1 Introduction

1.1 Whence This Book?

We would all like to have programs check that our programs are correct. Due in no small part to some bold but unfulfilled promises in the history of computer science, today most people who write software, practitioners and academics alike, assume that the costs of formal program verification outweigh the benefits. The purpose of this book is to convince you that the technology of program verification is mature enough today that it makes sense to use it in a support role in many kinds of research projects in computer science. Beyond the convincing, I also want to provide a handbook on practical engineering of certified programs with the Coq proof assistant. Almost every subject covered is also relevant to interactive computer theorem proving in general, such as for traditional mathematical theorems. In fact, I hope to demonstrate how verified programs are useful as building blocks in all sorts of formalizations.

Research into mechanized theorem proving began in the second half of the twentieth century, and some of the earliest practical work involved Nqthm [3], the Boyer-Moore theorem prover, which was used to prove such theorems as correctness of a complete hardware and software stack [25]. ACL2 [17], Nqthm's successor, has seen significant industry adoption, for instance, by Advanced Micro Devices, Inc. (AMD) to verify correctness of floating-point division units [26].

At the beginning of the twenty-first century, the pace of progress in practical applications of interactive theorem proving accelerated. Several well-known formal developments have been carried out in Coq, the system that this book deals with. In the realm of pure mathematics, Georges Gonthier built a machine-checked proof of the four-color theorem [13], a mathematical problem first posed more than one hundred years before, where the only previous proofs had required trusting ad hoc software to do brute force checking of key facts. In the realm of program verification, Xavier Leroy led the CompCert project to produce a verified C compiler back-end [19] robust enough to use with real embedded software.

Many other projects have attracted attention by proving important theorems using computer proof assistant software. For instance, the L4.verified project [18] led by Gerwin Klein has given a mechanized proof of correctness for a realistic microkernel, using the Isabelle/HOL proof assistant [29]. The amount of ongoing work in the area is so large that I cannot hope to list it all, so from this point I assume that readers are convinced that we ought to want machine-checked proofs and that such proofs are feasible to produce. (To readers not yet convinced, I suggest a Web search for "machine-checked proof.")

The idea of *certified program* features prominently in this book's title. Here "certified" does not refer to government rules for how the reliability of engineered systems may be demonstrated to sufficiently high standards. Rather, this concept of certification, a standard one in the programming languages and formal methods communities, has to do with the idea of a *certificate*, or formal mathematical artifact proving that a program meets its specification. Government certification procedures rarely provide strong mathematical guarantees, whereas certified programming provides guarantees about as strong as anything we could hope for. We trust the definition of a foundational mathematical logic, we trust an implementation of that logic, and we trust that we have encoded our informal intent properly in formal specifications, but few other opportunities remain to certify incorrect software. For compilers and other programs that run in batch mode, the notion of a *certifying program* is also common, where each run of the program outputs both an answer and a proof that the answer is correct. Any certifying program can be composed with a proof checker to produce a certified program, and this book focuses on the certified case while also introducing principles and techniques of general interest for stating and proving theorems in Coq.

A good number of tools are in wide use today for building machine-checked mathematical proofs and machine-certified programs. The following is a list of interactive proof assistants satisfying a few criteria. First, the authors of each tool must intend for it to be put to use for software-related applications. Second, there must have been enough engineering effort put into the tool that someone not doing research on the tool itself would feel time was well spent using it. A third criterion is an empirical validation of the second: the tool must have a significant user community outside its own development team.

ACL2	`http://www.cs.utexas.edu/users/moore/acl2/`
Coq	`http://coq.inria.fr/`
Isabelle/HOL	`http://isabelle.in.tum.de/`
PVS	`http://pvs.csl.sri.com/`
Twelf	`http://www.twelf.org/`

Isabelle/HOL, implemented with the proof assistant development framework Isabelle [32], is the most popular proof assistant for the HOL logic. The other implementations of HOL can be considered equivalent for purposes of the discussion here.

1.2 Why Coq?

This book is about certified programming using Coq, and I am convinced that it is the best tool for the job. Coq has a number of very attractive properties, which I summarize here, mentioning which of the other candidate tools lack which properties.

