Rochester Institute of Technology: Computer Science

Independent Study Final Report: Formal Proof Construction with Coq

Jodie Miu
Professor Matthew Fluet

December 14, 2018

1 Introduction

As stated in my proposal, the aim of this independent study was to learn more about the intersection between mathematics and computer science, and the use of theorem provers in mathematics. I learned that theorem provers are not the panacea for all problems because the restraints they place are sometimes too rigid and confining to produce new results in.

2 Reflection Papers

2.0.1 Proof Style

Jodie Miu
3 September 2018

The paper *Proof Style* is a simply laid out survey of the different ways to communicate a mathematical proof to a computer theorem prover. The author John Harrison begins by defining what ‘proof’ means in this context and explaining how he is classifying proof styles. There are five criteria: the level of automation, the degree of controllability, whether or not the emphasis is on a declarative or procedural style, the direction the proof is written, and whether or not the proof is processed interactively or submitted via batch process. Harrison then follows with a description of various existing systems and how they exemplify variations of the aforementioned criteria. Full programmability is discussed, as users may desire to write custom proof procedures. This brings the problem of possibly allowing the user to produce false theorems. After this, Harrison compares various aspects of proofs generated from these proof paradigms, such as efficiency of processing, maintainability, and so on. He concludes that a declarative style is better for "pure, abstract proofs", where a procedural style would be better for "big, ugly, concrete proofs".

I completely agree with the author's assertion that a full, Turing-complete programming language to write custom proof procedures in also brings the 'Java problem' with it, meaning that full programming power can also mean the ability to do something dangerous. I have unwittingly accessed memory I was not permitted to or corrupted my own data when programming in C or Java. Compare this to when I was working at Red Hat, on OpenStack, writing Heat Orchestration Templates. These are YAML-based and domain-specific, and I was never able to do anything dangerous. The most dangerous thing I could possibly do was to add in hooks to Bash scripts.

Additionally, I found that the author's statement "there are other properties that certain people may consider important" under the Other factors section to be weak. Other factors that are not mentioned by the author are things like quality of community and the learning curve involved. While these factors are harder to quantify, they are still important to consider, being that the life-cycle of these proof systems is just like any other software project. However, note that Harrison would have found it difficult to incorporate any discussion about this in his paper because he tried to stick to a more general style, comparing declarative versus procedural, introducing concrete examples sporadically.

I found it difficult to understand what the LCF approach was, because it is very different from how an abstract stack is used and implemented. Harrison never explained how an abstract data type of theorems would solve the 'Java problem'. I also did not understand how an abstract data type of theorems could be defined, being that theorems come from very different contexts and state very different things. It seems very possible to me that a false theorem might be produced in a definition of an abstract data type of theorems general enough to apply to all theorems.

Finally, I found the author's discussion of declarative versus procedural proof styles to be very interesting. Coming from a mostly procedural programming background, it seemed counterintuitive to me that a declarative proof style would be easier to read, write, and maintain. I am used to having an easier time of debugging C or Java code, as opposed to when I was working with Prolog. I enjoyed how the author explained the bias in his assessments, being that different problems (i.e., pure mathematics style proofs or verification proofs with large terms) lend themselves to different proof styles. As a result, I think I need to change my opinion because pure mathematical proofs are what I am focusing on this semester.

2.0.2 The Science of Brute Force

Jodie Miu
10 September 2018

The article *The Science of Brute Force* is written with a fairly technical audience in mind about a new usage of brute force reasoning. The authors Marijn J.H. Heule and Oliver Kullmann introduce the previous state of search problems. Heule and Kullmann explain that previously, brute force was only considered suitable for the simplest of problems. Progress in the last two decades on Satisfiability solving (SAT) has changed this view. SAT solving is the process of determining whether or not a propositional formula has a solution. Heule and Kullmann proceed to make the argument that not all results will be understandable by humans, and that these results are still meaningful. They discuss problems such as Pythagorean Triples, while describing how SAT solving algorithms and heuristics work with broad strokes. They introduce the idea of alien truths as a way to classify truth statements whose proofs may or may not be "humanly understandable".
I was extremely taken aback at the authors' statement that problems solved via brute force were still very important in mathematics. While I did not believe that long, mechanized proofs were completely useless, I will admit I was suspicious of what possible contributions brute force problem solving had to our understanding of mathematics and problem solving. I was introduced to the beauty of mathematics by strong pure mathematics proponents who were mainly concerned with elegant proofs and the understanding of every step of solving a problem. They were quite worried that computer-aided proofs were hard to understand, and would hamstring true mastery of an area. Thinking back on these statements, I don't think they are contradictory. Certain problems lend themselves better to human mathematical proofs, and other problems do not seem to be solvable with only human effort and ingenuity. These other problems may just be well-suited to brute force. As to the fear of a lack of true mastery or understanding, I don't think it is relevant necessarily. The problems that are classified strongly alien or extra alien may be problems that would never have been comprehensible with only human effort.
Like the authors stated, it may also be that exploring these problems and their long proofs will lead to a better understanding of questions like "why there are no short proof" and "what makes a problem hard". The thing that convinced me most was the assertion that we must study the limits of our current knowledge, even if brute force proofs were long and therefore uninteresting and not relevant. Iterating on what we already know can only get us so far.
There is still value in pursuing SAT solving further; even without application to widening the boundaries of mathematics, we can still use it for verification of mathematical results and to show correctness of highly complex systems.
I found it difficult to understand the Boolean Schur Triples example, because it is very different from the kinds of satisfiability problems I encountered in CS Theory. First of all, the definition of the problem was not already predefined with booleans. Although the authors did not show how the problem was translated to SAT encoding, I am extremely intrigued by the process of picking a problem, encoding it as a SAT problem, and solving

