# FORMALIZATION OF AUCTION THEORY USING LEAN AND MATHLIB

WANG HAOCHENG

Xiamen University Malaysia

August 7, 2024

## Introduction

*Overview of Formalization in Auction Theory*

- Mathematics is typically presented in natural language.
- Formal languages for writing statements and representing reasoning.
- Formalization: using a computer to reason about mathematics.
- Ensures correctness of formal proofs is objective and algorithmically checkable.

## Project Overview

- Formalization of theorems in Auction Theory using Lean4 and Mathlib4.
- Lean4: A software tool to assist with the development of formal proofs.
- Mathlib4: A user-maintained library for the Lean Theorem Prover.
- Goal: Contribute formal statements of theorems to Mathlib4.
- Challenges: Code style adherence, resolving maintainer suggestions.

## Main Results

- Focus: Key results from Auction Theory, including First-Price and Second-Price auctions and Myerson's Lemma.
- Results: Formalization of one lemma in First-Price auction, two lemmas in Second-Price auction, and Myerson's Lemma.

## Formalization Process

- Expressing theorem statements in a formal language.
- Writing proofs using formal reasoning rules.
- Defining auction structures, bidding strategies, and utility functions.
- Using Lean4 and Mathlib4 for proof development.

## Table of Contents

## Lean and Mathlib

- Lean is a software tool developed by Leonardo de Moura at Microsoft Research in 2013.
- Lean 4 is the latest version, designed as a general-purpose programming language.
- Mathlib is a user-maintained library for the Lean Theorem Prover, containing both programming infrastructure and mathematics.

## Table of Contents

## What is a Proof Assistant

**Programming Language and Theorem Prover**

Figure: The Lean Theorem Prover and Programming Language

- A proof assistant provides a language for defining objects, specifying properties, and proving specifications.
- Two components: the kernel (checks correctness) and the elaborator (translates human-written proofs).
- Examples: Coq, Isabelle/HOL, and Lean.

## The Lean Theorem Prover and Programming Language

- Developed by Leonardo de Moura in 2013.
- Lean 4 released in 2021, designed as a general-purpose programming language.
- Capable of generating fast executables, supports calling into C++, and uses reference counting.
- Uses dependent types based on the calculus of inductive constructions (CIC).

## Table of Contents

## Mathlib: The Lean Mathematical Library

- Mathlib is a repository of formally formalized mathematical theorems.
- Established in 2017, designed as a basis for research-level mathematics.
- Contains programming infrastructure, mathematics, and tactics.
- Dynamic and active community with regular contributions.

## Mathlib: Automation and Metaprogramming

- Mathlib contains not only mathematical objects but also Lean metaprograms.
- Extensive use of metaprogramming for powerful automation and specialized commands.
- Simplification tactics (e.g., `simp`) perform nondefinitional directional rewriting.
- The `#lint` command performs various tests on declarations.
- The `simp` tactic simplifies expressions using tagged expressions.
- The `dsimp` tactic performs only definitional reductions.
- The `#lint` command checks for unused arguments, malformed names, and documentation requirements.

## Table of Contents

## Definition (Sealed-Bid Auction)

A sealed-bid auction operates under the following rules:

1. Each bidder $i$ privately communicates a bid $b_i$ to the seller.
2. The seller decides who gets the item.
3. The seller decides on a selling price.

A sealed-bid auction is a type of auction in which refers to a written bid placed in a sealed envelope. The sealed bid is not opened until the stated date, at which time all bids are opened together. The highest bidder is usually the winner of the bidding process.

- A sealed-bid auction involves written bids placed in sealed envelopes.
- The bids are opened together at a specified time.
- The highest bidder usually wins the auction.

## Single-Item Sealed Bid Auction

### Single-Item Auction

**Definition (Single-Item Auction)**

A single-item auction is an auction where only one item is involved.

## Single-Item Sealed Bid Auction

### Key Assumptions

- Each bidder $i$ has a nonnegative *valuation* $v_i$.
- This valuation is *private*, meaning it is unknown to others.
- Quasilinear Utility Model:

$$\text{utility}_i = \begin{cases} 0 & \text{if bidder } i \text{ loses,} \\ v_i - p & \text{if bidder } i \text{ wins at price } p. \end{cases}$$

## Single-Item Sealed Bid Auction

### Natural Language vs. Formal Statement

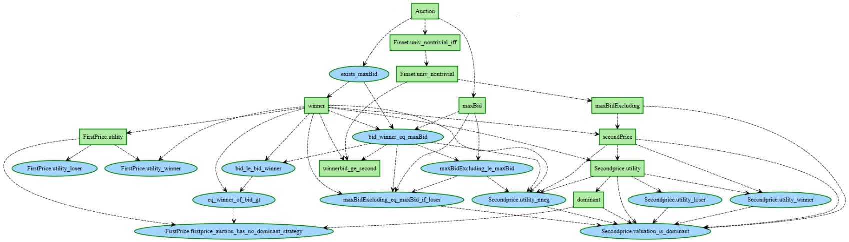
- Two primary concerns:
  1. Who wins the item.
  2. How much the winner should pay.
- Formalization includes:
  1. Specific number of bidders.
  2. Payment rules and criteria.
  3. Payoffs (utilities) for bidders.

For example, in a first-price auction, the highest bidder wins the item and pays their bid amount. Conversely, in a second-price auction, the highest bidder wins the item but pays the second highest bid. These setups form the basic configurations of these two auction types.

## Table of Contents

## Blueprint Tool

- The Blueprint tool by Patrick Massot helps in writing human-readable blueprints linked to Lean formalizations.
- Initially created for the Sphere Eversion Project.
- Extensively used in various projects since 2020.



