

**FORMALIZATION OF AUCTION THEORY USING
LEAN AND MATHLIB**

WANG HAOCHENG

XIAMEN UNIVERSITY MALAYSIA

2024



XIAMEN UNIVERSITY MALAYSIA
廈門大學 馬來西亞分校

FINAL YEAR PROJECT REPORT

**FORMALIZATION OF AUCTION THEORY USING
LEAN AND MATHLIB**

NAME OF STUDENT : WANG HAOCHENG
STUDENT ID : MEC2009470
SCHOOL/FACULTY : DEPARTMENT OF MATHEMATICS
PROGRAMME : MATHEMATICS AND APPLIED MATHEMATICS
INTAKE : 2021/09

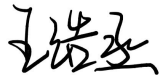
SUPERVISOR : PROF./DR. MA JIAJUN

JULY 2024

DECLARATION

I hereby declare that this thesis is based on my original work except for citations and quotations which have been duly acknowledged. I also declare that it has not been previously and concurrently submitted for any other degree or award at Xiamen University Malaysia or other institutions.

Signature :



Name : WANG HAOCHENG


ID No. : MEC2009470

Date : July 5, 2024

APPROVAL FOR SUBMISSION

I certify that this project/thesis, entitled “FORMALIZATION OF AUCTION THEORY USING LEAN AND MATHLIB”, and prepared by WANG HAOCHENG, has met the required standard for submission in partial fulfilment of the requirements for the award of Bachelor of Science in Mathematics and Applied Mathematics (Honours) at Xiamen University Malaysia.

Approved by,

Signature :  _____

Supervisor : Prof./Dr. MA JIAJUN _____

Date : July 5, 2024 _____

The copyright of this report belongs to the author under the terms of Xiamen University Malaysia copyright policy. Due acknowledgement shall always be made of the use of any material contained in, or derived from, this project report.

©2024, WANG HAOCHENG. All right reserved.

ACKNOWLEDGEMENTS

This academic journey has been a transformative experience for me. I faced many challenges, but this is the most difficult section in my dissertation. I have so many to say, but words failed to convey my feelings.

As a rule of thumb, I am deeply grateful to my supervisor, Prof./Dr. MA JI-AJUN, for the invaluable guidance, unwavering support and the enormous patience not just throughout the development of the project but to my undergraduate study. His mentorship has been instrumental in my growth and success. By no means I would gain the invaluable experience in the events at NUS and PKU which has greatly broaden my horizon of the formal mathematics world without him. I would also like to extend my sincere gratitude to my teammate Zhang Xinyuan, Wang Xuhui and Zhang Shaojun for their constructive comments and late-night time spent together, which significantly enhanced the quality and outcomes of my final year project.

Moreover, I would also like to express my gratitude to my peer Chen Yuxiang, my friends Li zheng, Su Bingkai, Huang Yuzhang, Jia Bohan, He Juncheng, Lai Xiangjun, Nong Tengxiao and so on, who have given me encouragement and dragged me out whenever I get stuck in the quagmire of life.

In addition, I would show my gratefulness to my girlfriend, Zong Huiyu, for illuminating my present and future. Thanks for her understanding, support, companionship, love and the pineapple fried rice sent when I worked on this.

Lastly, I must express my deepest appreciation for my biggest source of motivation, my beloved family, who stepped in not only with emotional reassurance, but also with critical financial support, when unexpected personal challenges arose. Their unconditional love and sacrifice have been the foundation during this challenging academic pursuit. My parents were the very first teachers of me. I owe an immense debt of gratitude to them, who nurtured my curiosity and supported my educational endeavours from the very beginning.

As I submit this thesis, my journey as an undergraduate comes to the full-stop. It is only fitting to express my heartfelt gratitude not just to those who contributed to its realization, but the hardships that have shaped my scholarly path. For me, myself and I, the person who is not mentioned above, and merely I have never show my thankfulness to. There is only one sentence that hoping you would always remember. *Keep bounding the rock.*

ABSTRACT

This thesis presents the formalization of fundamental theorems in Auction Theory using the Lean theorem prover and its accompanying mathematical library, Mathlib. Auction Theory is a component of economic theory, involves the design and analysis of auction mechanisms, which are formalized here to ensure mathematical precision. This project focuses on formalizing theorems related to first-price and second-price auctions, as well as Myerson's Lemma.

First, the fundamental concepts of first-price and second-price auctions are explored. In a first-price auction, the highest bidder wins and pays their bid amount, while in a second-price auction, the highest bidder wins but pays the second-highest bid. Formal statements and proofs for theorems regarding dominant strategies and bidder utility are developed. Specifically, the first-price auction is shown to have no dominant strategy, whereas the second-price auction incentivizes bidders to bid their true valuations.

Myerson's Lemma, a cornerstone of auction theory, is also formalized. It states that an allocation rule is implementable if and only if it is monotone, and it provides a unique payment rule ensuring the mechanism is dominant strategy incentive compatible (DSIC). The formalization process includes defining auction structures, bidding strategies, utility functions, and formal proofs of these theorems using Lean.

The contribution of these formalized results into Mathlib4 demonstrates the capabilities of modern proof assistants in handling complex economic models. This work not only contributes to the Lean community but also enhances the application of formal methods in economics and computer science, paving the way for future research and development in auction theory and mechanism design.

Keywords: Auction Theory, Lean Theorem Prover, Mathlib, Myerson's Lemma, Mechanism Design.

CONTENTS

1	Introduction	1
1.1	Main result	2
1.1.1	First-Price and Second-Price Auction	2
1.1.2	Myerson's Lemma	2
1.1.3	Formalization Process	3
2	Lean and Mathlib	4
2.1	The Lean Theorem Prover and Programming Language	4
2.1.1	What is a Proof Assistant	4
2.1.2	The Lean Theorem Prover and Programming Language	4
2.2	Mathlib: The Lean Mathematical Library	5
3	FIRST-PRICE AUCTION AND SECOND-PRICE AUCTION	7
3.1	Sealed Bid Auction	7
3.2	Lean blueprints Tool	8
3.2.1	Dependency Graph	8
3.3	Formalization of Auction Structure	11
3.3.1	Natural Language Statement of Auction Structure	11
3.3.2	Formal Statement of Auction Sturcture	11
3.4	First-price Auction	12
3.4.1	First-Price Auction	12
3.4.2	Formal Statement about Bid	13
3.4.3	Formal Statement about Winner	15
3.4.4	Dominant Strategy	16
3.4.5	Utility	16
3.4.6	First-Price Auction has no dominant strategy	18
3.5	Second-price Auction	21
3.5.1	Second-Price Auction	21

3.5.2	Formal Statement about Winner	22
3.5.3	Dominant Strategy	23
3.5.4	Utility	23
3.5.5	Incentives in Second-Price Auctions	24
3.5.6	Nonnegative Utility	25
3.5.7	Dominant Strategy Incentive Compatible	26
4	Myerson's Lemma	27
4.1	Single-Parameter Environments	27
4.1.1	Formal Statement of Single-Parameter Environment	27
4.1.2	Examples of Mechanisms in Single-Parameter Environments	27
4.2	Allocation and Payment Rules	28
4.3	Statement of Myerson's Lemma	29
4.3.1	Implementable Allocation Rule	29
4.3.2	Formal Statement of Implementable Allocation Rule	30
4.3.3	Monotone Allocation Rule	30
4.3.4	Formal Statement of Monotone Allocation Rule	30
4.3.5	Myerson's Lemma	32
5	Merge code to Mathlib4	33
5.1	Preparation Work	34
5.1.1	Zulip Discussion	34
5.2	Code Style	36
5.2.1	Naming Conventions	36
5.2.2	Length of Code	37
5.2.3	Header and Import Statement	37
5.2.4	Module Docstring	38
5.3	Making a Pull Request	40
5.4	Challenges in the Merging Process	40
5.4.1	Proof Simplification	41
5.4.2	Adding Theorems to Mathlib.Data.Fintype.Basic	43
6	Conclusion	45
6.1	The Significance and Challenges of Formalized Mathematics	45

6.1.1	The Significance of Formalized Mathematics	45
6.1.2	Challenges of Formalized Mathematics	46
6.2	Future vision of AI and Mathematics	47
6.2.1	Drawbacks of AIGC	47
6.2.2	AI for Mathematics: Integrating AI to assist Formalization	47
6.2.3	Mathematics for AI: Enhancing AI using Mathematics	48
6.3	Large Language Models for Formalized Mathematics	48
6.3.1	Large Language Models for Formalized Mathematics	48
6.3.2	Formalized Mathematics for Large Language Models	49
6.3.3	Scenarios of Applying Large Language Models in Formalized Mathematics	49
A	Formal Statement of Auction Theory	51
B	Formal Statement of Myerson’s Lemma	60

LIST OF TABLES

4.1	Correspondence of terms in auctions and mechanisms.	28
-----	---	----

LIST OF FIGURES

2.1	The Lean Theorem Prover and Programming Language [Com23e]	5
3.1	Blueprint of the Formalization	9
3.2	Blueprint dependency graph of the Formalization	9
3.3	Blueprint Book of the Formalization	10
3.4	Theroem <code>Finset.sup'</code> and <code>Finset.sup</code> in Mathlib4	14
3.5	Theroem <code>Function.update</code> in Mathlib4	19
5.1	Files changed in pull request.	33
5.2	Automatically check flow in each process.	34
5.3	Files changed in pull request.	34
5.4	Zulip chat: Xiamen University Malaysia LEAN Group Seeking to Contribute	35
5.5	Zulip chat: github permission_XMUM LEAN group	36
5.6	File modified in reference.bib in Mathlib4 project.	39
5.7	Comment and Header for pull request.	40
5.8	Conversation raised by the maintainer.	41
5.9	Add two theorems supporting Auction Theory formalizations	43

CHAPTER 1

INTRODUCTION

Mathematics is typically presented in natural language, results and proofs are given this way. Mathematicians use special symbols to denote mathematical objects and operations, but the reasoning is presented in natural language. Logicians have defined formal languages for writing statements and representing reasoning. The process of formalization of mathematics is a method of converting natural mathematical language into formal logic, ensuring the strictness and precision of mathematical reasoning.[Zip22]

These formal languages specify the rules of reasoning. Given these formal languages and the prescribed rules of reasoning, an reasoning is correct if it is licensed by the prescribed rules of reasoning at each step, and the correctness of the reasoning can be checked by checking whether it is allowed by the prescribed rules of reasoning at each step.[Avi19] Because the rules are mechanical, the process of checking for correctness is also mechanical. This ensures that the correctness of a formal proof is objective and algorithmically checkable.[Bra21]

This project finish the formalization process of several theorems in Auction Theory using the proof assistant Lean4 and its mathematical library, Mathlib4. Lean4 is a software tool to assist with the development of formal proofs. Mathlib4 is a user maintained library for the Lean Theorem Prover. It contains both programming infrastructure and mathematics, as well as tactics that might be used to complete the formalization.

The further goal is to contribute formal statement of theorems to the repository of Mathlib4 by creating a pull request. This is believed to be a challenging process. The code ready for merge has to follow the code style, including the length of each line, proper header and import statement, module docstring and comments on each proposition. Resolve the suggestions from maintainer is another long-lasting task to handle. This process do bridge the gap between informal and formal mathematical reasoning and demonstrate the capabilities of modern proof assistants in handling complex mathematical concepts.

1.1 Main result

Auction Theory is a branch of economics that deals with the design and analysis of auction mechanisms. We will focus on formalizing key results from Auction Theory, including first-price and Second-Price auctions and Myerson's Lemma. Main result of formalization including one lemma in First-Price auction, two lemmas in second-price auction and the Myerson's Lemma.

1.1.1 First-Price and Second-Price Auction

Lemma 1.1 (First-Price Auction has no dominant strategy). In a sealed-bid first-price auction, no bidder has a dominant strategy.

Lemma 1.1 state that there is no dominant strategy in first-price auction. This lemma highlights the complexity of strategy selection in first-price auctions and the need for rational decision-making in a competitive and uncertain environment. This understanding has significant theoretical and practical implications for auction designers and participants.

Lemma 1.2 (Incentives in Second-Price Auctions). In a second-price auction, every bidder i has a dominant strategy: set the bid b_i equal to her private valuation v_i . [Rou20]

Lemma 1.3 (Nonnegative Utility). In a second-price auction, every truthful bidder is guaranteed nonnegative utility. [Rou20]

The Lemma 1.2 states Incentives in Second-Price Auctions, in which states bidding one's valuation is a dominant strategy in second-price auction. The Lemma 1.3 Nonnegative Utility states that the utility of bidder is non-negative if the bid equals the valuation in second-price auction. So in a second-price auction everyone will bid truthfully, so we will actually give the item to the player who values it the most. The combination of the two lemmas is exactly the definition of one mechanism is dominant strategy incentive compatible.

1.1.2 Myerson's Lemma

Lemma 1.4 (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.

[Rou20]

Myerson’s Lemma states that an allocation rule x is implementable if and only if it is monotone. This means that if a bidder increases their bid, their allocation should not decrease. If the allocation rule x is monotone, there exists a unique payment rule such that the direct-revelation mechanism (x, p) is DSIC. Moreover, the payment $p_i(b)$ is zero whenever the bid b_i is zero. The payment rule mentioned in (b) is given by an explicit formula, which ensures the mechanism is DSIC. It generalizes beyond specific auction types (like first-price or second-price auctions) to a broader class of single-parameter environments, making it widely applicable.

1.1.3 Formalization Process

The formalization process can be divided into two main parts: expressing the statements of theorems in a formal language, and writing proofs using a fixed set of formal reasoning rules that can be checked algorithmically. This involves defining the auction structures, bidding strategies, and utility functions formally, and proving fundamental results about these structures using Lean4 and Mathlib4.

This project demonstrates that formal methods are capable of verifying complex mathematical theorems and providing a formalized foundation for further advancements in the field of auction theory. By integrating these formalized results into Mathlib4, this work not only contributes to the community but also enhances the practical applications of auction theory in economics and computer science. The successful formalization of these auction mechanisms and Myerson’s Lemma sets a precedent for the verification of economic models and opens up possibilities for automating and optimizing auction design.

Future work may explore extending these formalized results to more complex settings, integrating machine learning techniques to optimize auction mechanisms.

CHAPTER 2

LEAN AND MATHLIB

The Lean Theorem Prover and Programming Language is developed principally by Leonardo de Moura when he was at Microsoft Research in 2013. Lean is a software tool to assist with the development of formal proofs by human-machine collaboration. Lean 4 is the latest version.

Mathlib is a user maintained library for the Lean Theorem Prover. It contains both programming infrastructure and mathematics, as well as tactics that use the former and allow to develop the latter.

2.1 The Lean Theorem Prover and Programming Language

2.1.1 What is a Proof Assistant

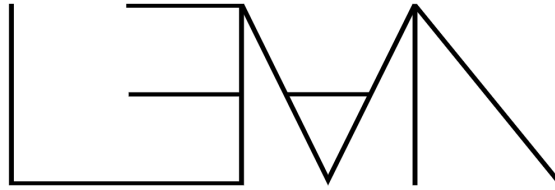
A proof assistant is a piece of software that provides a language for defining objects, specifying properties of these objects, and proving that these specifications hold.[Bra21] In a more general word, proof assistant translate state written by human being to the state that totally precise. A proof assistant has two components. First is the kernel, which checks that the proof is formally correct. The other is the elaborator. It translates a proof written by a human to something the kernel can check.