1.2.1 Based on a Higher-Order Functional Programming Language

There is no reason to give up the familiar comforts of functional programming when you start writing certified programs. All the tools I listed are based on functional programming languages and can be used without their proof-related features to write and run regular programs.

ACL2 is notable in this field for having only a *first-order language* at its foundation. That is, you cannot work with functions over functions and all those other treats of functional programming. By giving up this facility, ACL2 can make broader assumptions about how well its proof automation will work, but we can generally recover the same advantages in other proof assistants when we are programming in first-order fragments.

1.2.2 Dependent Types

A language with *dependent types* may include references to programs inside of types. For instance, the type of an array might include a program expression giving the size of the array, making it possible to verify absence of out-of-bounds accesses statically. Dependent types can go even further than this, effectively capturing any correctness property in a type. For instance, Section 2.2.3 introduces the technique of giving a compiler a type that guarantees that it maps well-typed source programs to well-typed target programs.

ACL2 and HOL lack dependent types outright. Each of PVS and Twelf supports a different strict subset of Coq's dependent type language. Twelf's type language is restricted to a bare-bones, monomorphic lambda calculus, which places serious restrictions on how complicated *computations inside types* can be. This restriction is important for the soundness argument behind Twelf's approach to representing and checking proofs.

In contrast, PVS's dependent types are much more general, but they are squeezed inside the single mechanism of *subset types*, where a normal type is refined by attaching a predicate over its elements. Each member of the subset type is an element of the base type that satisfies the predicate. Chapter 6 introduces that style of programming in Coq, and Chapters 7–12 deal with features of dependent typing in Coq that go beyond what PVS supports.

Dependent types are useful not only because they help express correctness properties in types. Dependent types also enable writing certified programs without writing anything that looks like a proof. Even with subset types, which in many contexts can be used to express any relevant property, the human driving the proof assistant usually has to build some proofs explicitly. Writing formal proofs is hard, so we want to avoid it as far as possible. Dependent types are invaluable for this purpose.

1.2.3 An Easy-to-Check Kernel Proof Language

Scores of automated decision procedures are useful in practical theorem proving, but it is unfortunate to have to trust in the correct implementation of each procedure. Proof assistants satisfy the de Bruijn criterion when they produce proof terms in small kernel languages, even when they use complicated and extensible procedures to seek out proofs in the first place. These core languages feature complexity on par with that in proposals for formal foundations for mathematics (e.g., ZF set theory). To believe a proof, we can ignore the possibility of bugs during *search* and just rely on a (relatively small) proof-checking kernel applied to the *result* of the search.

Coq meets the de Bruijn criterion, whereas ACL2 does not, because it employs fancy decision procedures that produce no evidence trails justifying their results. PVS supports *strategies* that implement fancier proof procedures in terms of a set of primitive proof steps, where the primitive steps are less primitive than in Coq. For instance, a propositional tautology solver is included as a primitive, so it is a question of taste whether such a system meets the de Bruijn criterion. The HOL implementations meet the de Bruijn criterion more manifestly; for Twelf, the situation is murkier.

1.2.4 Convenient Programmable Proof Automation

A commitment to a kernel proof language opens up wide possibilities for user extension of proof automation systems without allowing user mistakes to trick the overall system into accepting invalid proofs. Almost any interesting verification problem is undecidable, so it is important to help users build their own procedures for solving the restricted problems that they encounter in particular theorems.

Twelf features no proof automation marked as a bona fide part of the latest release; some automation code is included for testing purposes. The Twelf style is based on writing out all proofs in full detail. Because Twelf is specialized to the domain of syntactic metatheory proofs about programming languages and logics, it is feasible to use it to write those kinds of proofs manually. Outside that domain, the lack of automation can be a serious obstacle to productivity. Most kinds of program verification fall outside Twelf's scope.

Of the remaining tools, all can support user extension with new decision procedures by hacking directly in the tool's implementation language (such as OCaml for Coq). Since ACL2 and PVS do not satisfy the de Bruijn criterion (or at least do not satisfy it as strongly as Coq does), overall correctness is at the mercy of the authors of new procedures.

