THE UNIVERSITY OF CHICAGO

PROVING ARROW'S IMPOSSIBILITY THEOREM USING REFINEMENT TYPES

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
BENJAMIN WALDMAN

CHICAGO, ILLINOIS
MAY 2024

TABLE OF CONTENTS

ACKNOWLEDGMENTS

I would like to thank my advisor, Stuart Kurtz, for guiding me throughout this process and for teaching the class that initially piqued my interest in formal verification. I would like to thank Robert Rand and Adam Shaw for being on the committee and always having time to meet with me to talk through this project. I would like to thank my friends for supporting me and for listening to me ramble about Arrow's Theorem to no end. I would like to especially thank Kanchan Naik for suggesting that I execute the code I wrote for this project. I would like to thank Benjamin Caldwell for putting up with me during many Type Theory and Programming Proofs office hours. Lastly, I would like to thank my parents for their endless support.

ABSTRACT

Arrow's impossibility theorem is a landmark result in theoretical economics that led to the formation of modern social choice theory. I develop a new, computer-verified proof of Arrow's impossibility using refinement types, a concept from type theory that allows functions to return both data and predicates on that data. I then argue that refinement types are a natural paradigm for formally verifying many theorems in economics and political science.

CHAPTER 1
INTRODUCTION

In his 1950 paper "A Difficulty in the Concept of Social Welfare," Kenneth Arrow [1] proved that all systems to aggregate group preferences violate at least one (and possibly multiple) seemingly innocuous assumptions about how group preferences ought to be aggregated. When Arrow received the Nobel Prize in Economics, the Swedish Academy of Sciences described this theorem as "perhaps the most important of [Arrow's] many contributions to welfare theory." [8]

However, like most important social science results, Arrow's Theorem has been largely ignored as a target for formal verification. This is unfortunate, as the social sciences generally and social choice theory specifically are excellent targets for computer-verified proof. Many of the concepts involved in social choice theory (e.g. preference relations) are easy to define and reason about in proof assistants.

Furthermore, many proofs of Arrow's theorem are disappointingly unclear, including some of the most cited in the literature. For example, Amartya Sen's proof [7] notes that "We are cutting a corner here by assuming that a, b, x, and y are all distinct social states. The reasoning is exactly similar when two of them are the same alternative." This characterization of those edge cases is debatable at best. As seen in `LemmaFour` in `Arrow.Agda`, the reasoning for these edge cases is significantly different than the main case and was a significant barrier in completing a verification of Sen's proof. It is worth highlighting again how prominent Arrow's Theorem is: it is considered one of the most important theoretical results in economics. If even celebrated proofs of Arrow's Theorem suffer from vagueness, it is likely that many lesser known results could be substantially clarified by formal verification. Theoretical economics is thus both a technically feasible and academically rewarding site for work in formal verification.

However, there exist meaningful stumbling blocks to verification of social scientific theorems. While many of the basic concepts involved in social scientific proofs are easy to reason about in a proof assistant, many of the proof techniques used are not. For example, many hand-written proofs of Arrow's theorem [5] involve pictures that show how one can alter a list of candidates representing a voter's preferences in a certain way while maintaining a certain desired property. While these intermediate results are easy to see visually, the Agda type checker does not have eyes (as of version 2.7), and formalizing these pictures within a type system is not intuitive.

The limited verifications of Arrow's theorem that do tend to rely heavily on proof automation techniques that are difficult if not impossible for human eyes to parse. In addition, these proofs tend to be extremely long and difficult to compartmentalize into lemmas because the automation techniques require the context provided by the prior parts of the proof to solve the goal. For example, Peter Gammie [4] notes in his verification of Arrow's theorem in the proof assistant Isabelle that his proof "is unfortunately long. Many of the statements rely on a lot of context, making it difficult to split it up."

In sum, the vast majority of computer verified proofs of social scientific results are painful to produce and confusing to read. In this dissertation, I will propose a new paradigm for formally verifying social scientific results and prove its usefulness by producing a novel verification of Arrow's impossibility[1] theorem that does not contain any proof automation.

1. See https://github.com/bmwaldman0918/ArrowsTheorem/ for the completed proof.

CHAPTER 2

MATHEMATICAL LOGIC

Prior formalizations of Arrow's Theorem have been written in the language of mathematical logic [4]. This approach has a variety of obvious benefits: it allows extensive proof automation via the use of tactics optimized for logical reasoning (e.g. Isabelle's `blast`). Furthermore, it is the language that hand-written proofs of Arrow's Theorem are written in and that mathematicians tend to be most comfortable with. However, these proofs have a few obvious drawbacks.