it. In conclusion, I found great value in this paper because it gave context to satisfiability problems, helping me understand they could be used for.

2.0.3 Depth-First Search and Strong Connectivity in Coq

Jodie Miu
23 September 2018

The *Depth-First Search and Strong Connectivity in Coq* paper is about the mechanization of Wegener's proof of Kosaraju's algorithm for finding the strongly connected components of a directed graph. The author François Pottier begins by laying out the key concepts behind Kosaraju's algorithm. Pottier then describes some definitions. The author lists out properties of DFS forests, proves Kosaraju's algorithm, and briefly describes the process of writing an executable version of the proof.
I completely agree with the author's assertion that "one must distinguish between mathematical objects and their runtime representations", because I encountered the same thing when writing code for other courses. The author stated that this applies when writing executable code in Coq, but really, generalizing this statement to be "keep in mind the difference between abstract models and their runtime representations" still holds. I have been neck-deep in fixing a bug, realizing along the way that the bug came from a difference in the programmer's mental model and the code. That being said, I feel distinguishing between the two is more paramount in Coq than in any other programming language I have written. For one, Coq proof scripts are hard to read after the fact. If the proof script is unstructured or contains implicit assumptions, debugging the script becomes a monstrously difficult task. One must step through the proof script to see the context, whereas one does not need to run "normal" well-written code to understand what is happening.
I was surprised the paper was so heavily focused on the informal proof of Kosaraju's algorithm. I did expect the paper to be about the process of writing Coq proof scripts for Kosaraju's algorithm. In hindsight, this is understandable. Proofs in Coq cannot be divorced from their informal versions without loss of structure and clarity. I learned this lesson earlier on the hard way. Additionally, Pottier probably wrote this paper expecting that his readers were familiar with Coq. I noticed even the informal proofs showed the author's background in Coq in the inductive definitions, the methodical deconstruction of definitions, relations, and predicates, and the final, deceptively short lemma construction. This was my first time seeing Coq style definitions and proofs outside of the (admittedly) artificial context of Software Foundations. I enjoyed reading about how the author defined forests inductively simply because it was a real-world example.
I found it difficult to understand the process of proof mechanization, because it is very different from the simple proofs in Software Foundations. There are multiple moving parts, using dependent types and tactics I have never seen before. Comparing this to the proofs I have seen in SF, there is a world of difference. Pottier does not, however, completely omit a discussion of the process, including an aside about how the choice of

a recursive formulation led to some pain points. The question I have here is "how might the author have chosen the better formulation from the start?". Pottier did state that it occurred to him how the recursive formulation is likely to cause stack overflows with large graphs after extracting the code to OCaml, so he could have ostensibly considered how the extraction to OCaml would have changed the dynamics of the executing code.

2.0.4 Essential Incompleteness of Arithmetic Verified by Coq

Jodie Miu
2 October 2018

The paper *Essential Incompleteness of Arithmetic Verified by Coq* by Russell O'Connor outlines the process O'Connor went through in mechanizing a proof of the Gödel-Rosser incompleteness theorem in Coq. Before starting on the proof itself, the author begins by describing how he developed the theory of first-order classical logic within Coq. A dependent record of types with an arity function mapping from symbols to natural numbers is defined as being an arbitrary language. After all the necessary theorems have been defined, O'Connor creates two languages: the Language of Number Theory (LNT) and the Language of Natural Numbers (LNN). An encoding scheme for turning formulas and proofs into natural numbers is described, and the functions operating on these codes are proved to be primitive recursive. The incompleteness theorem was proved by showing that all primitive recursive functions are representable in a weak axiom system.
The lesson I learned overall from the paper was that mechanizing a proof required forethought into Coq's restrictions and method of function, attention to detail that may not even be explicit, and deep understanding of the proof to be mechanized. I noticed that because Coq's positivity requirement led to O'Connor's first attempt at defining a Term to fail. Upon first reading, the author's first definition does not seem to have anything wrong with it. Reading on further reveals that "expanding the definition of length reveals a hidden occurrence of Term0 ... passed as an implicit argument ...". Lesson learned: beware of function definitions and implicit arguments, because they might be hiding important information. Over and over, I also noted how frequently the author described multiple techniques for accomplishing the same thing and how he decides upon one for one reason or other. I would personally be worried about making a decision that makes the following steps more difficult, and not detecting it until after I had made many other implementation decisions. I wonder if this might make it harder to detect where the most problematic implementation was. This worry is compounded by the paragraph before the section on Primitive Recursive Functions, stating that O'Connor's usage of a weaker hypothesis meant that "the theorem could be used to prove that any complete and consistent theory of arithmetic cannot define its own axioms". What I took away from that paragraph was that definitions and proof approaches informed what the theorem could be used to prove. This exists in informal proofs as well, but not to the same extent. For example, in Abstract Algebra 2, we were discussing a quick sketch of a proof of how many Sylow-2 subgroups were in S5. My professor defined a set H as being all permutations