Isabelle/HOL and Coq both support coding new proof manipulations in ML in ways that cannot lead to the acceptance of invalid proofs. Additionally, Coq includes a domain-specific language for coding decision procedures in normal Coq source code, with no need to break out into ML. This language is called Ltac, and I think of it as the unsung hero of the proof assistant world. Not only does Ltac prevent the programmer from making fatal mistakes but it also includes a number of novel programming constructs that combine to make a "proof by decision procedure" style very pleasant.

1.2.5 Proof by Reflection

A surprising wealth of benefits follows from choosing a proof language that integrates a rich notion of computation. Coq includes programs and proof terms in the same syntactic class. This makes it easy to write programs that compute proofs. With rich enough dependent types, such programs are *certified decision procedures*. In such cases, these certified procedures can be put to good use without ever running them. Their types guarantee that if we did run them, we would receive proper ground proofs.

The critical ingredient for this technique, many of whose instances are referred to as *proof by reflection*, is a way of inducing nontrivial computation inside of logical propositions during proof checking. Further, most of these instances require dependent types in order to state the appropriate theorems. Of the proof assistants I listed, only Coq really provides support for the type-level computation style of reflection, though PVS supports very similar functionality via refinement types.

1.3 Why Not a Different Dependently Typed Language?

The logic and programming language behind Coq belongs to a type theory ecosystem with a good number of other thriving members. Agda[1] and Epigram[2] are the most developed tools among the alternatives to Coq, and there are others that are earlier in their life cycles. All the languages in this family are like different historical offshoots of Latin. The hardest conceptual epiphanies are, for the most part, portable among all the languages. Given this, why choose Coq for certified programming?

I think the answer is simple. None of the others has a well-developed system for tactic-based theorem proving. Agda and Epigram are designed and marketed more as programming languages than as proof assistants. Dependent types are helpful for proving deep theorems without needing anything that feels like proving. Nonetheless, almost any interesting certified programming project will benefit from some activity that deserves to be called proving, and many interesting projects absolutely require semiautomated proving to protect the sanity of the programmer. Informally, proving is unavoidable when any correctness

1. `http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php`
2. `http://www.e-pig.org/`

proof for a program has a structure that does not mirror the structure of the program itself. An example is a compiler correctness proof, which probably proceeds by induction on program execution traces, which have no simple relation with the structure of the compiler or the structure of the programs it compiles. In building such proofs, a mature system for scripted proof automation is invaluable.

On the other hand, Agda, Epigram, and similar tools have less implementation baggage associated with them, so they tend to be the default first homes of innovations in practical type theory. Some kinds of dependently typed programs are much easier to write in Agda and Epigram than in Coq. The former tools may very well be superior choices for projects that do not involve any proving. Anecdotally, I have gotten the impression that manual proving is orders of magnitude more costly than manual coping with Coq's lack of programming bells and whistles. In this book, I devote substantial space to patterns for programming with dependent types in Coq as it is today. We can hope that the type theory community is tending toward convergence on the right set of features for practical programming with dependent types and that eventually a single tool will embody those features.

1.4 Engineering with a Proof Assistant

In comparisons with its competitors, Coq is often derided for promoting unreadable proofs. It is easy to write proof scripts that manipulate proof goals imperatively, with no structure to aid readers. Such developments are nightmares to maintain, and they certainly do not convey why the theorem is true to anyone but the original author. An additional purpose of this book is to show why it is unfair and unproductive to dismiss Coq based on the existence of such developments.

You may favor some programming language and may have taught that language to undergraduates. I want to propose an analogy between coming to a negative conclusion about Coq after reading common Coq developments and coming to a negative conclusion about your favorite language after looking at the programs undergraduates write in it in the first week of class. The pragmatics of mechanized proving and program verification have been under serious study for much less time than the pragmatics of programming have been. The computer theorem–proving community is still developing the key insights corresponding to those that programming texts and instructors impart to new learners. Most of the insights for Coq are barely disseminated among experts, let alone set down in tutorial form. I hope this book goes a long way toward remedying that.

This book should be of interest even to people who have participated in classes or tutorials specifically about Coq. It should also be useful to those who have been using Coq for years but whose Coq developments prove impenetrable to colleagues. This book emphasizes that there are design patterns for reliably avoiding the tedious parts of theorem proving and that consistent use of these patterns can get you over the hump to the point where it is worth using Coq to prove your theorems and certify your programs even if formal verification is not your main concern. I follow this theme by pursuing two main methods for replacing manual proofs with more understandable artifacts: dependently typed functions and custom Ltac decision procedures.