First, proofs that rely heavily on automation are not particularly interesting or useful. Arrow's Theorem is a well-known result that has been proven many times in many ways. Formalizing Arrow's Theorem is an interesting project because new formalizations may provide us with new proof techniques or clarify certain aspects of the proof, not because there exists any meaningful doubt about the veracity of the conclusion. Heavily automated proofs hide most of the interesting work behind the automated tactic, making it extremely difficult to see how the proof obligation is actually fulfilled.

Furthermore, some proof automation techniques rely on large code bases that are hard to verify. This can lead to a solver 'proving' a goal incorrectly [11]. While these bugs are incredibly rare, they do exist, and the potential for their existence weakens one of the primary motivations for formal verification.

While automation has its drawbacks, it's easy to see why it is so heavily used: a proof that only used mathematical logic and no proof automation would likely be thousands of lines of mostly uninteresting code. Producing each of the ten or so witnesses required and proving they have every necessary property would be incredibly tedious.

Luckily, there is a better way. One fundamental idea in proof theory is the Curry-Howard isomorphism, which says there exists an isomorphism between constructive proofs in mathematical logic and programs in dependent type theory. In other words, if a program can produce a term of a certain type, that program is, in some sense, a proof that the logical proposition corresponding to that type is sound.

Using this insight, a formalization of Arrow's Theorem can be constructed without automation by using the Curry-Howard isomorphism to translate the theorem from a question of mathematical logic to a question of program verification. Using this paradigm means that a proof can be constructed from the many tools theoretical computer scientists have constructed to reason about functions.

This allows proof obligations to be decomposed into individual functions, and these proofs and terms can be passed between functions to verify theorems without the drawbacks of heavy proof automation.

CHAPTER 3
REFINEMENT TYPES

While decomposing proofs into helper functions that pass various proof objects between them has obvious benefits for code readability and avoiding code duplication, it raises an obvious question: how does a function return a proof?

The answer is simple. In a dependently typed language like Agda, a proof term is no different that an array or a list. Thus, functions can return a dependent product that contains a term and then a proof about the content of that term. This type signature is known as a 'refinement type' because it uses a predicate to 'refine' the outer term to something more specific.

Refinement types have many uses outside of proof assistants. One of the best-known projects to use refinement types is Liquid Haskell, which bundles the programming language Haskell with a system of logical predicates and uses an automatic proof solver (e.g. Z3) to ensure that functions satisfy the logical predicates provided by the programmer when they write the function's type signature.

For example, Pena [9] uses sorted lists as a case study to show Liquid Haskell can ensure that invariants are maintained within a data structure:

```
{-@ data IncList a = Emp
                   | (:<) hd::a, tl::IncList v:a | hd <= v @-}
insert :: (Ord a) => a-> IncList a-> IncList a
insert y Emp = y :< Emp
insert y (x :< xs) | y <= x = y :< x :< xs
                   | otherwise = x :< insert y xs
```

In the sorted lists example, Liquid Haskell queries an SMT solver to ensure that the 'IncList' predicate is maintained even after a new element is inserted. If someone wrote an insertion function that did not maintain this predicate, the code would not compile. For example, the following code does not compile in Liquid Haskell (though it would in regular Haskell).

```
insert* :: (Ord a) => a-> IncList a-> IncList a
insert* y xs = y :< xs
```

Because '`insert*`' naively inserts the new element at the head of the list, the SMT solver cannot prove to Liquid Haskell that the resulting list is sorted, so Liquid Haskell cannot prove the output is an instance of type IncList. Using Liquid Haskell thus acts as a check on writing bad code since code that does not provably satisfy the relevant predicates will not compile.

Most work with refinement types is motivated by a desire for better production code: code written in Liquid Haskell promises to be bug free because an SMT solver has proven that a given function meets its obligations. This is an ideal use-case for proof automation, since the steps of the proof are basically irrelevant to a Liquid Haskell programmer: all that matters is that the code is actually correct, and the programmer does not care why. In a sense, the Liquid Haskell programmer has outsourced their normal unit testing to an SMT solver that provides stronger assurances about code accuracy than unit testing ever could.

While the vast majority of refinement type-based systems use SMT solvers and other automation techniques, there is also substantial work being done to verify code manually [6] using the refinement type framework. SMT solvers are very powerful, but they have a variety of limitations including the potential for bugs [11] and the fact that they cannot decide statements that involve quantifiers. This leaves room for explicit refinement types as a useful tool in both theorem proving and program verification.

Proof assistants like Lean and Agda are already dependently typed, so incorporating refinement types into code can be done easily using the language's core features. The Σ type in Agda, which represents a dependent product, makes it easy to create type signatures with proofs embedded in them. For example, the following code uses a Σ type to prove there exists a natural number greater than one:

```
one : ℕ
one = suc zero

existsNat>one : Σ ℕ λ n → n > one
existsNat>one = (suc (suc zero)) , (s≤s (s≤s z≤n))
```

Agda is unique in that it is one of the only dependently typed proof assistants with no proof automation features, making it an ideal tool to demonstrate the benefits of using refinement types for theorem proving.