that fix 5 only, initially, but after a few questions from my classmates, he removed the word only. As he went on to the rest of the proof, it became clear that the word “only” prevented the set from being a subgroup of S5. This stood out to me as an example of how definitions mattered in informal proofs as well.
Unlike with informal proofs, the “entity” doing the proving (eliding over how Coq and Boyer-Moore operate) influences what can be proved and how it might be proved. On the topic of conversion from informal proofs to formal, the author stated that he found replacing ellipses with recursive functions was one of the hardest tasks, as there was no information on what inductive hypothesis should be used. I look forward to engaging with this problem myself.
I must also say that in reading this paper, I realized how much background knowledge went into this paper. This paper gave me a preview of Coq-generated induction principles, structural recursion, primitive recursion, what logic was like, and how languages can be defined for a proof. Additionally, I found it fascinating how there were two different types of logic simultaneously. All in all, I enjoyed this paper tremendously and found it very educational as to how much effort and education was involved in an undertaking like this.

2.0.5 Efficient Certified Resolution Proof Checking

Jodie Miu
16 October 2018

This paper is about the proof tracing format that the authors designed, eliminating complex processing. The goal behind this proof tracing format is to guarantee the computed results are correct, while being efficient enough to use on massive proofs. Prior to this paper, researchers in this field checked the results produced by SAT solvers, outputted an execution trace of the SAT solver, or worked on developing a certified proof checker. The problems with each of these approaches was generally one of the following: no guarantee of correctness, not scalable, or not suitable for unsatisfiable instances where there might not exist any succinct witness. After describing the proof format, the proof verification algorithm is implemented in Coq. A certified checker is extracted from this implementation, whose runtime execution is compared to checkers for other proof formats.
Following up from the article *The Science of Brute Force*, I found this paper easier to understand. The ACM article sets the stage, explaining how satisfiability solving can be used in a variety of applications. One of the formats mentioned by the authors, DRAT, is first described here. Although I am not familiar with the other proof formats, it was interesting to see the differences in opinion about DRAT. From the eyes of the authors of The Science of Brute Force, DRAT “seems to be a good proof format for existing and future SAT solvers”. For the most part, the authors of *Efficient Certified Resolution Proof Checking* seem to agree, stating that DRAT supports an extended range of options.
But would the authors of The Science of Brute Force think that GRIT is a good proof format? In the ACM article, five properties are described that the ideal proof format

should have: (1) proof production should be easy, (2) proofs should be compact, (3) proof validation should be simple, (4) proof validation should be efficient, and (5) all techniques should be expressible. By the arguments Filipe et al. are making, GRIT certainly seems to have property (3). Without the complex processing that previous proof trace formats used to require, unit-propagation becomes much simpler. GRIT also seems to be efficient (property (4)), as the uncertified C checker verified all 280 GRIT files in 14 minutes. The certified OCaml checker was much slower, but still within reason. Filipe et al. currently do not output the GRIT format directly, instead choosing to use a modified version of a drat-trim preprocessor. It becomes hard to gauge whether or not proof production is easy in this instance, since the goal of easy proof production is so that many solvers can support it. A modified version of a drat-trim preprocessor ties the GRIT format to the development of drat-trim, but I assume that this will be rectified eventually. Additionally, I find it rather difficult to evaluate whether or not all techniques are expressible, not knowing what the set of all techniques consist of. From here, it seems like the authors of The Science of Brute Force would tentatively approve.

2.0.6 A String of Pearls: Proofs of Fermat's Little Theorem

Jodie Miu
29 October 2018

The paper *A String of Pearls...* is a brilliant summary of a few of the different ways the authors implemented various proofs of Fermat's Little Theorem in HOL4. The authors Hing-Lun Chan and Michael Norrish start with describing Golomb's necklace proof and the classic number-theoretic proof. In the next section, elementary aspects of group theory are described and mechanized, followed with descriptions of how group theory can be applied to the first two proofs.

First of all, I found the organization of this paper slightly confusing. Because there are multiple proofs being described, I questioned whether or not the first two "proof" in the first section were really the authors' way of preparing the reader before introducing group theory, continuing with the two proofs. I now believe that Chan and Norrish intended for the paper to describe four proofs. My confusion arose from the natural similarity in the first proof to group theoretic concepts; simply put, I read that first proof, completing it with orbits and group actions.

After reading the paper, I went to the proof script repository. Surprisingly, the notation for HOL-4 is rather incomprehensible to uninformed eyes. The authors' description of HOL4 as being rather pleasant to the eyes is subjective.