Auction
Finset.univ_nontrivial_iff
exists_maxBid
Finset.univ_nontrivial
winner
maxBid
maxBidExcluding
FirstPrice.utility
bid_winner_eq_maxBid
secondPrice
FirstPrice.utility_loser
FirstPrice.utility_winner
bid_le_bid_winner
winnerbid_ge_second
maxBidExcluding_le_maxBid
Secondprice.utility
eq_winner_of_bid_gt
maxBidExcluding_eq_maxBid_if_loser
Secondprice.utility_nneg
dominant
Secondprice.utility_loser
Secondprice.utility_winner
FirstPrice.firstprice_auction_has_no_dominant_strategy
Secondprice.valuation_is_dominant

Figure: Blueprint of the Formalization

## Table of Contents

## Auction Structure in Lean

```
structure Auction where
    I : Type*
    hF : Fintype I
    hI : Nontrivial I
    v : I → ℝ
attribute [instance] Auction.hF Auction.hI
```

- `I`: Set of participants.
- `hF`: Finite participants.
- `hI`: Non-empty set of participants.
- `v`: Valuation function.

## Auction Structure in Lean

### Set of Participant

The variable $I$ represents the set of all participants in the auction. Each participant is considered as an element of a certain type, denoted by $I$.

```
structure Auction where
   I : Type*
```

# Auction Structure in Lean

## Finite Participants

The declaration *Fintype I* asserts that the set of participants is finite. This characteristic is crucial as it allows for the enumeration of participants, enabling calculations and analyses, such as determining the winner or calculating bids.

```
hF : Fintype I
```

## Auction Structure in Lean

### Existence of Participants

The *Nontrivial I* declaration ensures that the set of participants is non-empty and contains more than one element. This is intended to avoid trivial cases, ensuring that there is genuine competition among at least two bidders.

```
hI : Nontrivial I
```

# Auction Structure in Lean

## Valuation Function

The function $v : I \to \mathbb{R}$ maps each participant to their valuation of the item. This function represents the maximum amount of money that the participant is willing to pay for the item.

```
v : I → ℝ
```

## Auction Structure in Lean

### Attributes

The *attribute [instance]* declarations are used to indicate that *Fintype* and *Nontrivial* properties, are automatically applicable to the Auction structure whenever it is instantiated. These attributes help simplify further usage of the structure in the formalizing process by automatically incorporating these properties.

```lean
attribute [instance] Auction.hF Auction.hI
```

## Table of Contents

## First-Price Auction

### Definition of First-Price Auction

**Definition (First-Price Auction)**

In a first-price auction, each bidder submits a sealed bid of $b_i$. The highest bid wins the item and pays their bid amount $b_i$.

```
namespace Auction
variable {a : Auction} (b : a.I → ℝ)
```

## First-Price Auction
### Maximum Bid

**Definition (Maximum Bid)**

In the auction structure, the maximum bid refers to the highest bid among all bids.

```
@[simp]
def maxBid : ℝ := Finset.sup' Finset.univ
    Finset.univ_nonempty b
```

The definition of the maximum bid is the very first fuction defined, which will be useful in determination of the highest bidder in both first-price auction and second-price auctions. The function `maxBid` is defined to compute the highest bid provided by a bidding function *b*.

## First-Price Auction

### Maximum Bid

The `@[simp]` attribute in Lean serves a specific role in the simplification of proofs and computations. The `@` symbol is a collection of modifiers on a declaration, which apply modifiers or attributes to declarations. These attributes affect how functions, lemmas, or theorems are handled by Lean's compiler and proof assistant. The `simp` attribute, in particular, marks a function or theorem to be part of the simplification rules employed by Lean's simplification tactics. The `simp` tactic leverages these tagged expressions to simplify current goals, applying modifiers or attributes to declarations. These attributes influence how functions, lemmas, or theorems are treated by Lean's compiler and proof assistant.

## First-Price Auction

### Maximum Bid

```
@[simp]
def maxBid : ℝ := Finset.sup' Finset.univ
    Finset.univ_nonempty b
```

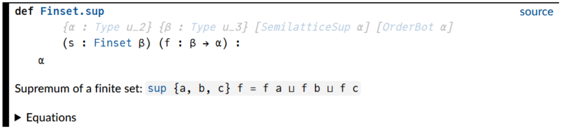
Here, the `@[simp]` attribute in the definition of `maxBid` instructs Lean to treat this function as a simplification rule. This approach enabling the Lean proof assistant to automatically simplify expressions that match the pattern defined by `maxBid`. It is particularly useful in keeping the proofs clear and concise.

# First-Price Auction

## Maximum Bid

```
@[simp]
def maxBid : ℝ := Finset.sup’ Finset.univ
    Finset.univ_nonempty b
```

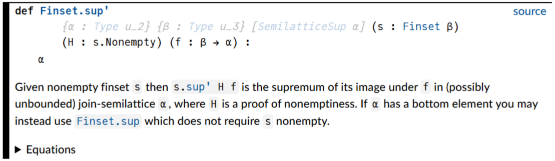
The `maxBid` function utilizes the `Finset.sup’` and `Finset.univ` operation from finset document in Mathlib4, which extracts the largest element from a universal set, therefore the highest bid is determined.



```
def Finset.sup                                                    source
      {α : Type u_2} {β : Type u_3} [SemilatticeSup α] [OrderBot α]
      (s : Finset β) (f : β → α) :
   α
```

Supremum of a finite set: `sup {a, b, c} f = f a ⊔ f b ⊔ f c`

▶ Equations



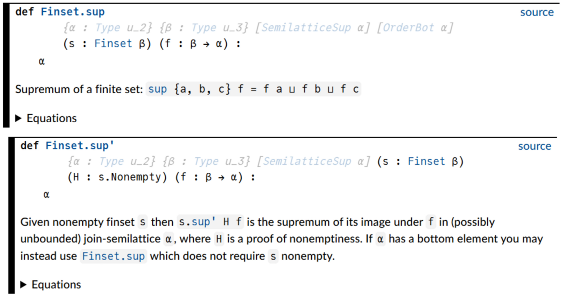
```
def Finset.sup'                                                   source
      {α : Type u_2} {β : Type u_3} [SemilatticeSup α] (s : Finset β)
      (H : s.Nonempty) (f : β → α) :
   α
```

Given nonempty finset `s` then `s.sup' H f` is the supremum of its image under `f` in (possibly unbounded) join-semilattice `α`, where `H` is a proof of nonemptiness. If `α` has a bottom element you may instead use `Finset.sup` which does not require `s` nonempty.