The system checks that these proofs are correct down to their logical foundation. This combination of human creativity with machine precision improves the reliability of proofs and supports complex mathematical reasoning. In a formalization, all definitions are precisely specified and all proofs are guaranteed to be correct. For instance, Coq, Isabelle/HOL (Higher-order logic) and Lean.

2.1.2 The Lean Theorem Prover and Programming Language

The Lean is first developed in 2013 by Leonardo de Moura at Microsoft Research.[Com23e] The Lean 4 was released at the beginning of 2021, marking a significant evolution from

Lean 3. The primary distinction is that Lean 4 is designed as a general-purpose programming language, not solely as an interactive theorem prover. It is capable of generating executables that are impressively fast. Additionally, Lean 4 supports calling into C++ and employs reference counting techniques that enable functionally written code to be compiled such that it updates in-place when appropriate.



Programming Language and Theorem Prover

Figure 2.1: The Lean Theorem Prover and Programming Language [Com23e]

Lean uses a system of dependent types based on the calculus of inductive constructions (CIC), and is particularly known for its use of dependent type theory, which allows types to depend on values. Lean checks that every expression has a type. Types are treated as first-class citizens in Lean.[Com23b] This means that just like values, types can be passed as arguments, returned from functions, and manipulated within the language. This feature is significant for modeling complex mathematical and computational concepts directly in the type system. Brand new types can be introduced to Lean using the 'structure' or 'inductive' features. These new types are not considered to be equivalent to any other type, even if their definitions are otherwise identical. This also enhances the flexibility and creativity of Lean.

2.2 Mathlib: The Lean Mathematical Library

Mathlib is a user maintained library for the Lean theorem prover. An easy way to understand is that mathlib is a repository of mathematical formalized proofs. Essentially, it functions as a comprehensive repository of formally formalized mathematical theorems, where the theorems can be imported into respective projects, thereby facilitating the proof development and formalization.[Com23f]

Mathlib was first established in 2017 by a group of users with very diverse backgrounds, where the Lean 3 is the latest version. It contains both programming infrastructure and mathematics, as well as tactics that use the former and allow to develop the latter. The mathlib library is designed as a basis for research level mathematics. The development of Mathlib is characterized by its dynamic and active community. With regular contributions from a global array of participants, The library contains useful definitions for programming. This project is very active, with many regular contributors and daily activity. Currently, much of mathlib consists of undergraduate level mathematics.

CHAPTER 3

FIRST-PRICE AUCTION AND SECOND-PRICE AUCTION

This chapter provides a comprehensive explanation for the process of formalization of Auction Theory. We first present core concepts and helping functions from Auction Theory and Game Theory as they are presented informally (i.e., in natural language statement), then present the concept statements formally. Some tactics and theorems in Mathlib4 used will be highlighted.

3.1 Sealed Bid Auction

A sealed-bid auction involves bidders submitting written bids in sealed envelopes. These bids remain confidential until a specified date, when all are simultaneously opened. Typically, the highest bidder wins the auction.

Definition 3.1 (Single-Item Auction). A single-item auction is an auction involving the sale of only one item.[Rou20]

Definition 3.2 (Sealed-Bid Auction). A sealed-bid auction follows these rules:

1. Each bidder i submits a private bid b_i to the seller.
2. The seller selects the winning bidder.
3. The seller determines the selling price.

[Rou20]

In this auction type, bidders do not know the amounts bid by others and can submit only one bid. The bidder's utility follows the *quasilinear utility model*, which is the simplest natural function component.

Assumptions for Single-Item Sealed Bid Auction

The first assumption is that each bidder i has a nonnegative *valuation* v_i , representing their maximum willingness-to-pay. Hence, bidder i aims to purchase the item at a price no higher than v_i .

Another critical assumption is that this valuation is *private*, meaning it is unknown to both the seller and other bidders. In a sealed-bid auction, all participants submit their bids simultaneously in sealed envelopes, ensuring that no bidder is aware of others' bids.

Definition 3.3 (Quasilinear Utility Model). If a bidder i loses the auction, her utility is 0. If the bidder wins at a price p , her utility is $v_i - p$. i.e.

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

[Rou20]

3.2 Lean blueprints Tool

The project has been greatly assisted by the innovative tools developed by Patrick Massot, notably the Lean Blueprints plugin.[Mas23] This tool is based on the *plasTeX* framework, enabling the creation of detailed, human-readable blueprints for Lean 4 formalizations. Originally inspired by Terence Tao's insightful blog posts *Formalizing the proof of PFR in Lean4 using Blueprint: a short tour*[Tao23], this infrastructure was first implemented in the Sphere Eversion Project in 2020. Since its inception, Lean Blueprints has become a critical resource for various projects, and fills the gap between complex theoretical formalizations and accessible mathematical proofs.

By providing a clear structure and dependency graph, it has greatly assist the in management and visualization of the complex details involved in formalization process for researchers and contributors in the Lean community.

3.2.1 Dependency Graph

In the Lean Blueprints tool, a color-coded system is employed to visually represent the status of mathematical results, such as definitions, theorems, and lemmas, during the formalization process. The use of boxes and ellipses helps differentiate between types of results, with specific color schemes indicating their current formalization status. A blue border around a result indicates that the statement of the result is fully prepared for formalization, meaning all necessary prerequisites have been completed. Similarly, a blue background signifies that the proof of the result is ready for formalization, with all preliminary work in

place. On the other hand, a green border shows that the statement of the result has been formally integrated into the repository, while a green background denotes that the proof itself has also been formalized.

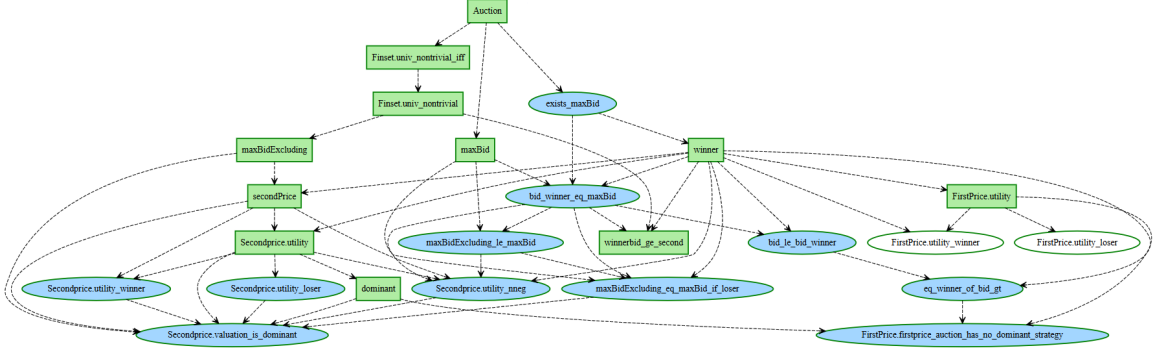


Figure 3.1: Blueprint of the Formalization

Blueprint provides a visual representation of the formalization process, allowing users to click on individual elements to view their definitions in the code. In Figure 3.2, the ‘maxBid’ element is highlighted as it represents the function used to compute the highest bid given a bidding function ‘b’. By exploring these elements interactively, users can gain a deeper understanding of the structure and dependencies within the formalization.

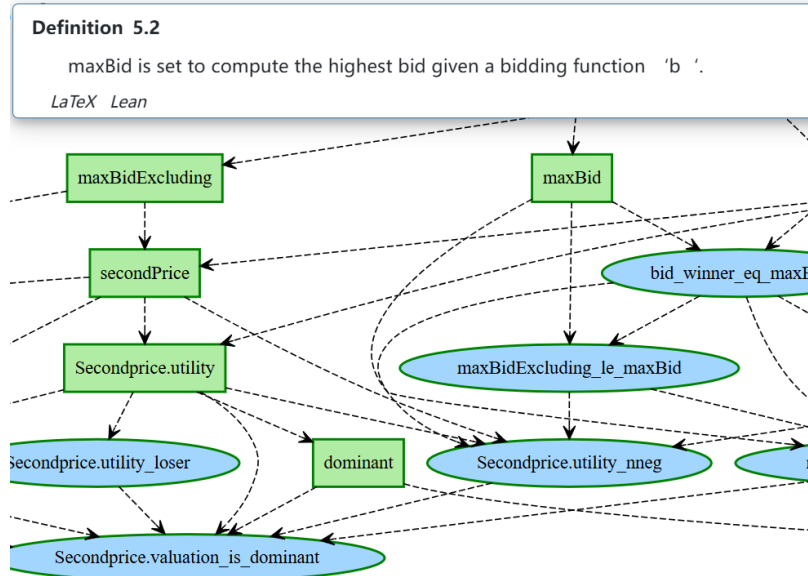


Figure 3.2: Blueprint dependency graph of the Formalization

This first line starts the definition environment and labels it as `maxBid`. This label can be

used for referencing this definition elsewhere in the document. `\leanok` indicates that the Lean code that follows has been verified and is correct. `\lean{maxBid}` specifies that the Lean code for `maxBid` is being defined or referenced here. `\uses{Auction}` indicates that the `maxBid` definition uses or depends on the `Auction` module or context within Lean. The description `maxBid is set to compute the highest bid given a bidding function 'b'.` provides a brief explanation of what the `maxBid` function does, which is to compute the highest bid given a bidding function `'b'`.

```

1      \begin{definition}\label{maxBid}
2      \leanok
3      \lean{maxBid}
4      \uses{Auction}
5      maxBid is set to compute the highest bid given a
        bidding function 'b'.
6      \end{definition}

```

The Blueprint tool also offers the capability to transform your LaTeX document into an interactive book, enhancing the readability and accessibility of complex formalizations. As shown in the figure, Blueprint organizes the document into a structured format with a detailed table of contents, allowing readers to easily navigate through chapters and sections. By clicking on individual elements, users can understand deeper specific definitions, theorems, and proofs. This format supports thorough exploration of the formalized content.

Auction Theory Blueprint	
1 Introduction	1 Introduction
2 Main result	2 Main result
2.1 First-Price and Second-Price Auction	2.1 First-Price and Second-Price Auction
2.5 Myerson's Lemma	2.5 Myerson's Lemma
2.7 Formalization Process	2.7 Formalization Process
3 Lean and Mathlib	3 Lean and Mathlib
3.1 The Lean Theorem Prover and Programming Language	3.1 The Lean Theorem Prover and Programming Language
3.2 The Lean Theorem Prover and Programming Language	3.2 The Lean Theorem Prover and Programming Language
4 Introduction	4 Introduction
5 Example	4.1 Main Definitions
6 Bibliography	4.2 Main Results
Dependency graph	4.3 Notations
	4.4 Implementation Notes
	4.5 References
	4.6 Tags
	5 Example
	6 Bibliography

Figure 3.3: Blueprint Book of the Formalization

3.3 Formalization of Auction Structure

3.3.1 Natural Language Statement of Auction Structure

In the setting of a single-item sealed bid auction, two primary concerns should be considered:

1. Who wins the item.
2. How much the winner should pay.

For instance, in a first-price auction, the item is awarded to the highest bidder, who then pays the amount they bid. In contrast, in a second-price auction, the item is still won by the highest bidder, but they pay the amount of the second highest bid. These setups represent the fundamental structures of these two auction types.

Further mechanism design is needed for the formalization of theorems and proof construction. This includes:

1. A specific number of bidders.
2. Payment rules and criteria for winning the auction.
3. Payoffs (utilities) for bidders.

3.3.2 Formal Statement of Auction Structure

The *Auction* structure is defined to represent the basic setting mentioned above of an auction.

Set of Participants 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 .

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

Listing 3.1: Definition of Auction structure with participants

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.


```
1    hF : Fintype I
```

Listing 3.2: Declaration of finite participants

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.

```
1    hI : Nontrivial I
```

Listing 3.3: Ensuring existence of participants

Valuation Function The function $v : I \rightarrow \mathbb{R}$ 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.

```
1    v : I → ℝ
```

Listing 3.4: Definition of the valuation function

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.

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

Listing 3.5: Attributes for Auction structure

3.4 First-price Auction

3.4.1 First-Price Auction

Definition 3.4 (First-Price Auction). In a first-price auction, each bidder submits a sealed bid of b_i . The bidder with the highest bid wins the item and pays the amount of their own bid b_i . [Rou20]

The formalization begins with the first-price auction. In the namespace `Auction`, the variable $a : \text{Auction}$ represents an instance of an auction, while $b : a.I \rightarrow \mathbb{R}$ is a function mapping each participant to their bid.

```

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

```

Listing 3.6: Namespace Declaration and Variable Definition for Auction

3.4.2 Formal Statement about Bid

The Maximum Bid

Definition 3.5 (Maximum Bid). In the auction structure, The Maximum Bid refers to the highest bid among all bids .[Rou20]

The definition of the maximum bid is the very first function 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 .

```

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

```

Listing 3.7: Definition of `maxBid` Function

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.

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.

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.

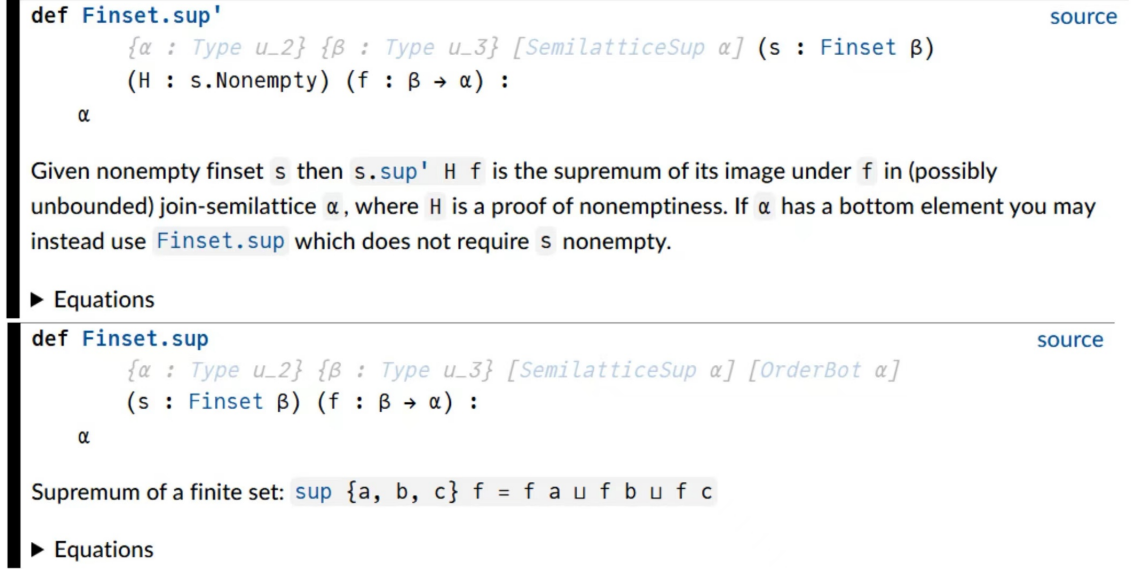


Figure 3.4: Theroem `Finset.sup'` and `Finset.sup` in Mathlib4

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`. 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.