Once again, I am disappointed with the lack of exposition on the experience of mechanizing the proofs. Perhaps, I am not the target audience and the authors wrote this paper with submission to a conference in mind, where they could expect that the people reading the paper were fully familiar with the difficulties of mechanizing proofs. After the midterm project, I developed a small understanding of this. Admittedly, the proof I mechanized is not a powerful or significant result by any means. I was hopeful that Chan

and Norrish would describe hardships unique to HOL4, or why HOL4 was chosen. I also wonder if the cross-disciplinary nature of the area makes defining the scope awkward.
What I did enjoy, however, was the final verdict of combinatorial proofs being more difficult to mechanize in HOL4 than group theoretic proofs. I wished that the authors had proposed some ideas as to what exactly about HOL4 may have made combinatorial proofs harder to mechanize. Instead, they deferred, preferring to state "HOL4's features make some proofs easier to automate, and some goals easier to express". A few features from other theorem provers are mentioned, such as sub-types in Coq and axiomatic type-classes in Isabelle.
In addition, I am doubtful of the use of the number of lines of code in a linearized version of each proof script as being a good indicator of effort or "goodness". This is the same kind of problem programmers face when trying to gauge quality.
In conclusion, this paper was extremely pleasant to read, but a different one would have been better for this independent study. The proofs were elegant and easy to understand, but at times, a lot of space was spent on describing basic group theory. All the same, this report focused more on the mathematics, and not enough on the mechanizing of it.

2.0.7 Propositions as Types

Jodie Miu
5 November 2018

The *Propositions as Types* article, written by Philip Wadler, is a fascinating overview of the principle by the same name. Wadler begins by describing a brief history of related fields and the theory of computation, continuing to explain that the "Propositions as Types" principle refers to a true isomorphism between proofs and programs. Several examples of applications are outlined, such as its underpinnings in functional programming, its usage in Coq, Scala, and so on. The aforementioned principle is then discussed in various contexts with different kinds of logics, such as Church's simply typed λ calculus and Gentzen's intuitionistic natural deduction.
I found it surprising at first that such a correspondence should exist for many kinds of logics and computation, but found that it made sense after some consideration. A program is but a series of manipulations upon some objects according to a set of rules. Similarly, a logic characterizes how objects might be manipulated in a formal system according to some set of rules.
I loved Wadler's writing style and the inclusion of the story *The Gondoliers* in order to illustrate intuitionistic logic. The inclusion of examples of the trickle-down of these concepts into the design of generic types in Java and C#, along with the type systems of functional languages. With this additional knowledge, I am starting to better understand the underpinnings of what I had previously learned to do with generic types in Java "just because". I also enjoyed Howard's observations and the connection between conjunction and a pair, disjunction and a tagged union, and implications with functions.

However, I wish that the author had included more description of what Descartes's coordinates, Planck's Quantum Theory, and Shannon's Information Theory have to do with Propositions as Types. Another thing I wish the author had included was how we know this isomorphism to be true. In the paper, Wadler states that this isomorphism holds true for a range of logics, but I wonder if it is possible that there is some system of logic in which no correspondence to a program exists. If such a logic exists, I wonder what about it breaks the principle of Propositions as Types, and if we could learn from this anomaly.
The fact that the same ideas are discovered independently by logicians and computer scientists was also something I found compelling. The PLC class that I took did not go into programming language design much; it was mostly focused on learning multiple different programming languages under various paradigms. I wish I had learnt more in that area. In conclusion, I found this article exciting, being that it describes a "grand unifying theory" of sorts. Being that logic is both used by and informed by mathematics, I also wonder what the implication of this isomorphism is for mathematics, like set theory. The author succeeded at piquing my interest; I was left wanting more details by the end.

2.0.8 Improving Real Analysis in Coq: A User-Friendly Approach to Integrals and Derivatives

Jodie Miu
12 November 2018

This paper discusses Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond's attempts to improve the current state of real analysis in Coq. The authors begin by introducing how formal proof assistants and analysis were not studied as much algebra. Boldo et al. continue to explain the current definitions of differentiability and integrability as described in various proof assistants. The next few sections contains a brief overview of the logical foundations of Coq, how real numbers are currently defined in Coq, more definitions, application, and a conclusion.
I enjoyed that the authors described exactly how real analysis was difficult in Coq. The libraries were not updated. Additionally, the fact that derivatives require a proof term to be written means that working with derivatives and integrals in Coq is not practical. There are no definitions and lemmas for partial derivatives, either, in addition to the fact that mathematicians do not prove their theorems with proof terms. Keeping this in mind, I could extrapolate that probability, statistics, and differential equations will also be difficult in Coq. As of right now, it seems to me that applied/computational mathematics in general could be harder to formalize. Algebra, graph theory, and possibly number theory are already formal and generally are centered on discrete objects. This may simply be my bias, not having learned about analysis yet.
I also appreciated that Boldo et al talk about the type hierarchy in Coq. Reading about the type hierarchy from authors other than those of SF gives me insight into how different people understand the concepts surrounded Coq. I did not know that Prop is

