Title: Verifying Arrow’s Impossiblity Theorem using Behavioral Subtyping

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.

# Introduction

Arrow’s impossibility theorem formalizes an unfortunate truth about collective decision making: all voting systems are provably flawed. In his [insert year] paper [insert title], Kenneth Arrow proves 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.”

However, like most important results from the social sciences, 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 of the proofs of Arrow’s theorem I read for this project were disappointingly unclear, including some of the most referenced in the literature. It’s worth emphasizing again how prominent Arrow’s theorem is: it is considered one of the most important theoretical results in all of economics. If even proofs of Arrow’s theorem suffer from vagueness, I am sure that many lesser-known results could be clarified substantially by formal verification. In addition, I suspect that there exist at least a few (though possibly many more) results whose proofs are just incorrect. Theoretical economics is thus both a technically feasible and academically rewarding site for work in formal verification.

Unfortunately, the limited work done so far in verifying social scientific theorems has failed to live up to this promise. Many human proofs of Arrow’s theorem 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. Furthermore, the limited verifications of Arrow’s theorem 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. In sum, the vast majority of computer verified proofs of social scientific results are painful to produce and confusing to read.

One cause of these problems is the paradigm from which theoretical computer scientists tend to approach the social sciences. Verifying social scientific results becomes substantially easier if we formalize their various components as programs and apply well-known techniques for reasoning about programs (e.g. Hoare logic). In the remainder of this paper, I will develop a paradigm for formalizing social scientific results in proof assistants using refinement types, a concept in type theory that allows programs to return both an output and a proof about the content of that output. I will then discuss my proof of Arrow’s theorem using refinement types, and finally I will discuss the differences between my computer verified proof and a selection of hand-written proofs of Arrow’s theorem.

# Refinement Types

Normal type signatures do not make guarantees about the content of the data returned, which can result in functions that fail to execute expected. For example, imagine there exists a function called “double” from N -> N. From its name, we might expect this function takes an input and multiplies it by two. However, there is nothing that forces one to implement the double function in this way. There are an infinite number of functions from N -> N, and the body of our double function could contain any of them. For example, there is nothing that stops our double function from statically returning 0 no matter.

[insert python examples here]

While we would hope that no human programmer would be silly enough to name a function that returns 0 every time “double,” there are many instances in which human written programs contain less obvious bugs such that their implementation violates the spirit of the function.

[0 indexed arrays]

One solution to this problem is incorporating refinement types into a programming language’s type system. Refinement types are a special kind of dependent type that allows a function to specify certain predicates about the data it returns. For example, we can rewrite our double function using refinement types to guarantee the number it returns is equal to the input times 2. The code written with refinement types will not compile if the type checker has not been convinced that the predicate will always be satisfied.

[insert agda example with double]

This allows programmers to be ensure that the spirit of their code is correct: if it is not, the code will not compile. One of the best-known projects using refinement types is Liquid Haskell. In Liquid Haskell, an automated proof solver verifies that functions are correct with respect to their type signature, including any predicates specified in a refinement type.

[contrast liquid vs normal haskell]

Unfortunately, however, there are a number of obstacles to writing production code in a project like Liquid Haskell, such as the difficulty of implementing it with already written code and the unfortunate reality that many common programming languages do not have strong enough native type systems to incorporate refinement types (e.g. Python). Furthermore, while not an issue intrinsic to refinement types, any use of refinement types in conjunction with an automated proof solver relies on the effectiveness of the solver to solve the proof. It may be possible to write correct code that the solver cannot prove is correct. In this case, the restrictions of the type system could prevent good code from compiling.

Despite the difficulties in incorporating them into production code, refinement types are an excellent resource to reason about programs in a proof assistant. Proof assistants usually have sufficiently expressive type systems that incorporating refinement types into code is as easy as deciding to do so. Furthermore, unlike other mechanisms like Hoare logic, a proof that the refinement type is satisfied can be easily included in the body of a function with little extra work.