CHAPTER 4
PROVING ARROW'S THEOREM WITH REFINEMENT TYPES

Discussing the usage of refinement types in a proof of Arrow's Theorem requires a basic understanding of the theorem itself and the concepts involved in its proof.

Arrow's theorem says it is impossible for any system of aggregating votes with at least three candidates to satisfy the following constraints:

- Pareto Efficiency: If all voters rank Alice over Ben, Alice will be above Ben in the aggregated rankings.
- Transitivity: If the aggregated rankings place Alice over Ben and Ben over Chris, they will also rank Alice over Chris.
- Non-dictatorship: there should be no single voter whose preferences determine the outcome of the election.
- Independence of Irrelevant Alternatives (IIA): If voters change their rankings of Chris relative to Alice and Ben but keep the relative rankings of Alice and Ben the same, the relative aggregated rankings of Alice and Ben will not change.

For example, take the following set of voters v:

a	a	a	b	b
b	b	b	a	c
c	c	c	c	a

If we assume that an election consisting of these preferences results in a final candidate ranking such that a is preferred to b, by IIA, the following set of voters v' will also prefer a to b. This is because every voter in v' has the same relative rankings of a and b (i.e. they prefer a to b in v if and only if they also do in v').

c	c	c	c	b
a	a	a	b	a
b	b	b	a	c

IIA seeks to formalize the notion that electoral systems should not be susceptible to the spoiler effect, which occurs when a third-party candidate 'steals' vote share from a major candidate. As Arrow's Theorem proves, even systems that minimize the impact of a spoiler effect cannot eliminate it entirely. For example, the 2022 Alaska at-large congressional district election saw moderate Republican Nick Begich eliminated in the first round of voting even though a majority of voters would have preferred him to both Democrat Mary Peltola and conservative Republican Sarah Palin [2]. This happened in spite of the fact that Alaska had recently switched to instant runoff voting, whose advocates promise that it will limit the spoiler effect [2].

Arrow's Theorem holds when a voter's preference can be modeled as a total pre-order. This is true of any electoral system where voters rank their candidates on their ballots. Mathematically, this means that a voter's preferences must be transitive and strongly connected.

There are a few definition necessary to understand the outline of a proof of Arrow's Theorem. First, a coalition is a subset of a list of voters. A coalition is decisive if, for every pair of candidates Alice and Bob, a proof that every member of the coalition prefers Alice to Bob implies the aggregate electorate prefers Alice to Bob. A coalition is a dictator if it is decisive and only contains one member.

There exist two main strategies for proving Arrow's Theorem known as proofs by pivotal voter and proofs by decisive coalition, respectively. The attached repository[1] contains the first published formalization of a proof of Arrow's Theorem using the decisive coalition strategy. Both proof strategies show that IIA, Pareto efficiency and Transitivity imply the

1. https://github.com/bmwaldman0918/ArrowsTheorem/

existence of a dictator. The proof by decisive coalition requires the following two lemmas:

1. Expansion of Decisiveness: If there exists an electorate such that if everyone in coalition C prefers Alice to Bob but everyone outside of C prefers Bob to Alice and the aggregated electorate prefers Alice to Bob, the members of C are decisive for every pair of candidates.

2. Contraction of Decisive Sets: Every decisive coalition with more than two members contains a decisive proper subset.

From these lemmas, Arrow's Theorem is simple. By the Pareto efficiency assumption, the coalition containing the entire electorate is decisive. By the contraction of decisive sets, any decisive coalition can be made smaller until it has size 1. A decisive coalition of size 1 is a dictator. For a more in-depth explanation of the details of the proof, see Maskin et al. 2014 [7].

To prove each of these lemmas requires manipulating voters in very specific ways such that they maintain certain preferences they had prior to manipulation but are also provably altered in specific ways required to apply either Pareto efficiency or IIA.

For example, the function `LemmaTwo` in `Arrow.Agda` involves taking a list of voters that collectively prefer Alice to Bob and manipulating their preferences to prove that they must also prefer Alice to Chris. This involves manipulating voters such that their preferences on Alice and Bob are identical, but Chris is above Bob.

This notion is easily formalized by the refinement type framework, since a function can take a voter and return a voter along with function that prove certain properties about the returned voter. Take the following type signature[2]:

```
Alter-Last : {_R_ : Fin n → Fin n → Set}
           → (p : Preference n _R_)
```

2. The full function is here: https://github.com/bmwaldman0918/ArrowsTheorem/blob/main/Util/Voter.agda

```
→ (z : Fin n)
→ Σ (Fin n → Fin n → Set) λ _R'_
  → Σ (Preference n _R'_)
    λ p' → (∀ a   → ¬ a ≡ z → P p' a z)
         × (∀ x y → ¬ x ≡ z → ¬ y ≡ z
                  → (x R y → x R' y)
                  × (x R' y → x R y))
```

While dense to read, the function is relatively simple intuitively. Alter-Last takes a voter p and a candidate z and returns a new voter p' who prefers every candidate over z and has the same preferences as p for every other pairing of candidates that does not include z. Analogous functions can be constructed to move a candidate to the top of a voter's rankings.