non-informative, nor had I heard of specification types or existential types. I wonder if existential types are types that include an existential qualifier. This served as a reminder that I have only just begun to scratch the surface when it comes to Coq; there is still so much I have not learned yet, like dependent types and automation.
In the same section about the logical foundations of Coq, I realized that the problem the authors described in attempting to express that both f and g are functions that are differentiable was a problem I had already occurred upon when working on the mid-term project. For example, I originally defined evenness as a definition, but I struggled later on, when trying to use evenness in a logical formula. In hindsight, it makes sense. I had not fully understood the differences between Prop and Type, and what I was trying to do was not a valid logical formula.
To be honest, I struggled reading this paper. It focused more on real and numerical analysis, which I don't know much about. This paper was a bad choice, since I was out of my depth when reading it and was not able to gain much insight.

2.0.9 A Computer-Checked Proof of the Four Colour Theorem

Jodie Miu
19 November 2018

The *A Computer-Checked Proof of the Four Colour Theorem* paper details how George Gonthier formalized the proof of the Four Colour Theorem. The first section describes the history of the Four Colour Theorem, including the work on it up until just prior to the author's attempt at proof formalization. Section Two is about the theorem and the different ways the problem could be represented. Section Three is a historical overview of the proof, describing previous work in formalizing the proof. Section Four is about instrumental features of Coq, and Section Five is about the formal proof. The rest of the report follows with a chronological description of the development process, and a conclusion.
I found that the author's discussion of boolean functions instead of logical predicates to be very interesting. Although I used to believe logical predicates were better, I think that my blanket belief needs to change. I never really had a good reason for believing that logical predicates were better; this impression came about erroneously when working on the midterm project because I wanted to use a boolean function in a proposition. I had to change my proof script to use propositions only, and I have been wary of these functions since. Gonthier uses the fact that evaluation is transparent to simplify the proof script considerably. Boolean functions also allowed him to use the rewriting of boolean expression to apply equivalences inside a complex statement. What I found the most intriguing was Gonthier's assertion that Boolean functions make it easy to reason by contradiction. While I haven't learned about the Discriminate tactic yet, it may come in useful on the final project, since proof by contradiction is used quite frequently in the proofs my abstract algebra professor presented. The combination of Boolean functions and the Discriminate tactic is one of my next targets to learn about.

However, I do wonder what types of proofs would lend themselves best to this sort of "generate-and-test" style of proof. Perhaps proofs for other combinatorial proofs, finite groups, or discrete math would be a good fit. I don't suppose it would make sense in problems with an unbounded number of test cases, like in number theory. The author did state that having Coq execute a bounded number of simple, deterministic programs is better than directing Coq with tactics in this problem. The Four Colour Theorem does not contain much symbolic algebra, but something more algebraically oriented (like algebra) may not allow for such an approach.
Another thing I noticed was how the decision to develop custom user extensions was borne out of trial and error. The author noticed the development was becoming tedious, leading to large proof contexts. One common thread amongst these papers is that many parts of a Coq project come out of experimentation. I assume these people to be Coq experts, but even the experts must try different things before finding something that works. Nothing is straightforward in Coq, or so it seems.
Similarly, I never realized how powerful computational reflexivity can be. So far, my usages of reflexivity have been trivial: to check things like $x = x$ or $2+2=4$. Compared to the other developments we have seen so far in this class, the concepts behind this computer-checked proof seem incredibly simple. Gonthier lets the system do the work for him whenever possible, using computational reflexivity or transparent evaluation. After reading the chapter on Automation in SF, it was a surprise to read that the author made very little use of the proof automation facilities in Coq. I had expected that a proof with as many cases as this one did would use a lot of automation to discharge cases, like the authors of SF did in their example program.

---

3 Midterm Project

3.1 Midterm Proposal

Jodie Miu
30 September 2018

After much thought, I settled upon proving "Let n be an integer. Then $n^2 + n$ is even". At first, I was considering suggesting proving the Fundamental Theorem of Arithmetic or the Chinese Remainder Theorem, or even the properties of divisibility for integers. Over the course of reading multiple versions of the informal proof, however, I came to realize how much machinery came into play. The idea behind this mid-term project is to (attempt to) isolate one aspect of writing proofs in Coq to the "conversion to a formal proof" skill, avoiding the "defining every relation and property" before even getting to the main proof body itself. I began to reconsider my idea after researching what a few Coq proofs would look like. I thought that I could not pick anything in number theory that came after the beginning of the course, since the proofs in mid- to later parts of the course relied on numerous relations, axioms, definitions, and previously proved theorems.

I found the theorem "Let n be an integer. Then $n^2 + n$ is even" in the Ethan Bloch book titled "Proofs and Fundamentals". This theorem does not need a ton of additional setup, so I can focus on the skills needed in reading an informal proof and translating it into a Coq proof script. There is one thing I do anticipate having to handle, and it's the fact that the theorem is defined for integers, not just natural numbers. This will give me an opportunity to examine how integers are implemented in Coq.