▶ Equations

# First-Price Auction

## Maximum Bid



```
def Finset.sup                                                    source
        {α : Type u_2} {β : Type u_3} [SemilatticeSup α] [OrderBot α]
        (s : Finset β) (f : β → α) :
    α
```

Supremum of a finite set: `sup {a, b, c} f = f a ⊔ f b ⊔ f c`

▶ Equations

```
def Finset.sup'                                                   source
        {α : Type u_2} {β : Type u_3} [SemilatticeSup α] (s : Finset β)
        (H : s.Nonempty) (f : β → α) :
    α
```

Given nonempty finset `s` then `s.sup' H f` is the supremum of its image under `f` in (possibly unbounded) join-semilattice `α`, where `H` is a proof of nonemptiness. If `α` has a bottom element you may instead use `Finset.sup` which does not require `s` nonempty.

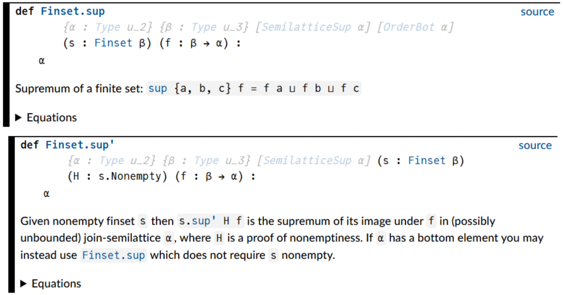
▶ Equations

Figure: Theroem Finset.sup and Finset.sup' in Mathlib4

It is important to note that Mathlib4 contains many definitions and theorems with similar names, which requires careful attention during the formalization of mathematics. In the formal proof of `maxBid`, we employ the theorem `Finset.sup'` rather than `Finset.sup`.

# First-Price Auction

## Maximum Bid



```
def Finset.sup                                                    source
        {α : Type u_2} {β : Type u_3} [SemilatticeSup α] [OrderBot α]
        (s : Finset β) (f : β → α) :
    α
```

Supremum of a finite set: `sup {a, b, c} f = f a ⊔ f b ⊔ f c`

▶ Equations

```
def Finset.sup'                                                   source
        {α : Type u_2} {β : Type u_3} [SemilatticeSup α] (s : Finset β)
        (H : s.Nonempty) (f : β → α) :
    α
```

Given nonempty finset `s` then `s.sup' H f` is the supremum of its image under `f` in (possibly unbounded) join-semilattice `α`, where `H` is a proof of nonemptiness. If `α` has a bottom element you may instead use `Finset.sup` which does not require `s` nonempty.

▶ Equations

Figure: Theroem Finset.sup and Finset.sup' in Mathlib4

The distinction between these two theorems is significant; `Finset.sup'` incorporates an additional condition `(H : s.Nonempty)`, which aligns more closely with the Nontrivial context found in first-price and second-price auctions. Therefore, we utilize `Finset.sup'` for the formal proof in this setting.

## First-Price Auction

### Existence of the Maximum Bid

```
lemma exists_maxBid : ∃ i : a.I, b i = a.maxBid b := by
    obtain ⟨i, _, h2⟩ := Finset.exists_mem_eq_sup'
     Finset.univ_nonempty b
    exact ⟨i, symm h2⟩
```

**Existance of the Maximum Bid** Existance of the Maximum Bid is formalized by the lemma `exists_maxBid` defines that there exists a participant 'i' whose bid equals the highest bid.

## First-Price Auction

### Formal Statement about Winner

**Winner** The `winner` is defined as the participant with the highest bid.

```
noncomputable def winner : a.I := Classical.choose
    (exists_maxBid b)
```

## First-Price Auction

### Formal Statement about Winner

Lemma `bid_winner_eq_maxBid` formalizes that he bid of the winner is always greater than or equal to the bids of all other participants.

```
lemma bid_winner_eq_maxBid : b (winner b) = maxBid b :=
    Classical.choose_spec (exists_maxBid b)
```

## First-Price Auction

### Using the Rewrite Tactic

In the lemma bid_winner_eq_maxBid, the tactic rw is first time used.rw (rewrite) tactic can transforming goals based on specified hypotheses.

```
lemma bid_le_bid_winner (j : a.I) : b j ≤ b (winner b) :=
    by
  rw [bid_winner_eq_maxBid b]
  exact Finset.le_sup' b (Finset.mem_univ j)
```

```
1 goal
a : Auction
b : a.I → ℝ
j : a.I
⊢ b j ≤ b (winner b)
```

Listing 1: Goal before Applying rw bid_winner_eq_maxBid b

## First-Price Auction

### Formalizing Dominant Strategy

**Definition (Dominant Strategy)**

A strategy is dominant in an auction if bidding $b_i$ ensures that the utility of `i` is maximized relative to any other bids `b` where `b i = b`$_i$.

```
def dominant (utility : (a.I → ℝ) → a.I → ℝ) (i : a.I)
    (bi : ℝ) : Prop :=
  ∀ b, utility b i ≤ utility (Function.update b i bi) i
```

## First-Price Auction

### Utility in First-Price Auction

**Definition (Utility in First-Price Auction)**

Consider any bidder $i$ with valuation $v_i$ and the bids $b_i$. The utility of bidder $i$ under strategy $b_i$ is given by:

$$\text{Utility}(b_i) = \begin{cases} v_i - b_i & \text{if } b_i > B \\ 0 & \text{if } b_i \leq B \end{cases}$$

```
namespace Firstprice
noncomputable def utility (i : a.I) : ℝ := if i = winner
    b then a.v i - b i else 0
```

## First-Price Auction

### Lemmas for Utility

```
lemma utility_winner (i : a.I) (H : i = winner b) :
    utility b i = a.v i - b i := by
  rw [H]
  simp only [utility, if_true]
lemma utility_loser (i : a.I) (H : i ≠ winner b) :
    utility b i = 0 := by
  rw [utility]
  simp only [H, if_false]
```