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.

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

Listing 3.8: Existence of Maximum Bid Lemma

3.4.3 Formal Statement about Winner

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

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

Listing 3.9: Definition of Winner

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

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

Listing 3.10: Lemma: Bid of Winner Equals maxBid

Formalization using the Rewrite Tactic In the lemma `bid_winner_eq_maxBid`, the tactic `rw` is first time used. `rw` (rewrite) tactic can transform goals based on specified hypotheses.

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

Listing 3.11: Formalization using the Rewrite Tactic

To be more specific, `rw [bid_winner_eq_maxBid b]` is used to replace `b (winner b)` with `maxBid b` directly in the goal, reflecting the hypothesis that the winner's bid is the maximum bid in the auction.

```
1 /-rw is like rewrite, but also tries to close the goal by
   "cheap" (reducible) rfl afterwards -/
2 1 goal
3 a : Auction
4 b : a.I → ℝ
5 j : a.I
6 ⊢ b j ≤ b (winner b)
```

Listing 3.12: Goal before Applying `rw bid_winner_eq_maxBid b`

Applying `rw [bid_winner_eq_maxBid b]` substitutes `b (winner b)` with `maxBid b` in the proof goal. After applying the `rw` tactic, the goal state transforms as follows:

```
1 ⊢ b j ≤ b (winner b)
```

Listing 3.13: Goal State Before Transformation: Comparing Bid to Winner's Bid

becomes:

```
1 ⊢ b j ≤ maxBid b
```

Listing 3.14: Goal State After Transformation: Comparing Bid to Maximum Bid

The lemma `eq_winner_of_bid_gt` states that if the bid of bidder `i` is higher than all other bids, then `i` is the winner.

```
1 lemma eq_winner_of_bid_gt (i : a.I) (H : ∀ j , j ≠ i → b j
  < b i) : i = winner b := by
2   contrapose! H
3   exact ⟨winner b, H.symm, bid_le_bid_winner b i⟩
```

Listing 3.15: Lemma: Bid of Winner is Greater Than All Other Bids

3.4.4 Dominant Strategy

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

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

Listing 3.16: Definition of Dominant Strategy

3.4.5 Utility

The utility function defines the utility of the winner by using the difference between their valuation and the bid, while losers receive zero utility.

Definition 3.6 (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:

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

[Rou20]

The formal statement of the utility function first-price auction can be defined exactly the same as the natural language statement.

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

Listing 3.17: Definition of Utility in First-Price Auction

Under auction structure, we only need to define the utility function, as well as the lemmas `utility_winner` and `utility_loser` to formalized prove the theorem `firstprice_auction_has_no_dominant_strategy`.

```
1 lemma utility_winner (i : a.I) (H : i = winner b) : utility
   b i = a.v i - b i := by
2   rw [H]
3   simp only [utility, if_true]

5 lemma utility_loser (i : a.I) (H : i ≠ winner b) : utility
   b i = 0 := by
6   rw [utility]
7   simp only [H, if_false]
```

Here, `rw [H]` is used to replace instances of the variable i with `winner b` directly in the goal, reflecting the hypothesis $H : i = \text{winner } b$. This substitution simplifies the proof by aligning the utility expression with the condition specified in the hypothesis that bidder i is the winner.

```
0 /-rw is like rewrite, but also tries to close the goal by
   "cheap" (reducible) rfl afterwards -/
1 1 goal
```

```

2 a : Auction
3 b : a.I → ℝ
4 j : a.I
5 ⊢ b j ≤ b (winner b)

```

Listing 3.18: Goal before Applying `rw bid_winner_eq_maxBid b`

After applying the `rw` tactic, the goal state transforms as follows:

```

0 ⊢ utility b i = a.v i - b i

```

Listing 3.19: Goal State: Utility Computation for General Bidders

towards:

```

0 ⊢ utility b (winner b) = a.v (winner b) - b (winner b)

```

Listing 3.20: Goal State: Utility Computation for the Winner

This is a more direct example shows the simplification directly linking the winner's utility to their bid and valuation by using tactic `rw`.

3.4.6 First-Price Auction has no dominant strategy

Recall that the definition of dominant strategy, it is trivial to note that there is no dominant strategy in a first price auction.

Lemma 3.7 (First-Price Auction has no dominant strategy). In a sealed-bid first-price auction, no bidder has a dominant strategy.[Rou20]

This is saying that for every bidder i in the first-price auction, given a strategy $b_i : a.I \rightarrow \mathbb{R}$ and any other strategy b'_i , there exists a scenario where $\text{Utility}(b_i) < \text{Utility}(b'_i)$. This can be easily proved by consider the utility of a bidder.

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. ϵ be the least increment value in the auction. For bidder i to have a dominant strategy, b_i must maximize $\text{Utility}(b_i)$ for all possible b_j .

However, if $b_i > B$ lead to winning with a positive utility, $\text{Utility}(b_i) = v_i - b_i$, 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 $v_i \geq B$. $\text{Utility}(b'_i) = v_i - B + \epsilon$ which is greater than $\text{Utility}(b_i) = v_i - b_i$ 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

```

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

```

Listing 3.21: Formal Proof of First-Price Auction having No Dominant Strategy

Explanation of Function.update In the above proof, the `Function.update` used to defining a counterexample. This theorem allows modifying the value of a function at a specific point and creating a new function with one altered value.

```

def Function.update
  {α : Sort u} {β : α → Sort v} [DecidableEq α] (f : (a : α) → β a)
  (a' : α) (v : β a') (a : α) :
  β a

```

Replacing the value of a function at a given point by a given value.

► Equations

Figure 3.5: Theroem Function.update in Mathlib4

`Function.update` replaces the value of a function at a designated point by a new value. For function types in Lean, it constructs a new function identical to the original except at one point where it assumes a new value.

The goal state before applying the update is as follows. The proof seeks to show the existence of a strategy where the utility is not maximized.


```

0 a : Auction
1 b : a.I → ℝ
2 i : a.I
3 bi : ℝ
4 ⊢ ∃ x, ¬utility x i ≤ utility (Function.update x i bi) i

```

Listing 3.22: Goal before Applying Function.update

The function is updated to that the bidder i set the bid slightly below the intended amount, used to test the robustness of the supposed dominant strategy bi . Initially, all bidders are assigned $bi - 2$. `Function.update` is then applied to set this value for bidder i to $bi - 1$. Then we only need to check the assignment of bidding strategy can prove the non-dominant. The goal becomes.

```

0 ⊢ ¬utility (Function.update (fun x ↦ bi - 2) i (bi - 1)) i
    ≤
1    utility (Function.update (Function.update (fun x ↦ bi -
    2) i (bi - 1)) i bi) i

```

Listing 3.23: Goal after Applying Function.update

This state shows the utility comparison between two different bid strategies: $bi - 1$ and bi . Showing that the supposed dominant bid bi may not always yield the highest utility, hence not a dominant strategy in first-price auction.

Explanation of <;> In the proof of the theorem `firstprice_auction_has_no_dominant_strategy` the <;> tactic combinator is used in line 5 and line 6. This operator is designed to perform sequential tactic execution, where the first tactic is applied to the main goal, and the second tactic is then applied to each goal produced by the first tactic. This combinator is particularly useful for managing the flow of goals in a proof, ensuring that all necessary simplifications and strategic moves are made across all generated subgoals.

```

0 tac <;> tac' runs tac on the main goal and tac' on each
    produced goal, concatenating all goals produced by tac'.

```

Listing 3.24: Explanation of the <;> tactic combinator in Lean

The first part before `<;>`, we apply the `rw` tactic to rewrite definitions of utility function to ensures that all expressions in the proof are in their simplest form.

```
0   rw [utility, utility, if_pos (eq_winner_of_bid_gt _ i _),
      if_pos (eq_winner_of_bid_gt _ i _)]
```

Listing 3.25: Rewriting Utility Conditions Using `if_pos`

After these rewritings, the `<;>` combinator is used to adjust the main goal by rewriting the utility definitions according to specific conditions.

Then the two tactics are then applied to each of the goals generated by the first part. Here `simp [*]` aggressively simplifies each resulting subgoal using all available simplification rules.

```
0   intros
1   simp [*]
```

Listing 3.26: Applying Introduction and Simplification Tactics

This combination enormously ensures the fluency that every aspect of the goal transformed or split by the initial tactics, leading to a streamlined proof process that effectively tackles each component of the proof.

3.5 Second-price Auction

3.5.1 Second-Price Auction

Definition 3.8 (Second-Price Auction). In a second-price auction, each bidder submits a sealed bid of b_i . The bidder with the highest bid wins the item but pays the amount of the second highest bid submitted by another bidder.[Rou20]

To state the most important property of second-price auctions, firstly we define the function `maxBidExcluding` calculating the maximal bid of all participants but bidder i , to assist the formal statemet of the second-price in the structure.

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

Listing 3.27: Formal Definition of Maximum Bid Excluding Specified Bidder

The definition of `secondprice` represents the the second highest bid, which is the highest bid excluding the winner's bid.

```
0 noncomputable def Secondprice : ℝ := maxBidExcluding b
   (winner b)
```

Listing 3.28: Formal Definition of Second Highest Bid

3.5.2 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. This is actually trivial to understand in the natural language statement, but we have to highlight it again in the formal statement.

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

Listing 3.29: Lemma: Winner's Bid Greater or Equal to Second Highest

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

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

Listing 3.30: Lemma: Maximum Excluding Bid Less Than or Equal to Maximum Bid

Here, the lemma `maxBidExcluding_eq_maxBid_if_loser` shows that if the bidder is not the winner, then the highest bid excluding the bidder is equal to the highest bid.

The two lemma is formally stated to assist the formalization of the theorem Incentives in Second-Price Auctions.

```

0 lemma maxBidExcluding_eq_maxBid_if_loser {i : a.I} (H : i ≠
  winner b) :
1   maxBidExcluding b i = maxBid b := by
2   apply le_antisymm
3   · exact maxBidExcluding_le_maxBid b
4   · rw [← bid_winner_eq_maxBid b]
5     apply Finset.le_sup'
6     exact Finset.mem_erase_of_ne_of_mem H.symm
      (Finset.mem_univ (winner b))

```

Listing 3.31: Lemma: Max Bid Excluding Equals Max Bid if Bidder is Not Winner

3.5.3 Dominant Strategy

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.

3.5.4 Utility

In a second-price auction, the utility function and related lemmas capture the unique dynamics of this auction type.

The utility of each bidder depends not on both their own bid and the second highest bid if they win. This is the only difference between the utility of first-price and second-price auction. The utility function defines the utility of the winner as the difference between their valuation and the second highest bid, while losers receive zero utility.

Definition 3.9 (Utility in Second-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:

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

[Rou20]

```

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

```

Listing 3.32: Definition of Utility Function 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.

```

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

```

Listing 3.33: Lemma: Utility for Winner in Second-Price Auction

```

0 lemma utility_loser (H : i ≠ winner b) : utility b i = 0 :=
  by
1   rw [utility]; simp only [ite_false, H]

```

Listing 3.34: Lemma: Utility for Loser in Second-Price Auction

3.5.5 Incentives in Second-Price Auctions

Lemma 3.10 (Incentives in Second-Price Auctions). In a second-price auction, every bidder i has a dominant strategy: set the bid b_i equal to her private valuation v_i . [Rou20]

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). \square

```

0 theorem valuation_is_dominant (i : a.I) : dominant utility
  i (a.v i) := by
1   intro b
2   have key : maxBidExcluding (Function.update b i (a.v i))
      i = maxBidExcluding b i :=
3     (Finset.sup'_congr _ rfl fun j hj ↦ dif_neg
      (Finset.ne_of_mem_erase hj))
4   by_cases h1 : i = winner b
5   · rw [utility_winner b h1, Secondprice, ← h1, ← key]
6     by_cases h2 : i = winner (Function.update b i (a.v i))
7     · rw [utility_winner _ h2, sub_le_sub_iff_left,
      Secondprice, ← h2]
8     · rw [utility_loser _ h2, sub_nonpos,
      maxBidExcluding_eq_maxBid_if_loser _ h2]
9     conv_lhs => rw [← Function.update_same i (a.v i) b]
10    exact Finset.le_sup' _ (Finset.mem_univ i)
11  · rw [utility_loser b h1]
12    apply utility_nneg
13    apply Function.update_same

```

Listing 3.35: Theorem: Valuation as Dominant Strategy in Second-Price Auction

Hence in a second-price auction, participants are tend to bid their true valuations. This mechanism makes sure that the item will be awarded to the bidder who values it the most. Additionally, ensuring voluntary participation is crucial. Note it is impossible to force anyone to bid. Based on this we can establish the follow theorem.

3.5.6 Nonnegative Utility

Lemma 3.11 (Nonnegative Utility). In a second-price auction, every truthful bidder is guaranteed nonnegative utility.[Rou20]

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 $v_i - p \geq 0$. [Rou20] □

Utility in the second-price auction being non-negative when a bid equals the bidder's valuation is indeed a critical aspect of the mechanism's Dominant Strategy Incentive Compatibility (DSIC). DSIC is a fundamental property in auction theory that ensures that truth-telling (bidding an amount equal to one's true valuation of the item) is a dominant strategy. This means that no matter what strategies other bidders bid, a bidder maximizes their expected utility by bidding their true valuation.

This property is significant because it simplifies the strategic decision-making process for bidders. They do not need to calculate optimal bids based on guesses or estimates of other bidders' strategies. Instead, they can bid truthfully and still be assured of not receiving a negative utility outcome.

The theorem `utility_nneg` can be formalized as below.

```

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

Listing 3.36: Lemma: Non-Negative Utility in Second-Price Auction

3.5.7 Dominant Strategy Incentive Compatible

Definition 3.12 (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).

[Rou20]

Based on the above proof, we can conclude that the second-price auction is dominant strategy incentive compatible.

CHAPTER 4

MYERSON’S LEMMA

4.1 Single-Parameter Environments

An appropriate level of abstraction for stating Myerson’s lemma is through *single-parameter environments*. In such an environment, there are n agents. Each agent i possesses a private non-negative valuation v_i , representing her value *per unit of item* acquired. Additionally, there is a *feasible set* X . Each member of X is a non-negative n -vector (x_1, x_2, \dots, x_n) , where x_i signifies the *quantity of item* allocated to agent i . [Rou20]

4.1.1 Formal Statement of Single-Parameter Environment

A single-parameter environment includes a number n of agents and a feasible set X , where each element is a non-negative vector (x_1, x_2, \dots, x_n) , with x_i indicating the quantity of item allocated.

