈ P i) = ((a,b) ∈ P′ i)
 hence [simp]: !!i x. (x,x) ~: P i ∧ (x,x) ~: P′ i
  by(simp add:Prof-def Pi-def slo-defs)
 have iff′: a≠b ⟶ (∀ i. ((b,a) ∈ P i) = ((b,a) ∈ P′ i))
```

using *iff* ‹*P*:*Prof*› ‹*P'*:*Prof*› **unfolding** *Prof-def Pi-def*
by *simp* (*metis iff notin-Lin-iff*)
let *?abP* = *Top* {*a*,*b*} *o P* **let** *?abP'* = *Top* {*a*,*b*} *o P'*
have ∀ *i c*. (*c*, *f ?abP*) : *?abP i* ⟶ (*c*, *f ?abP*) : *?abP' i*
using *una-Top*[*of P* {*a*,*b*}, *OF* ‹*P*:*Prof*›] *iff iff'* **by**(*auto simp add*:*Top-def*)
then have *f* (*Top* {*a*,*b*} ∘ *P*) = *f* (*Top* {*a*,*b*} ∘ *P'*)
using *Top-in-Prof*[*OF* ‹*P*:*Prof*›] *Top-in-Prof*[*OF* ‹*P'*:*Prof*›]
mono[*of Top* {*a*, *b*} ∘ *P*] **by** *metis*
thus $(a <_{swf\ f\ P}\ b) = (a <_{swf\ f\ P'}\ b)$ **by**(*simp add*: *swf-def*)
qed

lemma *dict-swf*: **assumes** *dictator* (*swf f*) *i* **shows** *dict f i*
proof (*auto simp*:*dict-def*)
fix *P a* **assume** *P*:*Prof a*≠*f P*
have *f* (*Top* {*a*,*f P*} ∘ *P*) = *f P*
using *mono*[*OF* ‹*P*:*Prof*› *Top-in-Prof*[*OF* ‹*P*:*Prof*›,*of* {*a*,*f P*}]]
by (*auto simp*:*Top-def*)
moreover have *P i* = {(*a*,*b*). *a*≠*b* ∧ *f*(*Top* {*a*,*b*} ∘ *P*) = *b*}
using *assms* ‹*P*:*Prof*› **by**(*simp add*:*dictator-def swf-def*)
ultimately show (*a*,*f P*) : *P i* **using** ‹*a*≠*f P*› **by** *simp*
qed

theorem *Gibbard-Satterthwaite*:
∃ *i*. *dict f i*
by (*metis Arrow SWF-swf una-swf IIA-swf dict-swf*)

end

theorem *Gibbard-Satterthwaite*:
¬ *manipulable f* ⟹ ∀ *a*.∃ *P*∈*Prof*. *a* = *f P* ⟹ ∃ *i*. *dict f i*
using *GS*.*Gibbard-Satterthwaite*[*of f*,*unfolded GS-def*]
by *blast*

end

References

[1] J. Geanakoplos. Three brief proofs of Arrow's impossibility theorem. *Economic Theory*, 26:211–215, 2005.

[2] N. Nisan. Introduction to mechanism design (for computer scientists). In N. Nisan, T. Roughgarden, E. Tardos, and V. Vazirani, editors, *Algorithmic Game Theory*. Cambridge University Press, 2007.