## First-Price Auction

### First-Price Auction has no Dominant Strategy

**Lemma (First-Price Auction has no dominant strategy)**

*In a sealed-bid first-price auction, no bidder has a dominant strategy.*

**Proof.**

Consider any arbitrary bidder $i$ with valuation $v_i$ and the bids $b_i$ , $b_j$ be bid of the other bidders. Define $B = \max_{j \neq i} b_j$ as the highest bid by any other bidder. $\epsilon$ be the least increment value in the auction. For bidder $i$ to have a dominant strategy, $b_i$ must maximize Utility$(b_i)$ for all possible $b_j$. However, if $b_i > B$ lead to winning with a positive utility, Utility$(b_i) = v_i - b_i$, which is maximized precisely when $b_i$ approaches $B$ from above. Since there always exist a $b'_i = B - \epsilon$ could lead to a higher utility when $v_i \geq B$. Utility$(b'_i) = v_i - B + \epsilon$ which is greater than Utility$(b_i) = v_i - b_i$ if $b_i = v_i$. Thus, there is no single $b_i$ that always lead to the highest utility, proving that no dominant strategy exists. $\square$

## First-Price Auction

### Formalization of No Dominant Strategy

```
theorem firstprice_auction_has_no_dominant_strategy (i :
    a.I) (bi : ℝ) :
    ¬ dominant utility i bi := by
  rw [dominant, not_forall]
  use Function.update (fun _ ↦ (bi - 2)) i (bi - 1)
  rw [utility, utility, if_pos (eq_winner_of_bid_gt _ i
    _), if_pos (eq_winner_of_bid_gt _ i _)]
  <;> intros <;> simp [*]
end Firstprice
```

## Table of Contents

## Second-Price Auction

**Definition (Second-Price Auction)**

In a second-price auction, each bidder submits a sealed bid of $b_i$. The highest bidder wins the item but pays the amount of the second highest bid submitted by another bidder.

```
noncomputable def maxBidExcluding (i : a.I) : ℝ :=
    Finset.sup' (Finset.erase Finset.univ i)
(Finset.Nontrivial.erase_nonempty
    (Finset.univ_nontrivial)) b
noncomputable def Secondprice : ℝ := maxBidExcluding b
    (winner b)
```

## Second-Price Auction
### Formal Statement about Winner

The lemma `winnerbid_ge_second` shows that the bid of the winner is always greater than or equal to the second highest bid.

```
namespace Secondprice
lemma winnerbid_ge_second : Secondprice b ≤ b (winner b)
    := by
  rw [bid_winner_eq_maxBid b]
  exact Finset.sup'_mono b (Finset.subset_univ _)
    (Finset.Nontrivial.erase_nonempty
    (Finset.univ_nontrivial))
```

The lemma `maxBidExcluding_le_maxBid` shows that the bid of the winner is always greater than or equal to the second highest bid.

```
lemma maxBidExcluding_le_maxBid {i : a.I}(b : a.I → ℝ):
    maxBidExcluding b i ≤ maxBid b := by
  apply Finset.sup'_mono
  exact Finset.subset_univ (Finset.erase Finset.univ i)
```

## Second-Price Auction

### Uility in Second-Price Auction

Notably, the definition of the dominant is applicable in both first-price auction and second-price auction. Hence we do not need to state it again in this namespace.

```
def dominant (utility : (a.I → ℝ) → a.I → ℝ) (i : a.I)
    (bi : ℝ) : Prop :=
  ∀ b, utility b i ≤ utility (Function.update b i bi) i
```

Quasilinear equation in `Secondprice` namespace.

```
noncomputable def utility (i : a.I) : ℝ := if i = winner
    b then a.v i - Secondprice b else 0
```

## Second-Price Auction

### Uility in Second-Price Auction

The two lemmas utility_winner and utility_loser formalize the conditions under which a participant's utility is calculated. If the bidder is the winner, then their utility is their valuation minus the second highest bid. If the bidder is not the winner, then their utility is 0.

```
lemma utility_winner (H: i = winner b) : utility b i =
    a.v i - Secondprice b := by
  rw [utility]; simp only [ite_true, H]
lemma utility_loser (H : i ≠ winner b) : utility b i = 0
    := by
  rw [utility]; simp only [ite_false, H]
```

## Lemma (Nonnegative Utility)

*In a second-price auction, every truthful bidder is guaranteed nonnegative utility.*

## Proof.

Losers receive utility 0. If a bidder $i$ is the winner, then her utility is $v_i - p$, where $p$ is the second-highest bid. Since $i$ is the winner (and hence the highest bidder) and bid her true valuation, $p \leq v_i$, and hence $v_i - p \geq 0$. □

## Second-Price Auction
### Nonnegative Utility

**Lemma (Nonnegative Utility)**

*In a second-price auction, every truthful bidder is guaranteed nonnegative utility.*

```
lemma utility_nneg (i : a.I) (H : b i = a.v i) : 0 ≤
    utility b i := by
  rcases eq_or_ne i (winner b) with rfl | H2
  · rw [utility, if_pos rfl, ← H, bid_winner_eq_maxBid b,
    sub_nonneg, Secondprice]
    exact maxBidExcluding_le_maxBid b
  · rw [utility, if_neg H2]
```

**Lemma (Incentives in Second-Price Auctions)**

*In a second-price auction, every bidder i has a dominant strategy: set the bid bi equal to her private valuation $v_i$.*

## Second-Price Auction

### Incentives in Second-Price Auctions

**Proof.**

Fix an arbitrary bidder $i$, valuation $v_i$, and the bids $b_{-i}$ of the other bidders. Here $b_{-i}$ means the vector of all bids, but with the $i$th component removed. We need to show that bidder $i$'s utility is maximized by setting $b_i = v_i$.
Let $B = \max_{j \neq i} b_j$ denote the highest bid by some other bidder. What's special about a second-price auction is that, even though there are an infinite number of bids that $i$ could make, only two distinct outcomes can result. If $b_i < B$, then $i$ loses and receives utility 0. If $b_i \geq B$, then $i$ wins at price $B$ and receives utility $v_i - B$.
We conclude by considering two cases. First, if $v_i < B$, the maximum utility that bidder $i$ can obtain is $\max\{0, v_i - B\} = 0$, and it achieves this by bidding truthfully (and losing). Second, if $v_i \geq B$, the maximum utility that bidder $i$ can obtain is $\max\{0, v_i - B\} = v_i - B$, and it achieves this by bidding truthfully (and winning). □