```

0  structure SingleParameterEnvironment where
1    -- The set of bidders
2    I : Type*
3    -- We require I to be nonempty
4    INonempty : Nonempty I
5    -- We require I to be finite
6    IFintype : Fintype I
7    -- The feasible set
8    feasibleSet : Set (I → ℝ)
9    -- We require the feasible set to be nonempty
10   feasibleSetNonempty : Nonempty feasibleSet

```

Listing 4.1: Definition of the SingleParameterEnvironment structure

4.1.2 Examples of Mechanisms in Single-Parameter Environments

Example 4.1 (Single-Item Auction) In a single-item auction (Section 2.1), X is the collection of 0-1 vectors containing no more than one 1—that is, $\sum_{i=1}^n x_i \leq 1$. [Rou20]

Example 4.2 (Sponsored Search Auction) In a sponsored search auction, X represents the set of n -vectors that correspond to the allocation of bidders to slots, ensuring each slot is assigned to a single bidder and each bidder gets at most one slot. If bidder i is allocated slot j , then the component x_i equals the click-through rate a_j of that slot.[Rou20]

Example 4.3 (Public Project) The decision to construct a public project, usable by everyone (such as a new bridge), can be modeled by the set $X = \{(0, 0, \dots, 0), (1, 1, \dots, 1)\}$. [Rou20]

Example 4.3 demonstrates that single-parameter environments are versatile enough to encompass applications beyond auctions. At this broad level of generality, we refer to participants as agents rather than bidders. The term *reports* is sometimes used in place of bids. A *mechanism* is a comprehensive procedure for decision-making when agents have private information (like valuations), while an *auction* specifically refers to a mechanism for the exchange of goods and money. Refer to Table 4.1 for more details.

Auction	Mechanism
Bidder	Agent
Bid	Report
Valuation	Valuation

Table 4.1: Correspondence of terms in auctions and mechanisms.

[Rou20]

4.2 Allocation and Payment Rules

Remember that a sealed-bid auction needs to make two primary decisions: who wins and what each participant pays. These decisions are captured by the *allocation rule* and the *payment rule*, respectively. Here are the steps involved:

1. Gather bids $b = (b_1, \dots, b_n)$ from all agents. The vector b is referred to as the **bid vector** or **bid profile**.
2. **Allocation rule:** Determine a feasible allocation $x(b) \in X \subseteq \mathbb{R}^n$ based on the bids.
3. **Payment rule:** Determine the payments $p(b) \in \mathbb{R}^n$ based on the bids.

Mechanisms of this sort are known as *direct-revelation mechanisms*, as agents are asked to disclose their private valuations directly in the first step. An example of an indirect mechanism is a progressive ascending auction.[Rou20]

Using our *quasilinear utility model*, in a mechanism with allocation and payment rules x and p , respectively, agent i obtains utility

$$u_i(b) = v_i \cdot x_i(b) - p_i(b)$$

when the bid profile is b .

We focus on payment rules that comply with

$$p_i(b) \in [0, b_i \cdot x_i(b)]$$

for every agent i and bid profile b . The constraint $p_i(b) \geq 0$ means the seller cannot pay the agents. The constraint $p_i(b) \leq b_i \cdot x_i(b)$ ensures a truthful agent receives nonnegative utility.

4.3 Statement of Myerson's Lemma

4.3.1 Implementable Allocation Rule

Definition 4.1 (Implementable Allocation Rule). An allocation rule x for a single-parameter environment is considered *implementable* if there exists a payment rule p such that the direct-revelation mechanism (x, p) is Dominant Strategy Incentive Compatible (DSIC).[Rou20]

Allocation rules that extend to DSIC mechanisms are termed implementable. In other words, the set of implementable rules is the projection of DSIC mechanisms onto their allocation rules. When designing a DSIC mechanism, we must restrict ourselves to implementable allocation rules, which define our “design space.” Thus, we can rephrase the question posed at the end of Lecture 2 as: is the welfare-maximizing allocation rule for sponsored search auctions, which assigns the i -th highest bidder to the i -th best slot, implementable?

Consider, for instance, a single-item auction (Example 4.1). Is the allocation rule that assigns the item to the highest bidder implementable? Indeed, we have already constructed a payment rule, the second-price rule, that makes it DSIC. What about the allocation rule

that assigns the item to the second-highest bidder? This case is less clear: while we haven't identified a payment rule that extends it to a DSIC mechanism, it is also challenging to argue that no payment rule could potentially work.

4.3.2 Formal Statement of Implementable Allocation Rule

An allocation rule is deemed implementable if there exists a payment rule that ensures the resulting direct-revelation mechanism is DSIC.

```

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

```

Listing 4.2: Definition of implementable allocation rule

4.3.3 Monotone Allocation Rule

Definition 4.2 (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 . [Rou20]

In a monotone allocation rule, bidding higher can only get you more stuff. For example, the single-item auction allocation rule that awards the item to the highest bidder is monotone: if you are the winner and you raise your bid (keeping other bids fixed), you continue to win. By contrast, awarding the item to the second-highest bidder is a non-monotone allocation rule: if you are the winner and you raise your bid high enough, you lose.

The welfare-maximizing allocation rule for sponsored search auctions (Example 4.2), with the i -th highest bidder awarded the i -th best slot, is monotone. When a bidder raises her bid, her position in the auction improves.

4.3.4 Formal Statement of Monotone Allocation Rule

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

```

0  @[simp]
1  def utility (v : E.I → ℝ) (b : E.I → ℝ) (i : E.I) : ℝ :=

```

```
2    v i * D.allocationRule b i - D.paymentRule b i
```

Listing 4.3: Definition of utility in a quasi-linear model

A dominant strategy for i is a strategy (i.e., a bid b_i) that is guaranteed to maximize i 's utility, no matter what the other bidders do; in other words, for any bids b and b' such that $b_i = b_i$, the utility from b should be not less than that of b' .

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

Listing 4.4: Definition of a dominant strategy

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. Goal for the definition `nondecreasing` is to define a monotone allocation rule.

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

Listing 4.5: Definitions of DSIC and nondecreasing

An allocation rule is monotone if replacing for every i , replacing the bid of i with something higher does not cause her to lose allocation.

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

Listing 4.6: Definition of monotone allocation rule

4.3.5 Myerson's Lemma

Lemma 4.3 (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.

[Rou20]

Myerson's lemma forms the cornerstone of our mechanism design theory. Part (a) asserts that Definitions 4.1 and 4.2 precisely characterize the same class of allocation rules. This equivalence is extremely potent: while Definition 4.1 outlines our design objective, it is cumbersome to utilize and verify, whereas Definition 4.2 is much more practical. Typically, determining whether an allocation rule is monotone is straightforward. Part (b) indicates that if an allocation rule is implementable, there is no uncertainty in determining the payments needed to achieve the DSIC property—there is only one correct method. Furthermore, part (c) provides a relatively simple and explicit formula for this payment rule.

CHAPTER 5

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, which is a substantial task compared to merely completing the formal statements of Auction Theory. However, as we studied deeper into the mathematical formalization and understood the relationship between Lean and Mathlib4 more clearly, we realized that we could try to contribute our formalized Auction Theory to the mathematics library of Lean4, mathlib4.

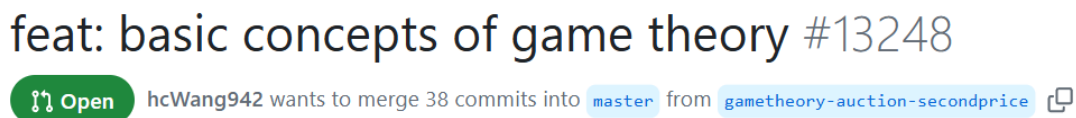


Figure 5.1: Files changed in pull request.

While working on a new contribution to mathlib, we should do this on a branch different from master. This can be achieved by giving an introduction on Zulip Chat and ask for write access to non-master branches of the mathlib repository. We cloned the copy of Mathlib4 in the local folder and connect it with the branch created named `gametheory-auction-secondprice`.

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`. In each push process, check for lint style, continuous integration(CI), files imported and so on, will automatically kick in. There will be a green tick on the line describing the most recent commit if everything works, a yellow circle if CI is still working, or a red cross if something went wrong. Click on the red cross to see details. The chcking process will be appears to be failed if one of the checking process went wrong.

This process proved to be more challenging than expected. In summary, we added 86 lines of comments. We have pushed 36 commit in the pull request in the new branch of Mathlib4.

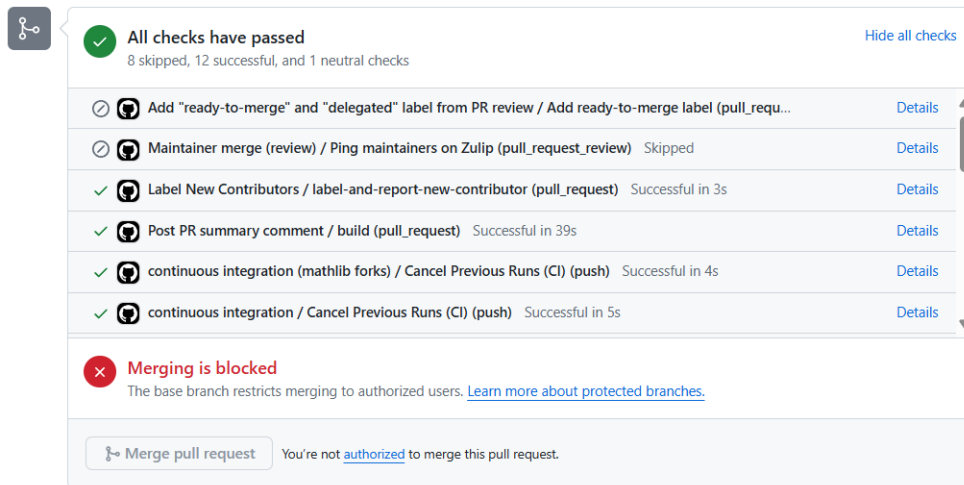


Figure 5.2: Automatically check flow in each process.

To ensure the coherence and readability of code, the total length of the code decreased from the original 353 lines to 203 lines. 4 files has been modified in my branch, awating the maintainers to check. Till today, we are still continuously refining and modifying the code in collaboration with the maintainers from mathlib.

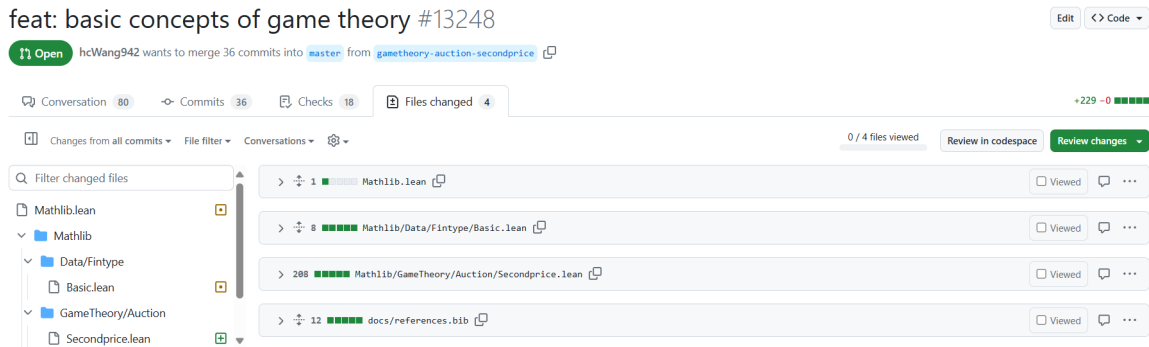


Figure 5.3: Files changed in pull request.

5.1 Preparation Work

5.1.1 Zulip Discussion

To make the process of contributing as smooth as possible, the document "How to contribute to mathlib" suggests using Zulip to discuss your contribution before and while we are working on it [Com23a] . Prior to this, during our participation in the NUS-hosted

Workshop on Formal Proofs and Lean, we had already discussed some aspects of contributing with Professor Kevin Buzzard.

Initially, we created a GitHub repository named *math-xmum* to store our existing code. We then submitted a project introduction on behalf of Xiamen University Malaysia and attached the repository link in a Zulip chat post. Everything went smoothly at first, and many developers provided helpful suggestions. For example, Josha Dekker commented Topological Standard Simplex is Compact is `isCompact_stdSimplex` in Mathlib.

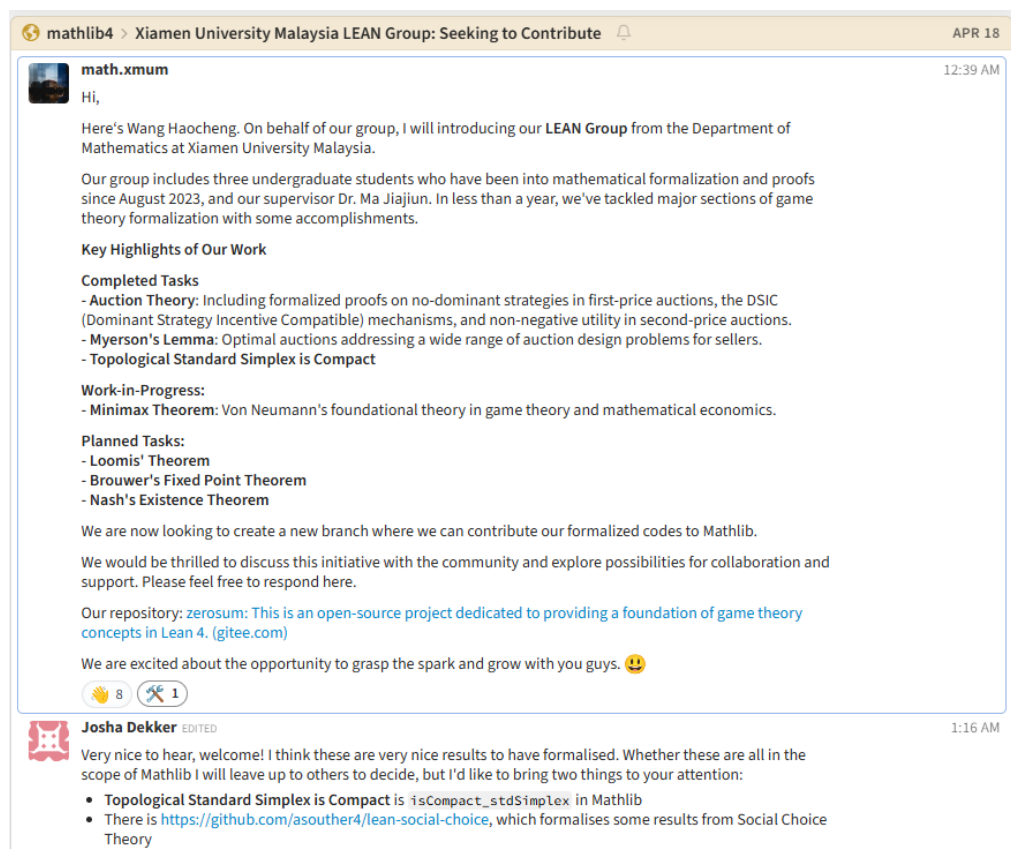


Figure 5.4: Zulip chat: Xiamen University Malaysia LEAN Group Seeking to Contribute [Com23i]

However, when we attempted to apply for contributor permissions using this GitHub account, our request was denied. The reason given was, it is a little strange to have a single GitHub account for everyone in your group, so that it is against the GitHub TOS. Following this, we applied for new contributor write access individually to facilitate the submission of pull requests.

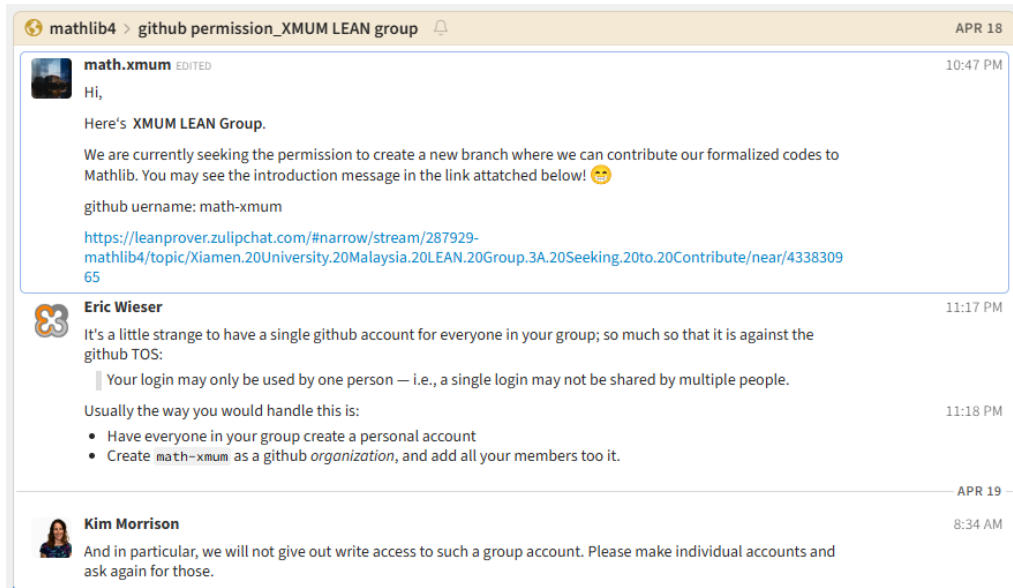


Figure 5.5: Zulip chat: github permission_XMUM LEAN group
[Com23c]

5.2 Code Style

5.2.1 Naming Conventions

As part of our integration into Mathlib4, it became imperative to align our code’s naming conventions with the established standards. The evolution of naming conventions from Lean 3 to Lean 4 has introduced a nuanced approach that we needed to adopt meticulously. In Lean 3, `snake_case` was predominantly used for all declarations. With the transition to Lean 4, Mathlib4 has embraced a mixed usage of `snake_case`, `lowerCamelCase`, and `UpperCamelCase`. [Com23g]

- Proofs and theorem names, being terms associated with Props, are consistently formatted in `snake_case`.
- Types such as inductive types, structures, and classes use `UpperCamelCase`, although there are notable exceptions where some fields might not follow this rule due to historical naming decisions or oversight.
- The naming of functions is determined by the type of their output, suggesting a name that reflects the return type, following an intuitive approach to identify function functionality at a glance.

- Non-function terms of Types typically use `lowerCamelCase` unless they are part of a structure or an inductive type named in `UpperCamelCase`, in which case they switch to `lowerCamelCase`.

Certain exceptions persist to maintain coherence within local contexts. For example, `Ne` is preferred over `NE` to maintain consistency with similar conventions like `Eq`. This flexibility allows for some level of discretion based on practical usage and historical context within the Mathlib4 community.

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.

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

Listing 5.1: Example of `snake_case` naming in theorem *valuation_is_dominant*

This definition illustrates the application of `snake_case` in theorem naming. It clearly indicates that `valuation_is_dominant` is a theorem sticking closely to the guidelines provided for Lean 4 contributors. [Com23g]

In addition to the naming conventions, our contributions to Mathlib4 must adhere to several other code style guidelines to ensure readability and maintainability. [Com23h]

5.2.2 Length of Code

One such guideline is the length of code lines. Lines should not exceed 100 characters. This limit helps in making the files more readable, especially on smaller screens or within smaller windows. For those using VS Code as their editor, a visual marker indicates when you have reached the 100 character limit. In a class or structure definition, fields are indented 2 spaces.

5.2.3 Header and Import Statement

Moreover, the file header must include copyright information, a list of contributors, and a brief description of the contents of file. Following the header, all import statements should be listed consecutively without a line break.