3.2 Midterm Reflection

Jodie Miu
23 October 2018

This project turned out to be more difficult than I had anticipated. In fact, I was not able to finish this project. First, the biggest and most time-consuming obstacle was in picking the right representations and proving any necessary mappings and correctness. There were four possible representations of evenness that I toyed with; these were the inductive ev proposition from IndProp, the recursive evenb function from Basics, the fact that the number can be written as double some other number from Logic, and the translation of the evenb boolean function to a Prop type object. I had previously defined Odd as the negation of the even function, but as I had learned, this makes it difficult to prove things.
There were three things that were crucial to this project: realizing my formal proof did not have to follow the informal proof, deciding on an appropriate representation, and understanding where and how the proof had to broken down.
Previously, I had assumed I had to mirror the informal proof. In this case, the informal proof depended upon rewriting even numbers as $n = 2k$, and odd numbers as $n = 2j + 1$. Personally, I found this harder to work with in Coq, not necessarily knowing what k or j looked like. After Tuesday's discussion, I realized that that was not necessary at all for the odd case. It is also probably not necessary for the even case, but I had already defined my theorems with it.
I also realized I should change my representation of odd to an inductive proposition. Knowing this now, I would say that in the future, I will remember how much information a definition or proposition would contribute to a proof. While defining odd as the negation of even may be true, it does not provide any new information about odd numbers.
In the future, when creating a new Coq proof script, I would also break down the proof on paper from top-down first. I had gotten used to the "guided tours" of Coq and had stopped applying the kinds of problem deconstruction skills I had developed from other math courses. My other problem was that I was deconstructing the proof in the wrong order. Honestly, I had forgotten about the order of operations. Because I would normally square n first, I kept trying to show that the product of two even numbers was even first accordingly. This did not work, because of the way Coq operates. This is really another instance of reasoning backwards as opposed to forward. While it may be more natural to

reason in a forward direction informally, I have often found backward reasoning to work better in Coq.
After I realized that I should be starting with the sum before the product, everything became much easier. The flow of the proof then became: to prove that $n^2 + n$ is even, start by stating that the sum of two even numbers is even. Then Coq pops two new subgoals onto the goal stack. It is now my job to show that n^2 is even and n is even. The flow of the proof for the odd case is very similar, only with the bulk of the work being to show that the sum of two odd numbers is even.
In conclusion, I found this project immensely rewarding. I may not have proved it true for all integers, but it was still really gratifying to see the organic growth of a proof script. I spent a lot of time lost, becoming more confused with each attempt. Then, suddenly, everything clicked and I completed the proof. I learned a lot about how what I had learned from SF were not an immediate jump to writing proof scripts from scratch. I noticed that there were exercises in chapters related to writing informal proof versions of the formal proof. We had previously agreed to skip it, but I can't help but wonder if it would have helped here.

---

4 Final Project

4.1 Final Proposal

Jodie Miu
11 November 2018

For the final project, I am thinking of working through the notebook and homework from Abstract Algebra 1. This is a good idea for several reasons: first, this is more challenging than the previous project, and will require more thought on representation. There are many objects defined in Abstract Algebra, which will allow me to practice defining types. The mid-term project only required a definition of natural numbers (originally integers) and a definition of evenness and oddness. Additionally, group theory in Coq has already been attempted by many people, so I will be able to compare and contrast different representations. This project is open-ended, allowing for growth and variety. This is important because I might find some theorems easier to prove. I might find some more difficult. The goal here is no longer to achieve one theorem, which may require any number of sub-results; the goal is now to learn more about the meta-theory of Coq. I want to learn what aspects of a result make it well-suited for Coq, as opposed to an informal proof. I would like to also learn whether or not Coq is suited to experimentation in mathematics when pursuing new results. When attempting to translate an informal result to a Coq proof script, does the core difficulty change? In last class, we already saw how, in translating informal definitions of groups to formal ones, the work was no longer done via checking for closure, but rather, via the carrier set. I anticipate this project being extremely challenging, but the focus here would be on learning heuristics about writing Coq scripts.