## Second-Price Auction

### Formal Proof of Incentives in Second-Price Auctions

```
theorem valuation_is_dominant (i : a.I) : dominant
    utility i (a.v i) := by intro b;
  have key : maxBidExcluding (Function.update b i (a.v
    i)) i = maxBidExcluding b i :=
    (Finset.sup'_congr _ rfl fun j hj ↦ dif_neg
    (Finset.ne_of_mem_erase hj))
  by_cases h1 : i = winner b
  · rw [utility_winner b h1, Secondprice, ← h1, ← key]
    by_cases h2 : i = winner (Function.update b i (a.v i))
    · rw [utility_winner _ h2, sub_le_sub_iff_left,
    Secondprice, ← h2]
    · rw [utility_loser _ h2, sub_nonpos,
    maxBidExcluding_eq_maxBid_if_loser _ h2]
      conv_lhs => rw [← Function.update_same i (a.v i) b]
      exact Finset.le_sup' _ (Finset.mem_univ i)
  · rw [utility_loser b h1]
    apply utility_nneg
    apply Function.update_same
```

## Second-Price Auction

### Dominant Strategy Incentive Compatible

**Definition (Dominant Strategy Incentive Compatible)**

A mechanism is dominant strategy incentive compatible (DSIC) or incentive compatible if:

1. Truthful bidding is a dominant strategy for all players, and
2. No player regrets participation (no player ever has negative utility).

## Second-Price Auction
### Conclusion

- The second-price auction is DSIC.
- Participants tend to bid their true valuations.
- This ensures that the item goes to the bidder who values it most.

## Table of Contents

## Myerson's Lemma

### Single-Parameter Environments

A good level of abstraction at which to state Myerson's lemma is *single-parameter environments*. Such an environment has some number $n$ of agents:

- Each agent $i$ has a private non-negative valuation $v_i$:
  - This valuation represents her value *per unit of stuff* that she acquires.
- There is a *feasible set* $X$:
  - Each element of $X$ is a non-negative $n$-vector $(x_1, x_2, \ldots, x_n)$.
  - $x_i$ denotes the *amount of stuff* given to agent $i$.

## Examples of Mechanisms in Single-Parameter Environments

### Formal Statement of Single Parameter Environment

A single parameter environment has some number $n$ of agents, and a feasible set $X$, in which each element is a non-negative vector $(x_1, x_2, \ldots, x_n)$, where $x_i$ denotes the amount of stuff given.

```
structure SingleParameterEnvironment where
  I : Type*
  INonempty : Nonempty I
  IFintype : Fintype I
  feasibleSet : Set (I → ℝ)
  feasibleSetNonempty : Nonempty feasibleSet
```

**Example 4.1 (Single-Item Auction)** In a single-item auction, $X$ is the set of 0-1 vectors that have at most one 1—that is, $\sum_{i=1}^{n} x_i \leq 1$.

## Correspondence of terms in auctions and mechanisms

| **Auction** | **Mechanism** |
|---|---|
| Bidder | Agent |
| Bid | Report |
| Valuation | Valuation |

Table: Correspondence of terms in auctions and mechanisms.

## Table of Contents

## Allocation and Payment Rules

Recall that a sealed-bid auction has to make two choices: who wins and who pays what. These two decisions are formalized via an *allocation rule* and a *payment rule*, respectively. Here are the three steps:

1. Collect bids $b = (b_1, \ldots, b_n)$ from all agents. The vector $b$ is called the **bid vector** or **bid profile**.
2. **allocation rule:** Choose a feasible allocation $x(b) \in X \subseteq \mathbb{R}^n$ as a function of the bids.
3. **payment rule:** Choose payments $p(b) \in \mathbb{R}^n$ as a function of the bids.

## Implementable Allocation Rule

Definition

**Definition (Implementable Allocation Rule)**

An allocation rule $x$ for a single-parameter environment is *implementable* if there is a payment rule $p$ such that the direct-revelation mechanism $(x, p)$ is Dominant Strategy Incentive Compatible (DSIC).

```
def implementable (ar : (E.I → ℝ) → E.feasibleSet) :=
  ∃ pr : (E.I → ℝ) → E.I → ℝ,
  @dsic E {allocationRule := ar, paymentRule := pr}
```

## Monotone Allocation Rule

Definition

**Definition (Monotone Allocation Rule)**

For a single-parameter environment, an allocation rule $x$ is *monotone* if for every agent $i$ and bids $b_{-i}$ by the other agents, the allocation $x_i(z, b_{-i})$ to $i$ is not decreasing in the bid $z$.

```
def monotone (ar : (E.I → ℝ) → E.feasibleSet) :=
  ∀ i : E.I,
  ∀ b : E.I → ℝ,
  nondecreasing (λ (bi : ℝ) => ar (with_hole b i bi) i)
```

## Table of Contents

## Utility and Dominant Strategy

### Utility in Single-Parameter Environment

We still need to state the Quasi-linear utility model under the new structure other than auction.

```
@[simp]
def utility (v : E.I → ℝ) (b : E.I → ℝ) (i : E.I) : ℝ
    :=
  v i * D.allocationRule b i - D.paymentRule b i
```

A dominant strategy for `i` is a strategy (i.e., a bid `bi`) that is guaranteed to maximize `i`'s utility, no matter what the other bidders do.

```
def dominant (v : E.I → ℝ) (bid_amount : ℝ) (i : E.I) :
    Prop :=
  ∀ b b': E.I → ℝ,
  b i = bid_amount →
  (∀ j : E.I, j ≠ i → b j = b' j) → @utility E D v b i
    ≥ @utility E D v b' i
```

## Dominant Strategy Incentive Compatible

### DSIC Definition

A system is dominant-strategy incentive compatible (DSIC) if truthful bidding is always a dominant strategy for every bidder and if truthful bidders always obtain nonnegative utility.

```
def dsic := ∀ (i : E.I), ∀ (v : E.I → ℝ), @dominant E D
    v (v i) i
def nondecreasing (f : ℝ → ℝ) := ∀ (x1 x2 : ℝ), x1 ≤
    x2 → f x1 ≤ f x2
```

## Myerson's Lemma

### Statement of Myerson's Lemma

**Lemma (Myerson's Lemma)**

*Fix a single-parameter environment.*

(a) *An allocation rule $x$ is implementable if and only if it is monotone.*

(b) *If $x$ is monotone, then there is a unique payment rule for which the direct-revelation mechanism $(x, p)$ is DSIC (Dominant Strategy Incentive Compatible) and $p_i(b) = 0$ whenever $b_i = 0$.*