1.5 Prerequisites

I try to keep the required background knowledge to a minimum in this book. I assume familiarity with the material from usual discrete math and logic courses taken by undergraduate computer science majors. I also assume significant experience programming in one of the ML dialects, in Haskell, or in some other closely related language. Experience with only dynamically typed functional languages might lead to befuddlement in some places, but readers who have come to understand Scheme deeply will probably be fine.

My background is in programming languages, formal semantics, and program verification. I sometimes use examples from that domain. As a reference on these topics, I recommend *Types and Programming Languages* [36], by Benjamin C. Pierce; however, I have tried to choose examples so that they may be understood without background in semantics.

1.6 Using This Book

This book is generated automatically from Coq source files using the coqdoc program. The latest PDF version, with hyperlinks from identifier uses to the corresponding definitions, is available at

`http://adam.chlipala.net/cpdt/cpdt.pdf`

An online HTML version is available, which also provides hyperlinks:

`http://adam.chlipala.net/cpdt/html/toc.html`

The source code for the book is freely available at

`http://adam.chlipala.net/cpdt/cpdt.tgz`

There, you can find all the code appearing in this book, with prose interspersed in comments, in exactly the order that you find here. You can step through the code interactively with your chosen graphical Coq interface. The code also has special comments indicating which parts of the chapters make suitable starting points for interactive class sessions, in which the class works together to construct the programs and proofs. The included Makefile has a target `templates` for building a fresh set of class template files automatically from the book source. The online versions will remain available at no cost, and I intend to keep the source code up-to-date with bug fixes and compatibility changes to track new Coq releases.

I believe that a good graphical interface to Coq is crucial for using it productively. I use the Proof General[3] mode for Emacs, which supports a number of other proof assistants besides Coq. There is also the standalone CoqIDE program developed by the Coq team. I like being able to combine certified programming and proving with other kinds of work inside the same full-featured editor. In the initial part of this book, I reference Proof General procedures explicitly in introducing how to use Coq, but most of the book is interface-agnostic, so feel free to use CoqIDE if you prefer it. The one issue with CoqIDE before version 8.4, regarding running through the book source, is that I sometimes begin a proof attempt but cancel it with the Coq `Abort` or `Restart` commands, which CoqIDE did not support until recently. It would be bad form to leave such commands intact in a finished development, but I find these commands helpful in writing single source files that trace a user's thought process in designing a proof.

1.6.1 Reading This Book

For experts in functional programming or formal methods, learning to use Coq is not hard. The Coq manual [7], the textbook by Bertot and Castéran [1], and Pierce et al.'s *Software Foundations*[4] have helped many people become productive Coq users. However, I believe that the best ways to manage significant Coq developments are far from settled. In this book, I propose my own techniques, and I employ them from

3. `http://proofgeneral.inf.ed.ac.uk/`
4. `http://www.cis.upenn.edu/~bcpierce/sf/`

the very beginning. After a first chapter showing off what can be done with dependent types, I retreat into simpler programming styles for the first part of the book. The other main thrust of the book, Ltac proof automation, is adopted from the start of the technical exposition.

Readers new to Coq and experienced Coq hackers can both benefit from reading Part I, which devotes substantial space to basic concepts, because as I introduce these concepts, I also develop my preferred automated proof style. I hope that my heavy reliance on proof automation early on will seem like the most natural way to go.

Coq is a very complex system, with many different commands driven more by pragmatic concerns than by any overarching aesthetic principle. When I use some construct for the first time, I try to give a one-sentence intuition for what it accomplishes, but I leave the details to the Coq reference manual [7]. I expect that readers interested in complete understanding will consult that manual frequently. I often use constructs in code snippets without first introducing them, but explanations follow in succeeding paragraphs.

Previous versions of the book included some suggested exercises at the ends of chapters. Since then, I have decided to remove the exercises and focus on the main book exposition. A database of exercises proposed by readers is available on the Web.[5] I do want to suggest, though, that the best way to learn Coq is to get started applying it in a real project rather than focusing on artificial exercises.