4.2 Final Reflection

Jodie Miu
13 December 2018

Unfortunately, this project became a lesson in how difficult it could be to develop in Coq. There were productive and unproductive obstacles we had to navigate together.
First, any weaknesses in my understanding were quickly uncovered. I had previously avoided partial function application because I was uncomfortable using it. In the past when programming, I had not needed to use partial function application. Additionally, I had been using a variety of programming languages, which either did not support it or did not commonly use it. I had to use partial function application throughout the project and became comfortable very quickly. I was also reminded of how definitions matter when I had to change the way I had defined associativity to match the way Coq.ZArith.BinInt had defined it. My original definition would have complicated my proof script. Unlike other programming languages and their libraries, definitions and lemmas in a library must be transparent. The way a lemma is defined will affect the difficulty of its proof script, and also the way said lemma is used in other projects.
Most of the obstacles were unproductive ones, however. When I asked about defining the group of integers modulo n, I had no idea how difficult it would be. We spent so much time trying to figure out how to define an equivalence relation that Coq's built-in tactics would comply with. There were so many different libraries, all seeming to provide what we needed. I found it rather hard to determine the difference between CEquivalence and Equivalence, for example. The documentation for these libraries also seemed to require a higher level of knowledge than what I had, which intimidated me at first. Simply put, the groups using Coq notion of equality were far easier to define. I had defined the additive group of integers in a matter of minutes, whereas the group of integers modulo n took days.
In the course of trying to define equivalence relations, I also realized that my understanding of Abstract Algebra was not as good as I had thought. When working in a system like Coq that forces clarification of every concept, the flexibility I had grown used to hindered me. I found myself questioning what relationship the equivalence relation on a group had to its operation, identity, inverse, or any of the group axioms. I had internalized these concepts so well I had muddled my understanding. I also realized that in mathematics, we moved back and forth between different representations of groups without even thinking about it. Case in point: we started trying to re-define the representation of the group of integers modulo n as being only its representative equivalence classes, instead of being all integers with multiples n. The representation here is subtly different, but that difference matters in Coq. The former is used more commonly in Abstract Algebra, as it simplifies computation to a bound number of representatives, including the understanding that each representative is interchangeable with any other member in its equivalence class.

The project may not have been challenging in the way we had hoped for, but I still feel that I took away the confidence to attempt other Coq projects on my own. Searching the documentation taught me not to fear the words I hadn't seen before.

5 Accomplished Work

5.1 Time Spent

Here, I document how I planned to spend my time, along with all deviations from the Independent Study plan.

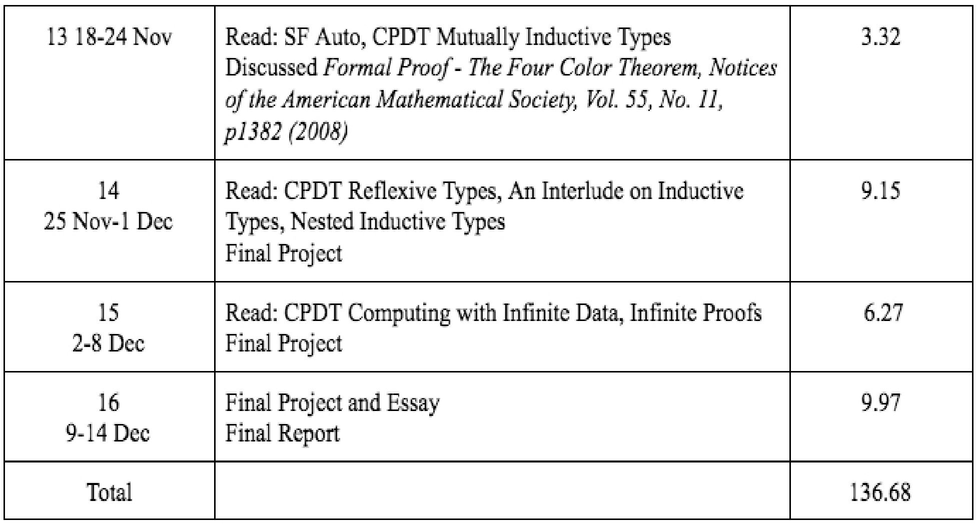
5.1.1 Schedule

Week 1: SF: Basics, CPDT introduction
Proof Style
Week 2: SF Induction, Lists, CPDT Some quick examples
The Science of Brute Force
Week 3: SF Poly, Tactics, Logic
Week 4: SF Total and Partial Maps, Inductively Defined Propositions, The Curry-Howard Correspondence
Depth-First Search and Strong Connectivity in Coq
Week 5: SF Induction Principles, Properties of Relations
Goedel's Incompleteness Theorem (Coq)
Week 6: SF Auto, CPDT Infinite Data and Proofs, Subset Types and Variations
Week 7: CPDT General Recursion, More Dependent Types
Efficient Certified Resolution Proof Checking (Coq)
Week 8: Midterm Project - Fermat's Little Theorem + CPDT Dependent Data Structures
Week 9: CPDT Reasoning About Equality Proofs, Generic Programming
A String of Pearls: Proofs of Fermat's Little Theorem
Week 10: CPDT Universes and Axioms, Proof Search by Logic Programming
Fundamental Theorem of Algebra (Coq)
Week 11: CPDT Proof Search in Ltac, Proof by Reflection
Improving Real Analysis in Coq: A User-Friendly Approach to Integrals and Derivatives
Week 12: Project - Formalize Relational Algebra
Formal Proof – The Four Color Theorem, Notices of the American Mathematical Society, Vol. 55, No. 11, p1382 (2008) (Coq)
Week 13: Project + CPDT Proving in the Large
Week 14: Project
Week 15: Wrap up Project
Week 16: Final submission

The intended work was as follows: I was to follow two textbooks: "Software Foundations" by Benjamin Pierce, et al. (abbreviated as SF), and "Certified Programming with

Dependent Types" by Adam Chlipala (abbreviated as CPDT). Both books are freely distributed online by their authors. These books have built-in exercises, which I worked through with each chapter. I will also be reading application papers, which are somewhat evenly dispersed. An additional, supplementary source is "Mathematical Components" by Assia Mahboubi and Enrico Tassi.
Finally, I was to undertake a mid-term and a final project. The mid-term project was to be fairly simple, intended to test whether or not I could expand the knowledge from completing guided exercises to writing a proof of something nontrivial from start to finish. The final project was far more complex.