(c) *The payment rule in (b) is given by an explicit formula.*

## Table of Contents

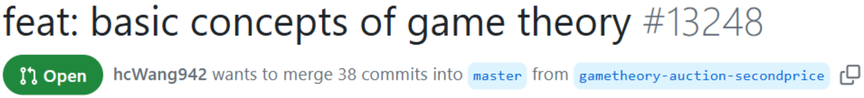
## Merge Code to Mathlib4

At the beginning of the project, we planned to sketch some proofs and then complete the formal proofs of three main theorems in First-Price and Second-Price auctions. After completing the formal statements, we attempted to extend the scope of the project to formalize Myerson's Lemma. We realized that we could try to contribute our formalized Auction Theory to the mathematics library of Lean4, Mathlib4.

## Merge Code to Mathlib4

### Files Changed

We created a new branch named `gametheory-auction-secondprice` for our contributions. Each time we want to upload the changes for the maintainer to review, we shall commit and push the updates use the command `git push`.



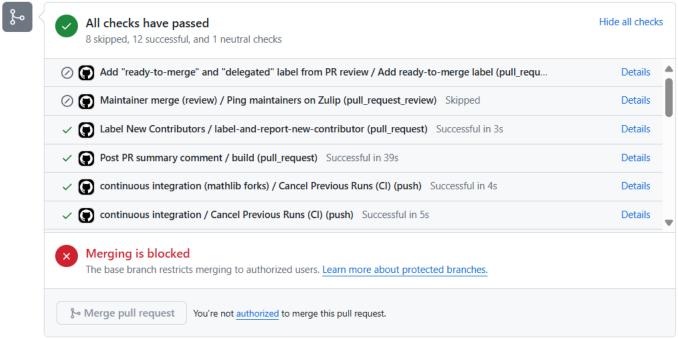
feat: basic concepts of game theory #13248

Open hcWang942 wants to merge 38 commits into `master` from `gametheory-auction-secondprice`

Figure: Files changed in pull request.

## Merge Code to Mathlib4
### Automatic Checks

In each push process, lint style, continuous integration (CI), files imported and so on will be automatically checked. The check process will show a green tick if everything works, a yellow circle if CI is still working, or a red cross if something went wrong.



All checks have passed
8 skipped, 12 successful, and 1 neutral checks

Hide all checks

⊘ Add "ready-to-merge" and "delegated" label from PR review / Add ready-to-merge label (pull_requ... Details
⊘ Maintainer merge (review) / Ping maintainers on Zulip (pull_request_review) Skipped Details
✓ Label New Contributors / label-and-report-new-contributor (pull_request) Successful in 3s Details
✓ Post PR summary comment / build (pull_request) Successful in 39s Details
✓ continuous integration (mathlib forks) / Cancel Previous Runs (CI) (push) Successful in 4s Details
✓ continuous integration / Cancel Previous Runs (CI) (push) Successful in 5s Details

Merging is blocked
The base branch restricts merging to authorized users. Learn more about protected branches.

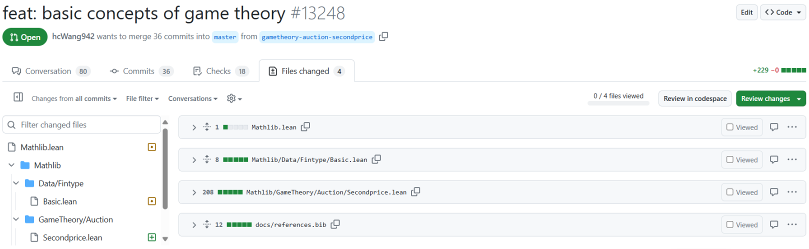
Merge pull request You're not authorized to merge this pull request.

Figure: Automatically check flow in each process.

# Merge Code to Mathlib4

## Summary of Changes

In summary, we added 86 lines of comments, pushed 36 commits, and reduced the total length of the code from 353 lines to 203 lines. We modified 4 files in our branch, awaiting the maintainers' review. We are still continuously refining and modifying the code in collaboration with the maintainers from Mathlib.



feat: basic concepts of game theory #13248

Open hcWang942 wants to merge 36 commits into master from gametheory-auction-secondprice

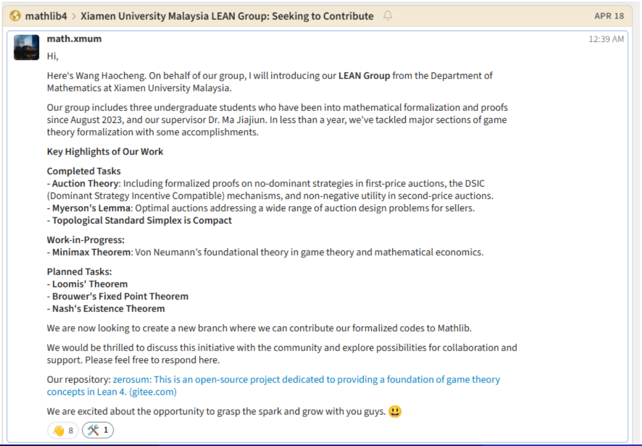
```
1    Mathlib.lean
8    Mathlib/Data/Fintype/Basic.lean
208  Mathlib/GameTheory/Auction/Secondprice.lean
12   docs/references.bib
```

Figure: Files changed in pull request.

## Table of Contents

# Preparation Work
## Zulip Discussion

To make the process of contributing as smooth as possible, we discussed our contribution on Zulip chat. Initially, we created a GitHub repository named *math-xmum* to store our existing code and submitted a project introduction on Zulip chat.



mathlib4 > Xiamen University Malaysia LEAN Group: Seeking to Contribute APR 18

**math.xmum** 12:39 AM

Hi,

Here's Wang Haocheng. On behalf of our group, I will introducing our **LEAN Group** from the Department of Mathematics at Xiamen University Malaysia.

Our group includes three undergraduate students who have been into mathematical formalization and proofs since August 2023, and our supervisor Dr. Ma Jiajiun. In less than a year, we've tackled major sections of game theory formalization with some accomplishments.