1.6.2 The Tactic Library

To make it possible to start from fancy proof automation rather than working up to it, I have included with the book source a library of *tactics*, or programs that find proofs, since the built-in Coq tactics do not support a high enough level of automation. I use these tactics from the first chapter with code examples.

Some readers have asked about the pragmatics of using this tactic library in their own developments. My position is that this tactic library was designed with the specific examples of the book in mind; I do not recommend using it in other settings. Part III should impart the necessary skills to reimplement these tactics and beyond. One generally deals with undecidable problems in interactive theorem proving, so no tactic can solve all goals, though the *crush* tactic that we will meet soon may sometimes seem as if it can. There are still very useful tricks found

5. `http://adam.chlipala.net/cpdt/ex/`

in the implementations of *crush* and its cousins, so it may be useful to examine the commented source file `CpdtTactics.v`. I implement a new tactic library for each new project because each project involves a different mix of undecidable theories where a different set of heuristics turns out to work well, and that is what I recommend others do, too.

1.6.3 Installation and Emacs Setup

At the start of the next chapter, I assume that you have installed Coq and Proof General. The code in this book is tested with Coq versions 8.4, 8.4pl1, and 8.4pl2. Though parts may work with other versions, it is expected that the book source will fail to build with earlier versions.

To set up your Proof General environment to process the source to the next chapter, a few simple steps are required:

1. Get the book source from

   `http://adam.chlipala.net/cpdt/cpdt.tgz`

2. Unpack the tarball to some directory `DIR`.
3. Run `make` in `DIR` (ideally with a `-j` flag to use multiple processor cores, if you have them).
4. There are some minor headaches associated with getting Proof General to pass the proper command line arguments to the `coqtop` program, which provides the interactive Coq toplevel. One way to add settings that will be shared by many source files is to add a custom variable setting to your `.emacs` file, like this:

```
(custom-set-variables
  ...
  '(coq-prog-args '("-I" "DIR/src"))
  ...
)
```

The extra arguments demonstrated here are the proper choices for working with the code for this book. The ellipses stand for other Emacs customization settings you may already have. It can be helpful to save several alternative sets of flags in your `.emacs` file, with all but one commented out within the `custom-set-variables` block at any given time.

Alternatively, Proof General configuration can be set on a per-directory basis, using a `.dir-locals.el` file in the directory of the source files for which you want the settings to apply. Here is an example that could be written in such a file to enable use of the book source. Note the need to include an argument that starts Coq in Emacs support mode.

```
((coq-mode . ((coq-prog-args .
  ("-emacs-U" "-I" "DIR/src")))))
```

Every chapter of this book is generated from a commented Coq source file. You can load these files and run through them step-by-step in Proof General. Be sure to run the Coq binary `coqtop` with the command line argument `-I DIR/src`. If you have installed Proof General properly, the Coq mode should start automatically when you visit a `.v` buffer in Emacs, and the preceding advice on `.emacs` settings should ensure that the proper arguments are passed to `coqtop` by Emacs.

With Proof General, the portion of a buffer that Coq has processed is highlighted in some way, such as being given a blue background. You step through Coq source files by positioning the point at the position you want Coq to run to and pressing C-C C-RET. This can be used both for normal step-by-step coding, by placing the point inside some command past the end of the highlighted region, and for undoing, by placing the point inside the highlighted region.

The chapter source files are as follows:

Chapter	**Source File**
Some Quick Examples	`StackMachine.v`
Introducing Inductive Types	`InductiveTypes.v`
Inductive Predicates	`Predicates.v`
Infinite Data and Proofs	`Coinductive.v`
Subset Types and Variations	`Subset.v`
General Recursion	`GeneralRec.v`
More Dependent Types	`MoreDep.v`
Dependent Data Structures	`DataStruct.v`
Reasoning About Equality Proofs	`Equality.v`
Generic Programming	`Generic.v`
Universes and Axioms	`Universes.v`
Proof Search by Logic Programming	`LogicProg.v`
Proof Search in Ltac	`Match.v`
Proof by Reflection	`Reflection.v`
Proving in the Large	`Large.v`
Reasoning about Programming Language Syntax	`ProgLang.v`