```

0  /-
1  Copyright (c) 2024 Wang Haocheng. All rights reserved.
2  Released under Apache 2.0 license as described in the file
   LICENSE.
3  Authors: Ma Jiajun, Wang Haocheng
4  -/
5  import Mathlib.Data.Fintype.Basic
6  import Mathlib.Data.Finset.Lattice
7  import Mathlib.Data.Real.Basic

```

Listing 5.2: File header and import statements

5.2.4 Module Docstring

Following the copyright header and the imports, it is essential to include a module docstring. This docstring, delimited with `'/-|` and `'-/`, should provide a title for the file, a summary of its contents (including main definitions, theorems, proof techniques, etc.), any specific notation used throughout the file, and references to relevant literature. This structured approach not only helps in organizing the code but also assists other developers and contributors in quickly understanding the file's purpose and navigating its contents effectively.

```

0  /-!
1  # Auction Theory

3  This file formalizes core concepts and results in auction
   theory. It includes the definitions of first-price and
   second-price auctions, as well as several fundamental
   results and helping lemmas.

5  ## Main Definitions

7  - 'Auction': Defines an auction with bidders and their
   valuations.

8  ...

```

```

10 ## Main Results

12 - 'utility_nneg': Utility is non-negative if the bid equals
    the valuation in second price auction.

14 ## References

16 * [T. Roughgarden, *Twenty lectures on Algorithmic Game
    Theory*][roughgarden2016]

18 ## Tags

20 auction, game theory, economics, bidding, valuation
21 -/

```

Listing 5.3: Module docstring

Meanwhile, we have to modify the file of reference reference.bib in Mathlib4 project to active the docstring.

Line	Code
2702	url = "https://math.stanford.edu/~vakil/216blog/"
2703	}
2704	
2705	+ @Book{ roughgarden2016,
2706	+ author = {Roughgarden, Tim},
2707	+ title = {Twenty lectures on Algorithmic Game Theory},
2708	+ publisher = {Cambridge University Press},
2709	+ year = {2016},
2710	+ pages = {11--23},
2711	+ isbn = {9781107172661},
2712	+ doi = {10.1017/cbo9781316779309},
2713	+ url = {https://doi.org/10.1017/CBO9781316779309}
2714	+ }
2715	+ }
2705	@Book{ rudin2006real,
2706	title = {Real and Complex Analysis},
2707	author = {Rudin, Walter},
3219	doi = {10.1090/S0002-9904-1937-06565-7},
3220	url = {https://doi.org/10.1090/S0002-9904-1937-06565-7}
3221	}
3230	doi = {10.1090/S0002-9904-1937-06565-7},
3231	url = {https://doi.org/10.1090/S0002-9904-1937-06565-7}
3232	}
3233	+ }

Figure 5.6: File modified in reference.bib in Mathlib4 project.

5.3 Making a Pull Request

A pull request is a proposal to merge a set of changes from one branch into another. Through a pull request, collaborators have the opportunity to examine and discuss the proposed changes before incorporating them into the primary codebase. Pull requests highlight the differences, known as diffs, between the content of the source branch and that of the target branch. [Git23]

As we have already asked for write access to non-master branches of the mathlib4 repository, in zulip chat. We shall directly make a pull request. The main goal is to push the changes to a branch on the main repository. We need to making a comment and header with tag feature (feat.), then add label for maintainers to manage the pull request.[Com23d]

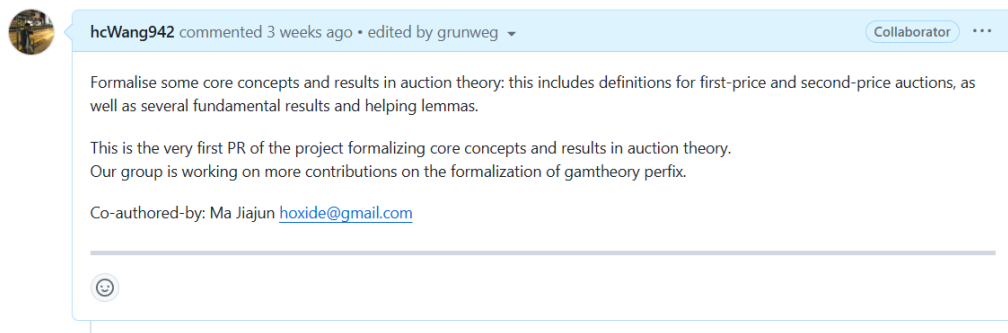


Figure 5.7: Comment and Header for pull request.

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

This is an example that the maintainer requesting to delete the declaration of variable `i`. We need to resolve the request and push again. In summary, this process is long-lasting but beneficial to refine the code to a more suitable version for merging.

5.4 Challenges in the Merging Process

Besides modifying the code style to meet the contribution level, we also encountered two obstacles during the merging process. The first is simplifying the theorem proof process. The second involves adding two additional theorems in the file `Mathlib.Data.Fintype.Basic` to support the pull request that formalizes some basic theorems in `GameTheory.AuctionTheory.Basic`.

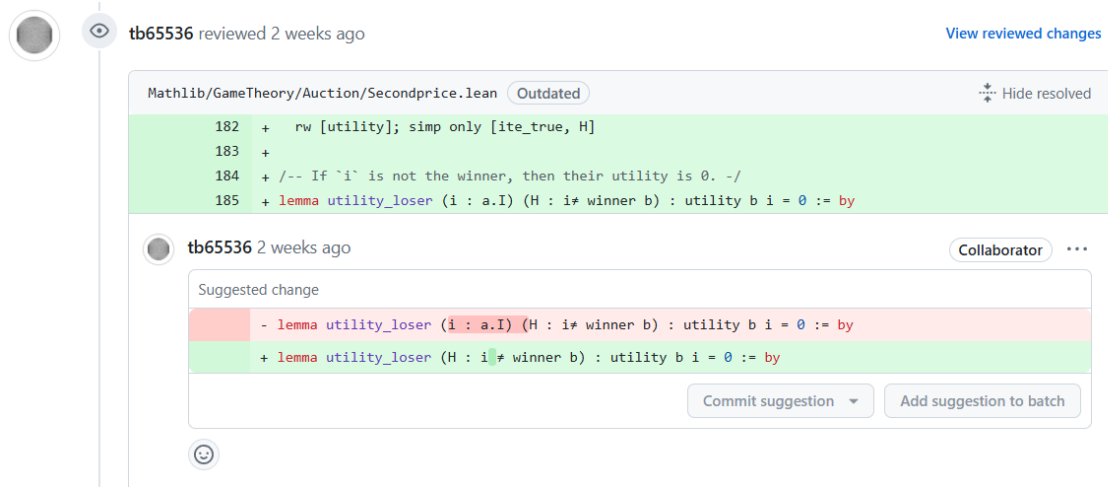


Figure 5.8: Conversation raised by the maintainer.

5.4.1 Proof Simplification

One major challenge in the merging process involves the simplification of theorem proofs. This task is crucial as it not only enhances the readability and maintainability of proofs but also aligns with the stringent quality standards of Mathlib. Simplification efforts may include optimizing the use of existing lemmas, employing more effective tactics, and restructuring proofs for greater efficiency.

For instance, consider the proof of `firstprice_auction_has_no_dominant_strategy`. This theorem asserts that in a first-price auction, there is no dominant bidding strategy for a bidder i with a bid b_i . The initial proof is provided below:

```
0 theorem firstprice_auction_has_no_dominant_strategy (i :
    a.I) (bi : ℝ) : ¬ dominant i bi := by
1   simp only [dominant, not_forall]
2   let b := fun j => if j = i then bi else bi - 2
3   let b' := fun j => if j = i then bi - 1 else bi - 2
4   use b, b'
5   simp only [exists_prop, ite_true, exists_const, true_and,
    b]
6   constructor
7   · intro j hj
8     simp only [if_false, hj]
9     exact (if_neg hj).symm
```

```

10 · have winner_b : i = winner b := by
11   apply eq_winner_of_bid_gt b i
12   intro j hj
13   simp [hj.symm, b]
14   have winner_b' : i = winner b' := by
15     apply eq_winner_of_bid_gt b' i
16     intro j hj
17     simp only [b, b', ite_true, hj.symm, ite_false,
18               gt_iff_lt, sub_lt_sub_iff_left]
18     exact one_lt_two
19   have h1 := utility_winner b i winner_b
20   have h2 := utility_winner b' i winner_b'
21   simp [h1, h2, b, b']

```

Listing 5.4: Initial proof of *firstprice_auction_has_no_dominant_strategy*

In this proof, we utilize the lemma `eq_winner_of_bid_gt` to simplify the goal, which states that if i 's bid is higher than all other bids, then i is the winner.

```

0 lemma eq_winner_of_bid_gt (i : a.I) (H : ∀ j , i ≠ j → b j
  < b i) : i = winner b := by
1   contrapose! H
2   exact ⟨winner b, H, b_winner_max b i⟩

```

Listing 5.5: Lemma *eq_winner_of_bid_gt* used in the initial proof

We made the following specific changes: The hypothesis H is changed from $\forall j, i \neq j \rightarrow b_j < b_i$ to $\forall j, j \neq i \rightarrow b_j < b_i$, which swaps the order of the variables in the inequality. This ensures consistency and simplifies the usage in subsequent proofs.

```

0 lemma eq_winner_of_bid_gt (i : a.I) (H : ∀ j , j ≠ i → b j
  < b i) : i = winner b := by
1   contrapose! H
2   exact ⟨winner b, H.symm, b_winner_max b i⟩

```

Listing 5.6: Modified lemma *eq_winner_of_bid_gt*

The proof of *firstprice_auction_has_no_dominant_strategy* can then be significantly simplified from 22 lines to just 3 lines as below.