**Key Highlights of Our Work**

**Completed Tasks**
- **Auction Theory**: Including formalized proofs on no-dominant strategies in first-price auctions, the DSIC (Dominant Strategy Incentive Compatible) mechanisms, and non-negative utility in second-price auctions.
- **Myerson's Lemma**: Optimal auctions addressing a wide range of auction design problems for sellers.
- **Topological Standard Simplex is Compact**

**Work-in-Progress:**
- **Minimax Theorem**: Von Neumann's foundational theory in game theory and mathematical economics.

**Planned Tasks:**
- **Loomis' Theorem**
- **Brouwer's Fixed Point Theorem**
- **Nash's Existence Theorem**

We are now looking to create a new branch where we can contribute our formalized codes to Mathlib.

We would be thrilled to discuss this initiative with the community and explore possibilities for collaboration and support. Please feel free to respond here.

Our repository: zerosum: This is an open-source project dedicated to providing a foundation of game theory concepts in Lean 4. (gitee.com)

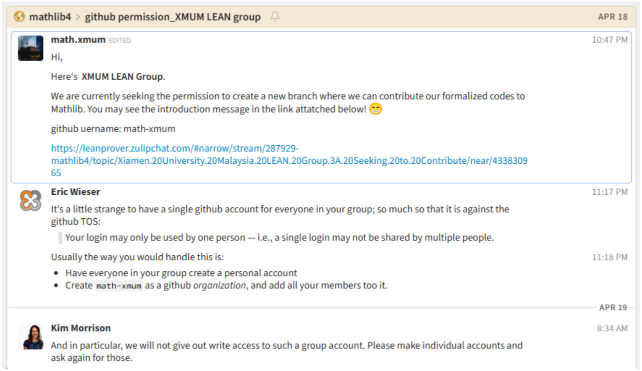
We are excited about the opportunity to grasp the spark and grow with you guys. 😃

👋 8 🛠 1

## Preparation Work

### GitHub Permission

When we attempted to apply for contributor permissions using the group GitHub account, our request was denied. We then applied for new contributor write access individually.



mathlib4 > github permission_XMUM LEAN group APR 18

**math.xmum** EDITED 10:47 PM

Hi,

Here's **XMUM LEAN Group**.

We are currently seeking the permission to create a new branch where we can contribute our formalized codes to Mathlib. You may see the introduction message in the link attatched below! 😁

github uername: math-xmum

https://leanprover.zulipchat.com/#narrow/stream/287929-mathlib4/topic/Xiamen.20University.20Malaysia.20LEAN.20Group.3A.20Seeking.20to.20Contribute/near/433830965

**Eric Wieser** 11:17 PM

It's a little strange to have a single github account for everyone in your group; so much so that it is against the github TOS:

> Your login may only be used by one person — i.e., a single login may not be shared by multiple people.

Usually the way you would handle this is: 11:18 PM

- Have everyone in your group create a personal account
- Create `math-xmum` as a github *organization*, and add all your members too it.

APR 19

**Kim Morrison** 8:34 AM

And in particular, we will not give out write access to such a group account. Please make individual accounts and ask again for those.

Figure: Zulip chat: github permission_XMUM LEAN group

## Naming Conventions
### Transition from Lean 3 to Lean 4

As part of our integration into Mathlib4, we aligned our code's naming conventions with the established standards. The evolution of naming conventions from Lean 3 to Lean 4 introduced a nuanced approach.

- Proofs and theorem names use `snake_case`.
- Types such as inductive types, structures, and classes use `UpperCamelCase`.
- Functions are named based on their output type.
- Non-function terms of Types typically use `lowerCamelCase`.

## Naming Conventions

### Examples

For example, when defining a theorem within our code, we ensure that its name is in snake_case to reflect its role as a Prop term.

```
theorem valuation_is_dominant (i : a.I) : dominant
    utility i (a.v i) := by
```

This definition illustrates the application of snake_case in theorem naming.

## Code Style

### Length of Code and Headers

Lines should not exceed 100 characters.
The file header must include copyright information, a list of contributors, and a brief description of the contents of the file. Following the header, all import statements should be listed consecutively without a line break.

```
/-
Copyright (c) 2024 Wang Haocheng. All rights reserved.
Released under Apache 2.0 license as described in the
    file LICENSE.
Authors: Ma Jiajun, Wang Haocheng
-/
import Mathlib.Data.Fintype.Basic
import Mathlib.Data.Finset.Lattice
import Mathlib.Data.Real.Basic
```

## Code Style
### Module Docstring

Following the copyright header and the imports, it is essential to include a module docstring. This docstring should provide a title for the file, a summary of its contents, any specific notation used, and references to relevant literature.

```
# Auction Theory
This file formalizes core concepts and results in auction
    theory.
## Main Definitions
- ‘Auction‘: Defines an auction with bidders and their
    valuations.
## Main Results
- ‘utility_nneg‘: Utility is non-negative if the bid
    equals the valuation in second price auction.
## References
* [T. Roughgarden, *Twenty lectures on Algorithmic Game
    Theory*][roughgarden2016]
## Tags
auction, game theory, economics, bidding, valuation
```

## Table of Contents

## Making a Pull Request

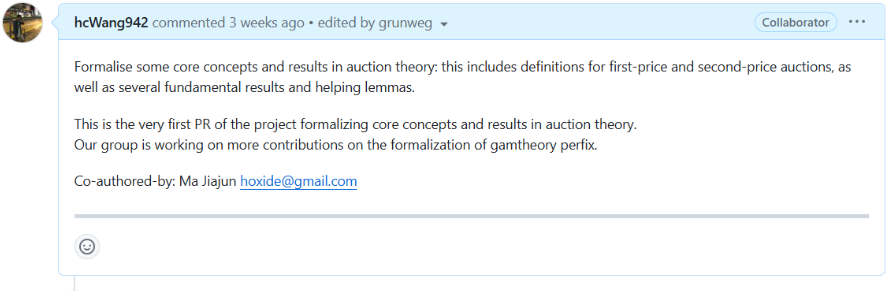
### Introduction

A pull request is a proposal to merge a set of changes from one branch into another. In a pull request, collaborators can review and discuss the proposed set of changes before they integrate the changes into the main codebase.