By making heavy use of functions like Alter-First and Alter-Last, the proof contained in the main file `Arrow.Agda` is relatively clean and readable, containing few if any fine-grain manipulation of voters since these manipulations can be passed off to auxiliary functions and the relevant information can be returned as part of the type signature while also avoiding automation.

Furthermore, the modular nature of these functions makes it easy to use them in future work. Other results in social choice theory (e.g. the Duggan-Schwartz theorem [3], Sen's paradox [10]) make heavy use of similar manipulation of voters. A proof of the Duggan-Schwartz theorem could use the same functions as this proof of Arrow's Theorem, making it much easier to verify.

CHAPTER 5
FUTURE WORK

Refinement types are a natural way to reason about and formalize results from the social sciences. They have a variety of benefits, including improved proof readability and increased ease of compartmentalizing code.

Furthermore, they are an especially natural fit for theorems from social choice theory, since they make it easy to alter preference relations in clear ways. There exist several natural directions from future work in this area. First, it would be relatively simple to use this framework to verify other theorems. Furthermore, since this proof of Arrow's theorem is constructively valid, it would be interesting to translate it into a programming language like Liquid Haskell that is more practical for executing code and constructively produce dictators for a given election.

Lastly, translating the proof into Liquid Haskell would also provide interesting opportunities to learn more about the bounds of various automated proof solvers. It is not obvious how good these tools would be about reasoning about voters and preferences in the Arrovian framework since they were not built for that purpose. Translating Arrow's theorem into Liquid Haskell would provide interesting insight into the generalizability of automated proof solvers.

BIBLIOGRAPHY

[1] Kenneth J. Arrow. “A Difficulty in the Concept of Social Welfare”. In: *Journal of Political Economy* 58.4 (1950), pp. 328–346. ISSN: 00223808, 1537534X. URL: http://www.jstor.org/stable/1828886 (visited on 04/29/2025).

[2] Jeanne N. Clelland. *Ranked Choice Voting And Condorcet Failure in the Alaska 2022 Special Election: How Might Other Voting Systems Compare?* 2024. arXiv: 2303.00108 [cs.CY]. URL: https://arxiv.org/abs/2303.00108.

[3] John Duggan and Thomas Schwartz. “Strategic manipulability without resoluteness or shared beliefs: Gibbard-Satterthwaite generalized”. In: *Social Choice and Welfare* 17.1 (2000), pp. 85–93. ISSN: 01761714, 1432217X. URL: http://www.jstor.org/stable/41106341 (visited on 04/29/2025).

[4] Peter Gammie. “Some classical results in Social Choice Theory”. In: *Archive of Formal Proofs* (Nov. 2008). https://isa-afp.org/entries/SenSocialChoice.html, Formal proof development. ISSN: 2150-914x.

[5] John Geanakoplos. “Three brief proofs of Arrow’s impossibility theorem”. In: *Economic Theory* 26.1 (2005), pp. 211–215.

[6] Jad Elkhaleq Ghalayini and Neel Krishnaswami. “Explicit Refinement Types”. In: *Proc. ACM Program. Lang.* 7.ICFP (Aug. 2023). DOI: 10.1145/3607837. URL: https://doi.org/10.1145/3607837.

[7] ERIC MASKIN et al. *The Arrow Impossibility Theorem.* Columbia University Press, 2014. URL: http://www.jstor.org/stable/10.7312/mask15328 (visited on 04/28/2025).

[8] NobelPrize.org. Oct. 1972. URL: https://www.nobelprize.org/prizes/economic-sciences/1972/press-release/.

[9] Ricardo Peña. “An Introduction to Liquid Haskell”. In: *Electronic Proceedings in Theoretical Computer Science* 237 (Jan. 2017), pp. 68–80. ISSN: 2075-2180. DOI: 10.4204/eptcs.237.5. URL: http://dx.doi.org/10.4204/EPTCS.237.5.

[10] Amartya Sen. “The Impossibility of a Paretian Liberal”. In: *Journal of Political Economy* 78.1 (1970), pp. 152–157. ISSN: 00223808, 1537534X. URL: http://www.jstor.org/stable/1829633 (visited on 04/29/2025).

[11] Dominik Winterer, Chengyu Zhang, and Zhendong Su. “On the unusual effectiveness of type-aware operator mutations for testing SMT solvers”. In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428261. URL: https://doi.org/10.1145/3428261.