```

0  theorem firstprice_auction_has_no_dominant_strategy (i :
    a.I) (bi :  $\mathbb{R}$ ) :  $\neg$  dominant utility i bi := by
1  rw [dominant, not_forall]
2  use Function.update (fun _  $\mapsto$  (bi - 2)) i (bi - 1)
3  rw [utility, utility, if_pos (eq_winner_of_bid_gt _ i _),
    if_pos (eq_winner_of_bid_gt _ i _)] <;> intros <;> simp
    [*]

```

Listing 5.7: Simplified proof of *firstprice_auction* has no dominant strategy

In this optimized proof, the usage of `eq_winner_of_bid_gt` is more concise and effective. The proof structure along with the updated lemma to directly achieve the goal with minimal lines of code. This approach not only improves the readability and maintainability of the code but also comply with the principles of efficient proof writing in Mathlib. By simplifying proofs in this manner, we ensure that the formalizations are more accessible and easier to verify, contributing to the overall quality and robustness of the Mathlib library.

5.4.2 Adding Theorems to Mathlib.Data.Fintype.Basic

Another challenge is the need to add two additional theorems to the file `Mathlib.Data.Fintype.Basic`.

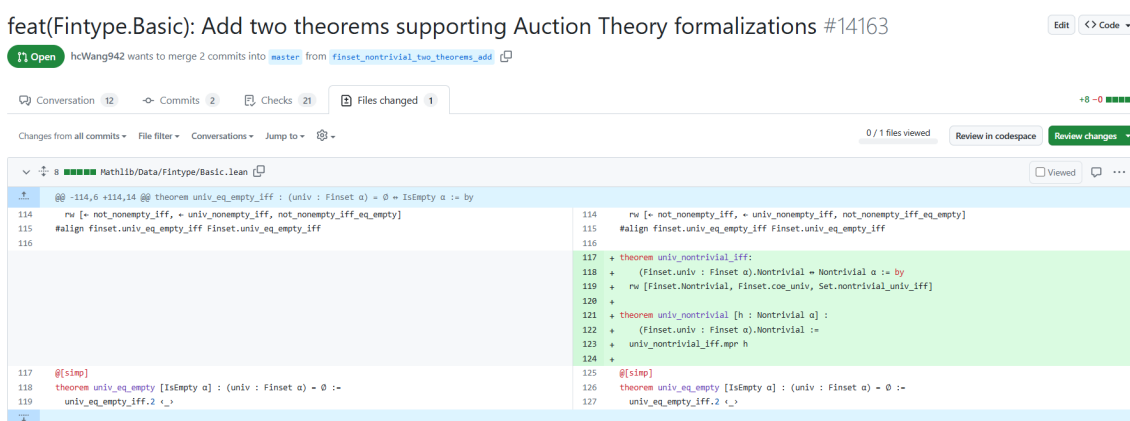


Figure 5.9: Add two theorems supporting Auction Theory formalizations

Original the two theorems was included in the pull request of auction theory. But maintainers from mathlib request me to could factor out both of these theorems into a separate PR and make the PR of auction theory dependent on the new one. That helps reduce the likelihood of merge conflicts.

This requirement arises as these theorems support the main pull request, which formalizes basic theorems in `GameTheory.AuctionTheory.Basic`. A noteworthy feature of one such theorem, `univ_nontrivial` is that it can take `Nontrivial α` automatically. For instance, see `univ_nonempty_iff` and `univ_nonempty` a few lines above, similar to `univ_nonempty_iff` and `univ_nonempty` already present in the codebase.

CHAPTER 6

CONCLUSION

6.1 The Significance and Challenges of Formalized Mathematics

6.1.1 The Significance of Formalized Mathematics

Formalized mathematics, unlike computer algebra systems designed for specific mathematical problems, provides a unified and universal framework rooted in modern logic. This allows for the rigorous verification of complex theorem proofs that span different mathematical domains and employ various techniques. This stems from the aforementioned distinction between form and meaning.

Once formalization is complete, proof assistants based on computer programs can thoroughly check all details of large-scale mathematical proofs, offering a level of certainty beyond human capability. This frees mathematical research from lengthy review cycles, as exemplified by Hales's project on the Kepler conjecture.

Mathematics is not just about symbolic deduction. a significant amount of a mathematician's work involves reading and writing mathematical explanations. This ensures that new work can effectively use existing mathematical conclusions and be understood within the overall mathematical framework. Formalized mathematics aims to formalize all branches of human mathematics, creating a unified digital database. This allows any proposition to be automatically traced and its prerequisite knowledge confirmed through programmatic analysis, including the concepts and lemmas needed to state and prove it. In traditional mathematics, academic papers often only briefly mention prior knowledge to keep the main points clear, assuming readers are already well-versed in the field. This prior knowledge needs to be traced back to previously published work or standard textbooks, which often emerge only after a field is well-established. Different mathematical texts frequently use varying notations and perspectives, complicating mathematics as a continuous and unified discipline. More seriously, some fields heavily rely on certain experts' understanding, and the mathematics community always worries that traditional areas lacking young mathematicians' attention might become lost arts with the passing of older mathematicians.

Formalized mathematics is seen as a crucial solution to these vital sustainability issues in human mathematics.

The unified digital mathematics library provided by formalized mathematics makes it easier to attempt modifications to foundational concepts and theorems. Modern mathematical development often brings about essential shifts in perspectives, typically involving modifications and generalizations of foundational concepts relied upon by many fields. Mathematicians are very concerned about how these foundational changes affect subsequent conclusions built upon them. With the aid of computer analysis, formalized mathematics can quickly compute which conclusions will fail under such changes, often providing mathematicians with fundamentally new insights into global mathematical problems. This benefit applies even to top mathematicians. For example, Scholze realized during the formalization of his theory of condensed mathematics that it clarified "what exactly makes the proofs work."

6.1.2 Challenges of Formalized Mathematics

Formalized languages have special syntactic rules, and encoding requires the correct use of predefined special names and symbols from the formalized mathematics library. The learning curve for mastering this technique is relatively steep.

Since formalized mathematics requires all involved concepts and assumed propositions to be fully encoded, and the mathematical techniques used in specific texts may differ from those commonly used in the unified mathematics library, significant time must be spent on equivalent substitutions for these techniques to be compatible with the unified framework.

Formalized mathematics has multiple concrete implementations, including the abstract design of formal systems and the specific engineering implementation of proof assistants. Different implementations affect the adaptability of mathematical branches; some branches are more challenging to handle under certain implementations.

6.2 Future vision of AI and Mathematics

6.2.1 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. In the vision of future, formalization of mathematics is highly beneficial for the training of Large Language Models (LLMs). The relationship between formalization of mathematics and AI can be viewed from two perspectives AI for Math and Math for AI.

6.2.2 AI for Mathematics: Integrating AI to assist Formalization

In recent years, the new paradigm of "AI for Science," where artificial intelligence empowers scientific research, has been flourishing. However, the application of artificial intelligence in the field of mathematics, particularly in theorem proving, remains full of challenges. Traditionally, mathematical formalization and theorem proving have been carried out by mathematicians, but this approach requires a substantial amount of time and effort and is prone to human oversights and errors. The development of artificial intelligence offers new opportunities for mathematical formalization and theorem proving.

To address the reasoning limitations in AI, the integration of proof assistants like Lean can play a crucial role. Proof assistants are designed to aid in the development of formal proofs by providing a structured framework that ensures the correctness of each step in a logical argument.

Integrating formalization of mathematics and proof assistants, AI systems can enhance accuracy by ensuring the precision and correctness of mathematical proofs and logical arguments, thereby reducing the likelihood of errors. They also improve rigor by enforcing a higher standard of reasoning, as every step of a proof must adhere to strict formal rules. Furthermore, proof assistants augment the capabilities of AI, enabling it to handle complex mathematical and logical tasks more robustly and reliably.

6.2.3 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 involves utilizing formal methods to verify the correctness of algorithms and systems, ensuring that they behave as intended under all conditions. Logic and proof theory apply principles from logic and proof theory to enhance the reasoning capabilities of AI, enabling it to construct and understand more complex arguments. Type theory is used to manage and ensure the consistency of data types within AI models, preventing type-related errors and enhancing the reliability of the system. Algorithmic game theory incorporates concepts from game theory to design AI systems that can make strategic decisions in competitive and uncertain environments.

By integrating these mathematical theories, AI systems can become more robust, reliable, and capable of sophisticated reasoning. This interdisciplinary approach not only addresses the current drawbacks of AI but also paves the way for future advancements in the field.

6.3 Large Language Models for Formalized Mathematics

6.3.1 Large Language Models for Formalized Mathematics

In modern interactive theorem proving, the human role involves guiding proof assistants with high-level proof commands, while the proof assistants complete the low-level details to form a complete proof. Large language models (LLMs) can partially replace humans in interactive theorem proving by planning high-level strategies through mathematical reasoning, thereby enhancing the automation of the entire system. In cases where LLMs are insufficient to generate correct strategies, humans only need to guide the model with natural language instructions to synthesize high-level proof commands. This transforms the human-computer interface from computer commands to more abstract and flexible natural language instructions, freeing human researchers from the complex commands of formalized mathematics.

LLMs with agent capabilities can achieve system-level automation, including autonomously planning the adaptations needed for formalizing a specific domain within an existing formalized mathematics library. They can also modify the existing mathematics library to improve its compatibility with more mathematical fields.

6.3.2 Formalized Mathematics for Large Language Models

Formalized mathematics provides effective supervision signals for LLMs to conduct long-chain accurate reasoning. Synthesizing large-scale formalized mathematics theorem proof data and incorporating it into the pre-training phase may enhance the reasoning capabilities of LLMs, similar to the success seen with the inclusion of code data.

Complex formalized proofs can be translated into natural language mathematical reasoning data, which has a solid guarantee of rigor in proof steps and scaling potential in complexity. This serves as a means to synthesize high-quality complex reasoning data.

Formalized mathematics systems can be invoked as tools during the reasoning phase, using symbolic systems to perform complex reasoning, aiding the model in tasks like the International Mathematical Olympiad (IMO) and other difficult computations and reasoning tasks.

Ideally, rather than merely using tools, the best scenario is the model achieving intrinsic alignment with formalized mathematics in its reasoning. This means that formalized mathematics serves as the implicit space for the model’s reasoning, with the model expressing itself in natural language while maintaining rigorous formal reasoning internally. This mode of operation uses formalization as the scaffolding of the mathematical edifice (used during construction and removed afterward), similar to the Bourbaki group’s approach of writing in an axiomatized manner while considering complete formalization possibilities, ensuring perfect rigor in their assertions.

6.3.3 Scenarios of Applying Large Language Models in Formalized Mathematics

In the short term, the most straightforward contribution of LLMs to formalized mathematics research could be fully functional code plugins. Due to the unique work style and challenges of formalized mathematics, such plugins at least need to:

- Possess the completion capabilities of current general code completion plugins. However, the latency requirements for such completions are less strict because the verification delays of proof assistants are already relatively high, and users in this field are quite tolerant of delays.

- Ensure a high degree of correctness in code completion due to the strict requirements of formalized systems. Tree search might be necessary here, or at least provide proof codes that are fundamentally correct and only require minor adjustments by human users.
- Flexibly and accurately understand existing codes, including both relatively stable standard mathematical libraries and temporary, frequently created codes in current projects. This implies that retrieval techniques or long-text inputs across files are almost essential.
- Have some agent capabilities to reasonably modify proof codes based on validator error messages or at least provide reasonable error attribution and modification suggestions to users.

In the long term, LLMs with fully autonomous formalized theorem proving capabilities could significantly advance mathematical research at a high level of automation. Feasible scenarios include:

- Strict formal verification of complex theorems that are difficult for humans to review, or at least the critical technical details within them.
- Participating in the daily work of human mathematicians, quickly automating the proof or disproof of conjectures in frontier research, thereby accelerating discoveries in advanced mathematics.
- Automating the modeling of applied mathematics problems in a formalized manner, providing alternative solutions for issues like supply chain optimization. This contrasts with current solutions, which involve months of modeling and solving by human experts, thus promoting the resolution of practical production problems.

The collaboration between AI and formal methods in mathematics represents a promising direction for overcoming current limitations and enhancing the capabilities of AI systems. Ideally, if an AI reaches superhuman levels of mathematical understanding, like AlphaGo, it could greatly advance the progress of mathematical sciences.

APPENDIX A

FORMAL STATEMENT OF AUCTION THEORY

PR for Mathlib4 can be view in the following link

- feat: basic concepts of game theory #13248
<https://github.com/leanprover-community/mathlib4/pull/13248>
- feat(Fintype.Basic): Add two theorems supporting Auction Theory formalizations #14163
<https://github.com/leanprover-community/mathlib4/pull/14163>

Github Repository math-xmum/gametheory can be view in the following link:

<https://github.com/math-xmum/gametheory>

```
0  /-
1  Copyright (c) 2024 Wang Haocheng. All rights reserved.
2  Released under Apache 2.0 license as described in the file
   LICENSE.
3  Authors: Ma Jiajun, Wang Haocheng
4  -/
5  import Mathlib.Data.Fintype.Basic
6  import Mathlib.Data.Finset.Lattice
7  import Mathlib.Data.Real.Basic

9  /-!
10 # Auction Theory

12 This file formalizes core concepts and results in auction
   theory. It includes the definitions of
13 first-price and second-price auctions, as well as several
   fundamental results and helping lemmas.

15 ## Main Definitions
```



```

17 - 'Auction': Defines an auction with bidders and their
    valuations.
18 - 'maxBid': Function that computes the highest bid given a
    bidding function.
19 - 'winner': Identifies the winner of the auction as the
    bidder with the highest bid.
20 - 'maxBidExcluding': Function that computes the highest bid
    excluding a given participant.
21 - 'dominant': Defines a dominant strategy in the context of
    a auction.
22 - 'Secondprice': Computes the second highest bid in the
    auction.
23 - 'Secondprice.utility': Computes the utility of each
    bidder based on the outcome of the auction.
24 - 'Firstprice.utility': Computes the utility for a first
    price auction.

26 ## Main Results

28 - 'utility_nneg': Utility is non-negative if the bid equals
    the valuation in second price auction.
29 - 'valuation_is_dominant': Bidding one's valuation is a
    dominant strategy in second price auction.
30 - 'first_price_has_no_dominant_strategy': There is no
    dominant strategy in first price auction.

32 ## Helping Lemmas

34 - 'eq_winner_of_bid_gt': If 'i's bid is higher than all
    other bids, then 'i' wins.
35 - 'exists_maxBid': There exists a participant whose bid
    matches the highest bid
36 - 'bid_winner_eq_maxBid': The winner's bid is the highest.
37 - 'winnerbid_ge_second': The winner's bid is at least the
    second highest bid.

```

```

38 - `Secondprice.utility_winner`: If `i` wins, utility is the
    valuation minus the second highest bid.
39 - `Secondprice.utility_loser`: If `i` does not win, their
    utility is 0.
40 - `bid_le_bid_winner`: The winner's bid is greater than or
    equal to all other bids.
41 - `maxBidExcluding_eq_maxBid_if_loser`: If `i` does not
    win, the highest bid excluding `i` equals
42 the highest bid.
43 - `Firstprice.utility_winner`: If `i` wins in a first price
    auction utility is valuation minus bid.
44 - `Firstprice.utility_loser`: If `i` does not win in a
    first price auction, utility is 0.

46 ## Implementation Notes

48 The structure and functions assume the existence of
    multiple bidders to allow for meaningful
49 auction dynamics. Definitions like `winner` and `maxBid`
    make use of Lean's `Finset` and `Classical`
50 logic to handle potential non-constructive cases
    effectively.