## Making a Pull Request

### Example

We made a pull request for our branch in the Mathlib4 repository. The main goal is to push the changes to a branch on the main repository. We need to make a comment and header with the tag feature (feat.), then add a label for maintainers to manage the pull request.



Formalise some core concepts and results in auction theory: this includes definitions for first-price and second-price auctions, as well as several fundamental results and helping lemmas.

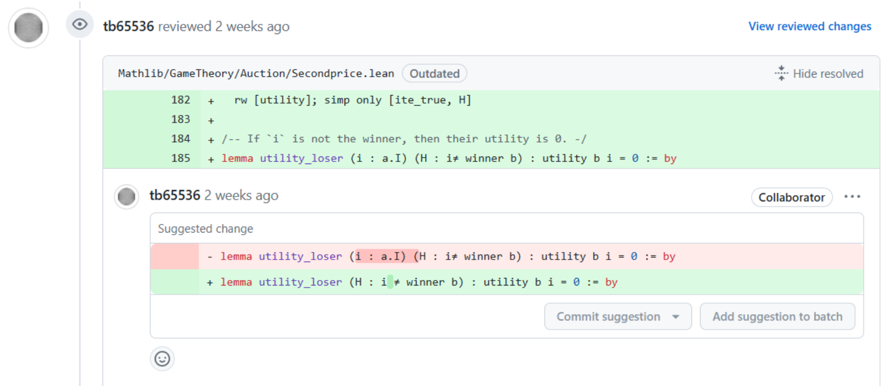
This is the very first PR of the project formalizing core concepts and results in auction theory.
Our group is working on more contributions on the formalization of gamtheory perfix.

Co-authored-by: Ma Jiajun hoxide@gmail.com

Figure: Comment and Header for pull request.

## Making a Pull Request
### Review and Modify

After reviewing, we follow the conversation raised by the maintainer to modify the original code in VS Code and push the modified code in a new commit.


tb65536 reviewed 2 weeks ago

Mathlib/GameTheory/Auction/Secondprice.lean

```
+   rw [utility]; simp only [ite_true, H]
+
+ /-- If `i` is not the winner, then their utility is 0. -/
+ lemma utility_loser (i : a.I) (H : i≠ winner b) : utility b i = 0 := by
```

tb65536 2 weeks ago

Suggested change

```
- lemma utility_loser (i : a.I) (H : i≠ winner b) : utility b i = 0 := by
+ lemma utility_loser (H : i ≠ winner b) : utility b i = 0 := by
```

Figure: Conversation raised by the maintainer.

## Future Vision of AI and Mathematics

### Drawbacks of AIGC

While advancements in Artificial Intelligence Generated Content (AIGC), such as GPT-4, have significantly improved the ability to generate human-like text, there remain notable limitations. One of the primary drawbacks is the current weakness in reasoning.
Despite its impressive capabilities in natural language processing and generation, GPT-4 can struggle with complex logical reasoning and maintaining consistent, coherent arguments over extended text.

## Future Vision of AI and Mathematics

### AI for Mathematics: Integrating AI to Assist Formalization

- **AI for Science Paradigm:**
  - AI has been successfully empowering scientific research.
  - However, applying AI to mathematics, especially theorem proving, remains challenging.
- **Traditional Mathematical Formalization:**
  - Traditionally done by mathematicians, this approach is time-consuming and prone to errors.
  - AI development offers new opportunities for formalization and theorem proving.
- **Role of Proof Assistants:**
  - Proof assistants like Lean help develop formal proofs by providing a structured framework.
  - They ensure correctness at each step of a logical argument.
- **Benefits of Integrating AI and Proof Assistants:**
  - Enhances accuracy and precision in mathematical proofs and logical arguments, reducing errors.
  - Augments AI capabilities, enabling it to handle complex tasks more reliably.

## Future Vision of AI and Mathematics

### Mathematics for AI: Enhancing AI using Mathematics

To further improve AI systems and address the inherent weaknesses, several mathematical theories and techniques can be employed.

- **Formal Verification:**
  - Utilizes formal methods to verify the correctness of algorithms and systems.
  - Ensures that systems behave as intended under all conditions.
- **Logic and Proof Theory:**
  - Applies principles from logic and proof theory to enhance AI's reasoning capabilities.
- **Type Theory:**
  - Manages and ensures the consistency of data types within AI.
- **Algorithmic Game Theory:**
  - Enables AI to make strategic decisions in competitive and uncertain environments.

By integrating these mathematical theories, AI systems can become more reliable, and capable of sophisticated reasoning. This approach paves the way for future advancements in addresses the current drawbacks.

## Future Vision of AI and Mathematics

### Conclusion

The collaboration between AI and formal methods in mathematics represents a promising direction for overcoming current limitations and enhancing the capabilities of AI systems.
By using proof assistants and mathematical theories, we can develop AI that is not only more powerful but also more accurate and reliable. This relationship between AI and mathematics will be essential in tackling complex problems and advancing the development of LLM.

## References

The Lean Community.
*The Lean Book: Formal Methods with Lean*.
Online Book, Cambridge University Press, 2023.
https://doi.org/10.1017/CBO9781316779309.

Max Zipperer.
*Using Lean in the Mathematics Classroom*.
Master's Thesis, 2022.
https://www.contrib.andrew.cmu.edu/~avigad/Students/zipperer_ms_thesis.pdf.

Riccardo Brasca.
*Luxembourg 2021 Presentation*.
Conference Presentation, 2021.
https://webusers.imj-prg.fr/~riccardo.brasca/event/luxembourg-2021/pres.pdf.

## References

Jeremy Avigad.
*Practical Foundations for Formal Methods - Slides.*
Lecture Slides, 2019.
https://www.andrew.cmu.edu/user/avigad/Teaching/practical/slides_upitt.pdf

The Lean Community.
*Functional Programming in Lean.*
Online Documentation, 2023.
https://lean-lang.org/functional_programming_in_lean/getting-to-know/summary.html.

The Lean Community.
*The Lean Community Website.*
Website, 2023.
https://leanprover-community.github.io/.

Thanks for listening.

Q&A Session