Week	Activity	Time Spent (hrs)
1 27-31 Aug	Exercises: SF Basics Read: CPDT Introduction Wrote Paper About: *Proof Style*	6.53
2 2-8 Sept	Exercises: SF Basics, Induction Read: CPDT Some quick Examples Wrote Paper About: *The Science of Brute Force* Discussed *Proof Style*	18.98
3 9-15 Sept	Exercises: SF Lists, Poly, Tactics Discussed *The Science of Brute Force*	14.72
4 16-22 Sept	Exercises: SF Logic, IndProp Wrote Paper About: *Depth-First Search and Strong Connectivity in Coq*	9.07
5 23-29 Sept	Exercises: SF IndProp Wrote Paper About: *Goedel's Incompleteness Theorem Essential Incompleteness of Arithmetic Verified by Coq* Discussed *Depth-First Search and Strong Connectivity in Coq*	5.92
6 30 Sept-6 Oct	Midterm Project Proposal Exercises: SF IndProp Discussed *Goedel's Incompleteness Theorem*	8.07
7 7-13 Oct	Wrote Paper About: *Efficient Certified Resolution Proof Checking*	6.98
8 14-20 Oct	Midterm Project Discussed *Efficient Certified Resolution Proof Checking*	6.18
9 21-27 Oct	Midterm Project Wrote Paper About: *A String of Pearls: Proofs of Fermat's Little Theorem*	8.3
10 28 Oct-3 Nov	Exercises: SF Maps Wrote Paper About: *Fundamental Theorem of Algebra* Discussed *A String of Pearls: Proofs of Fermat's Little Theorem*	8.37
11 4-10 Nov	Final Project Proposal Exercises: SF ProofObjects, IndPrinciples Wrote Paper About: *Improving Real Analysis in Coq: A User-Friendly Approach to Integrals and Derivatives* Discussed *Fundamental Theorem of Algebra*	6.8
12 11-17 Nov	Exercises: SF IndPrinciples, Rel Wrote Paper About: *Formal Proof - The Four Color Theorem, Notices of the American Mathematical Society, Vol. 55, No. 11, p1382 (2008)* Discussed *Improving Real Analysis in Coq: A User-Friendly Approach to Integrals and Derivatives*	8.07



13 18-24 Nov	Read: SF Auto, CPDT Mutually Inductive Types Discussed *Formal Proof - The Four Color Theorem, Notices of the American Mathematical Society, Vol. 55, No. 11, p1382 (2008)*	3.32
14 25 Nov-1 Dec	Read: CPDT Reflexive Types, An Interlude on Inductive Types, Nested Inductive Types Final Project	9.15
15 2-8 Dec	Read: CPDT Computing with Infinite Data, Infinite Proofs Final Project	6.27
16 9-14 Dec	Final Project and Essay Final Report	9.97
Total		136.68

5.1.2 Deviations from Schedule

The biggest deviation was in the pace for the SF chapters and exercises. I found it more difficult than anticipated. I suspect this was for several reasons: first, I did not have the usual background many people starting to learn Coq have. I had not taken Programming Language Theory (CSCI 740) and had not studied type theory, formal logic, syntax analysis, or parsing. Prior to that, I had not written much in functional programming languages, either. Because of that, I spent time in multiple classes asking about currying, structural recursion, or other concepts in functional programming or type theory. I nearly started reading up on type theory, but was redirected away from that path. Even so, I believe that Coq can be taught to people without understanding of these concepts. The use of Coq does not depend so much on a deep understanding of type theory or syntax analysis. It would have come in helpful when I could not follow what Coq was doing 'behind the scenes', but a brief understanding would have sufficed. Similarly, my lack of experience with functional programming and its related concepts was remedied by reading a few blog posts.

Additionally, I had to drop most of the content from CPDT for multiple reasons: (1) it used a different version of Coq than SF did, (2) the author of CPDT has structured the content very differently, making it harder to interweave chapters from books, and (3) it did not have as many maintained exercises.

Moreover, project proposals were added and the projects were also completely changed. The addition of project proposals was to ensure that the project ideas were feasible, which was wise. The project ideas I had before starting the independent study would have been unapproachable, given how much time and work was put into the simpler projects we settled upon.

6 Conclusion

If I were to redo the class, I would change the pace to emphasize material covered, not number of exercises completed and read each chapter before starting on the exercises. The fact that each chapter in SF is interactive makes it very tempting to attempt exercises before even finishing reading, but that would not leave much time for the new concepts to sink in. I might include more, smaller independent projects, since I felt that the format of SF does not encourage students to create projects from scratch. I do, however, understand SF is primarily an introduction book. I would also ask questions focused on conceptual understanding, not questions centered around the exercises.
Otherwise, I found the course extremely rewarding. The Software Foundations series of books are very self-guided for the most part and I intend to continue self-studying the material.