52 ## References

54 * [T. Roughgarden, *Twenty lectures on Algorithmic Game
    Theory*][roughgarden2016]

56 ## Tags

58 auction, game theory, economics, bidding, valuation
59 -/

61 open Classical

```

```

63 /-!### Structure Definition -/

65 /-- The Auction structure is set with components including:
    -/
66 structure Auction where
67   /-- A set of participants.-/
68   I : Type*
69   /-- A `Fintype` instance. -/
70   hF : Fintype I
71   /-- An `Nontrivial` instance. -/
72   hI : Nontrivial I
73   /-- A function mapping each participant to their
       valuation. -/
74   v : I → ℝ

76 attribute [instance] Auction.hF Auction.hI

78 namespace Auction

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

82 /-!### Helper Functions and Definitions -/
83 /-- Computes the highest bid given a bidding function `b`.
    -/
84 @[simp]
85 def maxBid : ℝ := Finset.sup' Finset.univ
       Finset.univ_nonempty b

87 /-- There exists a participant `i` whose bid equals the
       highest bid. -/
88 lemma exists_maxBid : ∃ i : a.I, b i = a.maxBid b := by
89   obtain ⟨i, _, h2⟩ := Finset.exists_mem_eq_sup'
       Finset.univ_nonempty b
90   exact ⟨i, symm h2⟩

```

```

92 /-- winner: The participant with the highest bid. -/
93 noncomputable def winner : a.I := Classical.choose
    (exists_maxBid b)

95 /-- The bid of the winner equals the highest bid. -/
96 lemma bid_winner_eq_maxBid : b (winner b) = maxBid b :=
    Classical.choose_spec (exists_maxBid b)

98 /-- The bid of the winner is always greater than or equal
    to the bids of all others. -/
99 lemma bid_le_bid_winner (j : a.I) : b j ≤ b (winner b) := by
100   rw [bid_winner_eq_maxBid b]
101   exact Finset.le_sup' b (Finset.mem_univ j)

103 /-- If 'i''s bid is higher than all other bids, then 'i' is
    the winner. -/
104 lemma eq_winner_of_bid_gt (i : a.I) (H : ∀ j , j ≠ i → b j
    < b i) : i = winner b := by
105   contrapose! H
106   exact ⟨winner b, H.symm, bid_le_bid_winner b i⟩

108 /-- A strategy is dominant if bidding 'bi' ensures that
109 'i''s utility is maximized relative to any other bids 'b'
    where 'b i = bi'. -/
110 def dominant (utility : (a.I → ℝ) → a.I → ℝ) (i : a.I)
    (bi : ℝ) : Prop :=
111   ∀ b, utility b i ≤ utility (Function.update b i bi) i

113 /-! ### Proofs and Lemmas -/

115 namespace Firstprice

117 /-- Computes the utility for a first price auction where
    the winner pays their bid. -/
118 noncomputable def utility (i : a.I) : ℝ := if i = winner b

```

```

    then a.v i - b i else 0

120 /-- If 'i' is the winner in a first price auction, utility
    is their valuation minus their bid. -/
121 lemma utility_winner (i : a.I) (H : i = winner b) : utility
    b i = a.v i - b i := by
122   rw [H]
123   simp only [utility, if_true]

125 /-- If 'i' is not the winner in a first price auction,
    their utility is 0. -/
126 lemma utility_loser (i : a.I) (H : i ≠ winner b) : utility
    b i = 0 := by
127   rw [utility]
128   simp only [H, if_false]

130 /-- Shows that there is no dominant strategy in a first
    price auction for any 'i' and bid 'bi'. -/
131 theorem firstprice_auction_has_no_dominant_strategy (i :
    a.I) (bi : ℝ) :
132   ¬ dominant utility i bi := by
133   rw [dominant, not_forall]
134   use Function.update (fun _ ↦ (bi - 2)) i (bi - 1)
135   rw [utility, utility, if_pos (eq_winner_of_bid_gt _ i _),
    if_pos (eq_winner_of_bid_gt _ i _)]
136   <|> intros <|> simp [*]

138 end Firstprice

140 /-- 'maxBidExcluding i' is the maximal bid of all
    participants but 'i'. -/
141 noncomputable def maxBidExcluding (i : a.I) : ℝ :=
    Finset.sup' (Finset.erase Finset.univ i)
142 (Finset.Nontrivial.erase_nonempty (Finset.univ_nontrivial))
    b

```

```

144 /--The second highest bid: the highest bid excluding the
    'winners bid.-/
145 noncomputable def Secondprice :  $\mathbb{R}$  := maxBidExcluding b
    (winner b)

147 namespace Secondprice

149 /-- The bid of the winner is always greater than or equal
    to the second highest bid. -/
150 lemma winnerbid_ge_second : Secondprice b  $\leq$  b (winner b) :=
    by
151   rw [bid_winner_eq_maxBid b]
152   exact Finset.sup'_mono b (Finset.subset_univ _)
153     (Finset.Nontrivial.erase_nonempty
      (Finset.univ_nontrivial))

155 /-- The bid of the winner is always greater than or equal
    to the second highest bid. -/
156 lemma maxBidExcluding_le_maxBid {i : a.I} (b : a.I  $\rightarrow$   $\mathbb{R}$ ):
    maxBidExcluding b i  $\leq$  maxBid b := by
157   apply Finset.sup'_mono
158   exact Finset.subset_univ (Finset.erase Finset.univ i)

160 /-- If 'i' is not the winner, then the highest bid
    excluding 'i' is equal to the highest bid. -/
161 lemma maxBidExcluding_eq_maxBid_if_loser {i : a.I} (H : i  $\neq$ 
    winner b) :
162   maxBidExcluding b i = maxBid b := by
163   apply le_antisymm
164   · exact maxBidExcluding_le_maxBid b
165   · rw [← bid_winner_eq_maxBid b]
166     apply Finset.le_sup'
167     exact Finset.mem_erase_of_ne_of_mem H.symm
      (Finset.mem_univ (winner b))

```

```

169 /-- Defines the utility of participant `i`,
170 which is their valuation minus the second highest bid if
    `i` is the winner, otherwise, it's 0. -/
171 noncomputable def utility (i : a.I) : ℝ := if i = winner b
    then a.v i - Secondprice b else 0

173 variable {i : a.I}
174 /-- If `i` is the winner, then their utility is their
    valuation minus the second highest bid. -/
175 lemma utility_winner (H: i = winner b) : utility b i = a.v
    i - Secondprice b := by
176   rw [utility]; simp only [ite_true, H]

178 /-- If `i` is not the winner, then their utility is 0. -/
179 lemma utility_loser (H : i ≠ winner b) : utility b i = 0 :=
    by
180   rw [utility]; simp only [ite_false, H]

182 /-- utility is non-negative if the bid equals the
    valuation. -/
183 lemma utility_nneg (i : a.I) (H : b i = a.v i) : 0 ≤
    utility b i := by
184   rcases eq_or_ne i (winner b) with rfl | H2
185   · rw [utility, if_pos rfl, ← H, bid_winner_eq_maxBid b,
        sub_nonneg, Secondprice]
186     exact maxBidExcluding_le_maxBid b
187   · rw [utility, if_neg H2]

189 /-- Proves that the strategy of bidding one's valuation is
    a dominant strategy for `i`. -/
190 theorem valuation_is_dominant (i : a.I) : dominant utility
    i (a.v i) := by
191   intro b
192   have key : maxBidExcluding (Function.update b i (a.v i))

```

```

    i = maxBidExcluding b i :=
193   (Finset.sup'_congr _ rfl fun j hj ↦ dif_neg
      (Finset.ne_of_mem_erase hj))
194 by_cases h1 : i = winner b
195 · rw [utility_winner b h1, Secondprice, ← h1, ← key]
196   by_cases h2 : i = winner (Function.update b i (a.v i))
197   · rw [utility_winner _ h2, sub_le_sub_iff_left,
      Secondprice, ← h2]
198   · rw [utility_loser _ h2, sub_nonpos,
      maxBidExcluding_eq_maxBid_if_loser _ h2]
199     conv_lhs => rw [← Function.update_same i (a.v i) b]
200     exact Finset.le_sup' _ (Finset.mem_univ i)
201 · rw [utility_loser b h1]
202   apply utility_nneg
203   apply Function.update_same

205 end Secondprice

207 end Auction

```


APPENDIX B

FORMAL STATEMENT OF MYERSON'S LEMMA

```
0 import Mathlib.Algebra.Order.Group.Abs
1 import Mathlib.Data.Real.Basic
2 import Mathlib.Data.Fintype.Basic
3 import Mathlib.Data.Fintype.Lattice
4 import Mathlib.MeasureTheory.Integral.IntervalIntegral
5 import Mathlib.MeasureTheory.Integral.Lebesgue
6 import Mathlib.Topology.MetricSpace.Basic

8 open Classical
9 open BigOperators Finset

11 /-- A single parameter environment has
12 - some number n of agents, and
13 - a feasible set X, in which each element is a nonnegative
    vector
14   `(x1, x2, . . . , xn)`,
15   where xi denotes the "amount of "stuff given. -/
16 structure SingleParameterEnvironment where
17   -- The set of bidders
18   I : Type*
19   -- We require I to be nonempty
20   INonempty : Nonempty I
21   -- We require I to be finite
22   IFintype : Fintype I
23   -- The feasible set
24   feasibleSet : Set (I → ℝ)
25   -- We require the feasible set to be nonempty
26   feasibleSetNonempty : Nonempty feasibleSet

28 instance (E : SingleParameterEnvironment) : Nonempty E.I :=
    E.INonempty
```

```

29 instance (E : SingleParameterEnvironment) : Fintype E.I :=
    E.IFintype
30 instance (E : SingleParameterEnvironment) : Nonempty
    (E.feasibleSet) :=
31   E.feasibleSetNonempty
32 instance (E : SingleParameterEnvironment) :
33   CoeFun E.feasibleSet (fun _ => E.I → ℝ) where coe f := f

35 -- Throughout let E denote a single-parameter environment.
36 namespace SingleParameterEnvironment

38 /-- A direct-revelation mechanism for a single-parameter
    environment
39 is formalized by an allocation rule and a payment rule. -/
40 structure DirectRevelationMechanism (E :
    SingleParameterEnvironment) where
41   allocationRule : (E.I → ℝ) → E.feasibleSet
42   paymentRule : (E.I → ℝ) → E.I → ℝ

44 section definitions
45 /- Henceforth let E be a single parameter environment and D
    be a direct
46 revelation mechanism on E. -/
47 variable {E : SingleParameterEnvironment} {D :
    DirectRevelationMechanism E}

49 /-- Quasi-linear utility -/
50 @[simp]
51 def utility (v : E.I → ℝ) (b : E.I → ℝ) (i : E.I) : ℝ :=
52   v i * D.allocationRule b i - D.paymentRule b i

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

```

```

56 bidders do; in other words, for any bids `b` and `b'` such
    that `b i = bi`,
57 the utility from `b` should be not less than that of `b'` -/
58 def dominant (v : E.I → ℝ) (bid_amount : ℝ) (i : E.I) :
    Prop :=
59   ∀ b b' : E.I → ℝ,
60   b i = bid_amount →
61   (∀ j : E.I, j ≠ i → b j = b' j) → @utility E D v b i ≥
    @utility E D v b' i

63 /-- A system is dominant-strategy incentive compatible
    (DSIC) if
64 truthful bidding is always a dominant strategy for every
    bidder and if
65 truthful bidders always obtain nonnegative utility. -/
66 def dsic := ∀ (i : E.I), ∀ (v : E.I → ℝ), @dominant E D v
    (v i) i

68 -- Goal here: Define a monotone allocation rule
69 def nondecreasing (f : ℝ → ℝ) := ∀ (x1 x2 : ℝ), x1 ≤ x2 →
    f x1 ≤ f x2

71 @[simp]
72 noncomputable def with_hole (f : E.I → ℝ) (i : E.I) (bi : ℝ)
    (j : E.I) : ℝ :=
73   if j = i then bi else f j

75 lemma filled_hole_retrieve {f : E.I → ℝ} {i : E.I} {bi : ℝ}
    :
76   with_hole f i bi i = bi := by
77     rw [with_hole]; simp

79 lemma filled_hole_retrieve_other
80   {f : E.I → ℝ} {i j : E.I} {hyp : i ≠ j} {bi : ℝ} :
81   with_hole f i bi j = f j := by

```

```

82   rw [with_hole, ite_eq_right_iff]
83   intro H
84   exfalse
85   exact hyp (symm H)

87 lemma filled_hole_almost_equal :
88    $\forall (j : E.I), j \neq i \rightarrow \text{with\_hole } b \ i \ x1 \ j = \text{with\_hole } b \ i$ 
       $x2 \ j := \text{by}$ 
89   intro j hyp
90   rw [filled_hole_retrieve_other]
91   rw [filled_hole_retrieve_other]
92   { symm; exact hyp }
93   { symm; exact hyp }

95 lemma almost_equal_fill_hole (b b' : E.I  $\rightarrow \mathbb{R}$ ) (i : E.I) :
96   ( $\forall (j : E.I), \neg j = i \rightarrow b \ j = b' \ j$ )  $\rightarrow \text{with\_hole } b \ i =$ 
       $\text{with\_hole } b' \ i := \text{by}$ 
97   intro hyp
98   funext x j
99   by_cases eq : j = i
100  { simp; split; rfl; rfl; }
101  { simp; split; rfl; exact hyp j eq }

103 lemma filled_hole_replace
104   {f : E.I  $\rightarrow \mathbb{R}$ } {i : E.I} {bi :  $\mathbb{R}$ } :
105   with_hole (with_hole f i bi) i = with_hole f i := by
106   funext bi' j
107   by_cases j = i
108   { simp; split; rfl; rfl }
109   { simp; split; rfl; rfl; }

111 lemma unfill_fill_hole {f : E.I  $\rightarrow \mathbb{R}$ } {i : E.I} : f =
      with_hole f i (f i) := by
112   funext x; simp; split; rename_i p; rw [p]; rfl

```

```

114 /-- An allocation rule is monotone if replacing for every
      i, replacing the
115 bid of i with something higher does not cause her to lose
      allocation. -/
116 def monotone (ar : (E.I → ℝ) → E.feasibleSet) :=
117   ∀ i : E.I,
118   ∀ b : E.I → ℝ,
119   nondecreasing (λ (bi : ℝ) => ar (with_hole b i bi) i)

121 /-- An allocation rule is implementable if
122 there is a payment rule such that the resulting
      direct-revelation mechanism
123 is DSIC. -/
124 def implementable (ar : (E.I → ℝ) → E.feasibleSet) :=
125   ∃ pr : (E.I → ℝ) → E.I → ℝ,
126   @dsic E {allocationRule := ar, paymentRule := pr}

128 end definitions

130 /- The remaining part of the file is used to prove
      Myersons' lemma,
131 we will show that
132 - An allocation rule is implementable iff it is monotone,
133 - if x is monotone, then there exists a unique payment rule
      (given by an
134 explicit formula) for which the direct revelation system is
      DSIC and bidders
135 that bid 0 pay 0. -/

137 section myerson

139 variable {E : SingleParameterEnvironment}

141 theorem payment_sandwich
142   (ar : (E.I → ℝ) → E.feasibleSet)

```

```

143   (p : (E.I → ℝ) → E.I → ℝ) (y z : ℝ):
144   @dsic E {allocationRule := ar, paymentRule := p}
145   → ∀ i : E.I,
146   z * (ar (with_hole b i y) i - ar (with_hole b i z) i)
147   ≤ p (with_hole b i y) i - p (with_hole b i z) i
148   ∧ p (with_hole b i y) i - p (with_hole b i z) i
149   ≤ y * (ar (with_hole b i y) i - ar (with_hole b i z) i) :=
      by
150   intro d i
151   have h1 :
152     y * (ar (with_hole b i z)) i ≤ y * (ar (with_hole b i
153     y)) i
154     - p (with_hole b i y) i + p (with_hole b i z) i
155     := by
156     have h : (with_hole b i y i = if i = i then y else 0) :=
157     by simp
158     specialize d i (fun j => if j = i then y else 0)
159     (with_hole b i y) (with_hole b i z) h
160     filled_hole_almost_equal
161     rw [utility] at d
162     simp at d
163     exact d
164   -- Set z to be the valuation of i here
165   have h2 :
166     z * (ar (with_hole b i y)) i ≤ z * (ar (with_hole b i
167     z)) i
168     - p (with_hole b i z) i + p (with_hole b i y) i
169     := by
170     have h : (with_hole b i z i = if i = i then z else 0) :=
171     by simp
172     specialize d i (fun j => if j = i then z else 0)
173     (with_hole b i z) (with_hole b i y) h
174     filled_hole_almost_equal
175     rw [utility] at d
176     simp at d

```