Refinement types are a natural fit in the context of Arrow’s theorem and other results from social choice theory, since these proofs often require feeding an object through a function and then proving the output is “similar” to the input in some way. Without refinement types, it is difficult to convince the Agda type checker that the input and the output are connected in the relevant way necessary for the proof, but with refinement types it is easy. Furthermore, refinement types substantially improve the readability of proofs since they allow proofs to be divided between arbitrarily many functions as long as the relevant refinement types are passed between them. This approach is thus very promising as a mechanism to simplify both the reading and writing of computer verified proofs in social choice theory.

# Arrow’s theorem

## Why Arrow’s?

Arrow’s theorem is a natural first theorem to formalize in the refinement type framework. Its prominence means there are many proofs of it, so we can choose which proof is most amenable to formal verification. Furthermore, many other theorems in social choice theory are closely related to Arrow’s theorem (e.g. Gibbard’s theorem), so a verified proof of Arrow’s theorem would be useful for others seeking to formalize other theorems in the social sciences.

## The Arrovian Framework

Take an election with at least 3 candidates (named Alice, Ben and Chris for convenience). Arrow’s theorem says it is impossible for any system of aggregating votes to satisfy the following constraints:

1. Pareto Efficiency: If all voters rank Alice over Ben, Alice will be above Ben in the aggregated rankings.
2. Transitivity: If the aggregated rankings place Alice over Ben and Ben over Chris, they will also rank Alice over Chris.
3. 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 (insert figure).

This 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. In some instances, a spoiler candidate can alter the result. In the Florida presidential election in 2000, if Green candidate Ralph Nader’s voters had voted for Democrat Al Gore, Gore would’ve won the presidency.

1. Non-dictatorship: there should be no single voter whose preferences determine the outcome of the election.

By convention, my proof proves the existence of the first three conditions imply the existence of a dictator, but it is possible to prove that any combination of 3 conditions can produce a violation of the fourth. It is also worth noting that virtually all real electoral systems violate IIA and satisfy the other three conditions.

Arrow’s theorem holds when voters can be represented as preference relations. In other words, a voter’s preferences between candidates satisfies the following properties for all candidates Alice, Ben and Chris:

1. Transitivity: If Alice >= Ben and Ben >= Chris, Alice >= Chris
2. Completeness: Either Alice >= Ben or Ben >= Alice
3. Decidability: Either Alice >= Ben or we can prove not Alice >= Ben

Any system that involves voters selecting a single candidate or ranking a list of candidates can be modelled this way. Note that this framework allows for candidates to be tied in a voter’s rankings. In other words, Arrow’s theorem successfully proves that most common voting systems are subject to a spoiler effect, including ranked choice and plurality systems.

# Outline

1. What is arrows?
   1. Every electoral system is provably terrible
   2. Intuition for why conditions are interesting/useful
      1. IIA formalizes no spoilers
      2. Transitivity is just … how we rank things, nec to be useful
      3. Pareto
   3. Every real world system just has spoilers basically
2. Why arrows
   1. Natural choice for behavioral subtyping
      1. Also a natural choice for constructive logic since in some sense we “construct” a dictator
   2. Not a lot of work in formal verification of social sciences
   3. Rich basis for expansions
3. What’s interesting about this strategy
   1. All previous proofs are tactic mode proofs
      1. Which makes sense! Those people all were math people
      2. And they were probably way easier
      3. I couldn’t get tactic mode to work on my computer ☹
   2. This approach is basically wholly on the program side of the curry-howard isomorphism
4. How does the proof work?
   1. Combo of border notes and sen strategy
5. Lessons learned
   1. Lots of proofs are bad!
      1. Conflations between product types and functions
      2. Products carry state and functions do not
6. Future work
   1. Gibbard satterwaite
   2. Alt strategies to finish the proof
   3. Do it in a real programming language and execute the code (don’t just compile it)
   4. Liquid Haskell/do it in an smt solver