```

171     exact d
172     constructor; { linarith }; { linarith }

174 -- Goal here: Implementable → Monotone
175 theorem implementable_impl_monotone (ar : (E.I → ℝ) →
    E.feasibleSet) :
176   implementable ar → monotone ar := by
177   rintro ⟨p, impl⟩ i b x1 x2 xhyp
178   have := @payment_sandwich E b ar p x1 x2 impl i
179   have y : (x2 - x1) * (ar (with_hole b i x1) i - ar
    (with_hole b i x2) i) ≤ 0
180   := by linarith
181   by_cases l : x2 - x1 > 0
182   { have := nonpos_of_mul_nonpos_right y l
183     linarith }
184   { have : x1 = x2 := by linarith
185     rw [this] }

187 -- Will start wrangling with integrals here
188 open Monotone intervalIntegral

190 -- Goal here: "Explicit formula" works
191 -- TODO: Clean this proof up thoroughly
192 -- I don't have a better name for this, sorry
193 @[simp]
194 noncomputable def magic_payment_rule
195   (ar : (E.I → ℝ) → E.feasibleSet) (b : E.I → ℝ) (i :
    E.I) : ℝ :=
196   (b i) * ar b i - ∫ t in (0)..(b i), (fun t' => ar
    (with_hole b i t') i) t

198 @[simp]
199 noncomputable def with_magic (ar : (E.I → ℝ) →
    E.feasibleSet)
200   : DirectRevelationMechanism E :=

```

```

201   { allocationRule := ar, paymentRule := magic_payment_rule
      ar }

203 def utility_exp {v : E.I → ℝ} (b : E.I → ℝ) :
204   @utility E (with_magic ar) v b i
205     = (v i - b i) * ar b i
206     + ∫ x in (0)..(b i), (fun t' => ar (with_hole b i t')
      i) x := by
207       rw [utility]; simp; ring_nf

209 theorem magic_payment_rule_works (ar : (E.I → ℝ) →
      E.feasibleSet)
210   : (monotone ar) → @dsic E (with_magic ar) := by
211   -- Suppose 'ar' is monotone and let 'i' be the bidder in
      consideration.
212   -- Let 'v' be the valuation of the bidders.
213   -- Let 'b' and 'b'' be bids such that 'b j = b' j' for
      all 'j ≠ i',
214   -- and 'b i = v i'.
215   intro mon i v b b' b_i_eq_v_i almost_eq
216   push_neg at almost_eq
217   -- The goal now is to show that 'utility v b i ≥ utility
      v b' i'.

219   -- We establish a bunch of integrability statements here,
      no content here
220   have func_is_monotone : Monotone (fun x => ar (with_hole
      b' i x) i) := by
221     rw [Monotone]; intro x y h; exact mon i b' x y h
222   have b_i_to_b'_i : IntervalIntegrable
      (fun x => ar (with_hole b' i x) i) MeasureTheory.volume
      (b i) (b' i) := by
223     exact intervalIntegrable func_is_monotone
224   have zero_to_b_i : IntervalIntegrable

```



```

226   (fun x => ar (with_hole b' i x) i) MeasureTheory.volume
      0 (b i) := by
227   exact intervalIntegrable func_is_monotone
228   have zero_to_b'_i : IntervalIntegrable
229     (fun x => ar (with_hole b' i x) i) MeasureTheory.volume
      0 (b' i) := by
230   exact intervalIntegrable func_is_monotone
231   have const_to_int :
232      $\int x \text{ in } (b' i)..(b i), (\text{fun } _ \Rightarrow \text{ar } b' i) x = (b i - b' i) * \text{ar } b' i$  := by
233     rw [integral_const]; simp

235   -- We will compute 'utility v b' i - utility v b i' by
      cases.
236   suffices h : utility v b' i - utility v b i  $\leq$  0 by
      linarith
237   repeat rw [utility_exp]
238   rw [← b_i_eq_v_i]
239   ring_nf
240   rw [almost_equal_fill_hole b b' i almost_eq, ← sub_mul]
241   rw [integral_interval_sub_left zero_to_b'_i zero_to_b_i]
242   rw [← const_to_int]
243   rw [integral_symm]
244   rw [← integral_neg, ← integral_add (by simp) b_i_to_b'_i]

246   -- It remains to show that
247   --  $\int (x : \mathbb{R}) \text{ in } b i..b' i,$ 
248   --  $-(\text{fun } x \mapsto \uparrow(\text{ar } b') i) x + \uparrow(\text{ar } (\text{with\_hole } b' i x)) i \leq 0$ 
249   -- We proceed by cases.
250   by_cases limit_ineq : b' i  $\leq$  b i
251   { rw [integral_symm, ← integral_neg]
252     simp
253     have :  $\forall u \in \text{Set.Icc } (b' i) (b i),$ 

```

```

254      0 ≤ (fun x => (ar (with_hole b' i x)) i - (ar b') i) u
:= by
255      intro x; simp
256      intro le _
257      rw [@unfill_fill_hole E b' i, filled_hole_replace]
258      exact mon i b' (b' i) x le
259      have := @integral_nonneg
260      (fun x => (ar (with_hole b' i x)) i - (ar b') i)
261      (b' i) (b i) MeasureTheory.volume limit_ineq this

263      have : (-∫ u in (b' i)..(b i),
264      (fun x => (ar (with_hole b' i x)) i - (ar b') i) u) ≤
0
265      := by linarith
266      rw [← integral_neg] at this
267      simp at this
268      have t :
269      (fun x => (ar b') i - (ar (with_hole b' i x)) i)
270      = (fun x => -(ar (with_hole b' i x)) i + (ar (b')
i))
271      := by funext x; linarith
272      rw [t] at this
273      exact this }
274 { simp
275      have : ∀ u ∈ Set.Icc (b i) (b' i),
276      0 ≤ (fun x => - (ar (with_hole b' i x)) i + (ar b') i)
u := by
277      intro x; simp
278      intro _ ge
279      rw [@unfill_fill_hole E b' i, filled_hole_replace]
280      exact mon i b' x (b' i) ge
281      have := @integral_nonneg
282      (fun x => - (ar (with_hole b' i x)) i + (ar b') i)
283      (b i) (b' i) MeasureTheory.volume (by linarith) this

```

```

285     have : (- ∫ u in (b i)..(b' i),
286       (fun x => - (ar (with_hole b' i x)) i + (ar b' i) u)
287       ≤ 0
288       := by linarith
289     rw [← integral_neg] at this
290     simp at this
291     exact this }

292 theorem magic_payment_bid_zero_implies_payment_zero
293   (ar : (E.I → ℝ) → E.feasibleSet) :
294   ∀ b : E.I → ℝ, ∀ i : E.I, b i = 0 → magic_payment_rule
295     ar b i = 0 := by

296   intro b i hyp; rw [magic_payment_rule, hyp]; simp

297   -- Goal here: Works → "Explicit formula"
298   -- TODO: figure out a proof and then finish this

299 theorem magic_payment_rule_unique (ar : (E.I → ℝ) →
300   E.feasibleSet)
301   : ∀ p q : ((E.I → ℝ) → E.I → ℝ),
302   (monotone ar)
303   → @dsic E {allocationRule := ar, paymentRule := p}
304   → @dsic E {allocationRule := ar, paymentRule := q}
305   → (∀ b : E.I → ℝ, ∀ i : E.I, b i = 0 → p b i = 0)
306   → (∀ b : E.I → ℝ, ∀ i : E.I, b i = 0 → q b i = 0)
307   → p = q := by
308   intro p q _ dp dq hyp hyq
309   funext b i

310   -- Set d = p - q.
311   set d := p - q
312   -- It suffices to show that d b i ≤ ε for all ε ≥ 0.
313   suffices : ∀ ε > 0, |d b i| ≤ ε
314   { exact eq_of_forall_dist_le this }

315   -- Therefore let ε ≥ 0.

```

```

317   intro  $\varepsilon$  h $\varepsilon$ 
318   -- For notational simplicity let c _ := d (with_hole b i
      _ ) i
319   set c := fun (y :  $\mathbb{R}$ ) => d (with_hole b i y) i
320   -- For notational simplicity let c' _ := ar (with_hole b
      i _ ) i
321   set c' := fun (y :  $\mathbb{R}$ ) => ar (with_hole b i y) i

323   have : d b i = c (b i) := by simp; conv => lhs; rw
      [@unfill_fill_hole E b i]
324   rw [this]

326   have useful :  $\forall y z : \mathbb{R}, y \geq z \rightarrow |c y - c z| \leq (y - z) *
      (c' y - c' z)$  := by
327     intro y z _; rw [abs_le]
328     obtain ⟨h1, h2⟩ := @payment_sandwich E b ar p y z dp i
329     obtain ⟨h3, h4⟩ := @payment_sandwich E b ar q y z dq i
330     simp; constructor <;> linarith

332   have c_zero_is_zero : c 0 = 0 := by
333     specialize hyp (with_hole b i 0) i (by simp)
334     specialize hyq (with_hole b i 0) i (by simp)
335     simp; rw [hyp, hyq]; simp

337   -- deal first with the situation where the allocation is
      equal.
338   by_cases r : c' (b i) - c' 0 = 0
339   { by_cases b i  $\geq$  0
340     { specialize useful (b i) 0 (by assumption)
341       rw [c_zero_is_zero, r] at useful; simp at useful
342       simp; rw [useful]; simp; linarith }
343     { specialize useful 0 (b i) (by linarith)
344       have : c' 0 - c' (b i) = 0 := by linarith
345       rw [c_zero_is_zero, this] at useful; simp at useful
346       simp; rw [abs_sub_comm, useful]; simp; linarith }}

```

```

348 -- Let N be large so that the bottom holds. This makes
      sense since we can
349 -- assume the denominator is not 0.
350 obtain ⟨N, Nhyp⟩ := exists_nat_ge (|b i| * |c' (b i) - c'
      0| / ε)

352 have c'_diff_gt_zero : 0 < |c' (b i) - c' 0| := by
353   rw [lt_abs]; push_neg at r; rw [ne_iff_lt_or_gt,
      or_comm] at r
354   conv => rhs; rw [lt_neg, neg_zero]
355   conv => lhs; rw [← GT.gt]
356   exact r

358 have N_gt_zero : N > 0 := by
359   rw [div_le_iff hε] at Nhyp
360   have : b i ≠ 0 := by by_contra r'; rw [r'] at r; simp
      at r
361   have : 0 < |b i| * |c' (b i) - c' 0| := by
362     apply mul_pos
363     { rw [lt_abs, or_comm]
364       conv => left; rw [lt_neg]; simp
365       have := ne_iff_lt_or_gt.mp this
366       exact this }
367     { exact c'_diff_gt_zero }
368   have : 0 < N * ε := by linarith
369   have := (mul_pos_iff_of_pos_right hε).mp this
370   simp at this
371   exact this

373 have sane : N * (b i / N) = b i := by
374   rw [← mul_div_assoc, mul_comm]
375   apply mul_div_cancel
376   simp
377   linarith

```

```

380 have : c (b i) = c (N * (b i / N)) - c (0 * (b i / N)) :=
    by
381   rw [sane]
382   have : 0 * (b i / N) = 0 := by ring_nf
383   rw [this, c_zero_is_zero]
384   simp
385 rw [this]

387 have := sum_range_sub (fun l => c (l * (b i / N))) N
388 simp at this; simp
389 rw [← this]

391 have := Finset.abs_sum_le_sum_abs
392   (fun (l : ℕ) => c ((l + 1) * (b i / N)) - c (l * (b i /
    N))) (range N)
393 simp at this; apply le_trans this

395 have : ∀ x ∈ range N, |c ((x + 1) * (b i / N)) - c (x *
    (b i / N))|
396   ≤ (b i / N) * (c' ((x + 1) * (b i / N)) - c' (x * (b i
    / N))) := by
397   intro x _
398   by_cases r : b i ≥ 0
399   { have : (x + 1) * (b i / N) ≥ x * (b i / N) := by
400     ring_nf; simp; rw [← div_eq_mul_inv]; exact
    div_nonneg r (by linarith)
401     specialize useful ((x + 1) * (b i / N)) (x * (b i /
    N)) this
402     apply le_trans useful; ring_nf; simp }
403   { push_neg at r
404     have : x * (b i / N) ≥ (x + 1) * (b i / N) := by
405       ring_nf; simp; rw [← div_eq_mul_inv]

```

```

406         exact div_nonpos_of_nonpos_of_nonneg (by linarith)
         (by linarith)
407     specialize useful (x * (b i / N)) ((x + 1) * (b i /
N)) this
408     rw [abs_sub_comm]
409     apply le_trans useful; ring_nf; simp }
410 have := Finset.sum_le_sum this
411 apply le_trans this
412 rw [← Finset.mul_sum]
413 have := sum_range_sub (fun x => c' (x * (b i / N))) N
414 simp
415 simp at this
416 rw [this, sane, div_mul_eq_mul_div]
417 apply (div_le_iff _).mpr
418 apply (div_le_iff hε).mp at Nhyp
419 conv => right; rw [mul_comm]
420 apply le_trans' Nhyp
421 rw [← abs_mul]
422 simp
423 apply le_trans' (le_abs_self _)
424 rfl
425 simp
426 assumption

```

REFERENCES

- Jeremy Avigad. Practical foundations for formal methods - slides, 2019. Lecture Slides.
- Riccardo Brasca. Luxembourg 2021 presentation, 2021. Conference Presentation.
- The Lean Community. Contribution guide, 2023. Website Section.
- The Lean Community. Functional programming in lean, 2023. Online Documentation.
- The Lean Community. Github permission discussion for xnum lean group, 2023. Topic on Lean Prover Community Zulip Chat.
- The Lean Community. How to contribute to lean, 2023. Contribution Guidelines on the Lean Community Website.
- The Lean Community. The lean programming language, 2023. Official Website.
- The Lean Community. Mathlib4: The lean mathematical library, 2023.
- The Lean Community. Naming conventions, 2023. Contribution Guidelines on the Lean Community Website.
- The Lean Community. Style guide, 2023. Contribution Guidelines on the Lean Community Website.
- The Lean Community. Zulip chat search: Xiamen, 2023. Lean Prover Community Zulip Chat.
- GitHub. About pull requests, 2023. GitHub Documentation.
- Patrick Massot. Lean blueprint, 2023. GitHub Repository.
- Tim Roughgarden. *Twenty Lectures on Algorithmic Game Theory*. Cambridge University Press, 2020.
- Terence Tao. Formalizing the proof of pfr in lean4 using blueprint: A short tour, 2023. Blog Post.
- Max Zipperer. Using lean in the mathematics classroom, 2022. Master’s Thesis.