SOFTWARE FOUNDATIONS

VOLUME 1: LOGICAL FOUNDATIONS

TABLE OF CONTENTS INDEX ROADMAP

PREFACE

Welcome

This is the entry point in a series of electronic textbooks on various aspects of *Software Foundations* — the mathematical underpinnings of reliable software. Topics in the series include basic concepts of logic, computer-assisted theorem proving, the Coq proof assistant, functional programming, operational semantics, logics for reasoning about programs, and static type systems. The exposition is intended for a broad range of readers, from advanced undergraduates to PhD students and researchers. No specific background in logic or programming languages is assumed, though a degree of mathematical maturity will be helpful.

The principal novelty of the series is that it is one hundred percent formalized and machine-checked: each text is literally a script for Coq. The books are intended to be read alongside (or inside) an interactive session with Coq. All the details in the text are fully formalized in Coq, and most of the exercises are designed to be worked using Coq.

The files in each book are organized into a sequence of core chapters, covering about one semester's worth of material and organized into a coherent linear narrative, plus a number of "offshoot" chapters covering additional topics. All the core chapters are suitable for both upper-level undergraduate and graduate students.

This book, *Logical Foundations*, lays groundwork for the others, introducing the reader to the basic ideas of functional programming, constructive logic, and the Coq proof assistant.

Overview

Building reliable software is really hard. The scale and complexity of modern systems, the number of people involved, and the range of demands placed on them make it extremely difficult to build software that is even more-or-less correct, much less 100%

correct. At the same time, the increasing degree to which information processing is woven into every aspect of society greatly amplifies the cost of bugs and insecurities.

Computer scientists and software engineers have responded to these challenges by developing a whole host of techniques for improving software reliability, ranging from recommendations about managing software projects teams (e.g., extreme programming) to design philosophies for libraries (e.g., model-view-controller, publish-subscribe, etc.) and programming languages (e.g., object-oriented programming, aspect-oriented programming, functional programming, ...) to mathematical techniques for specifying and reasoning about properties of software and tools for helping validate these properties. The *Software Foundations* series is focused on this last set of techniques.

The text is constructed around three conceptual threads:

(1) basic tools from *logic* for making and justifying precise claims about programs;

(2) the use of *proof assistants* to construct rigorous logical arguments;

(3) *functional programming*, both as a method of programming that simplifies reasoning about programs and as a bridge between programming and logic.

Some suggestions for further reading can be found in the Postscript chapter. Bibliographic information for all cited works can be found in the file Bib.

Logic

Logic is the field of study whose subject matter is *proofs* — unassailable arguments for the truth of particular propositions. Volumes have been written about the central role of logic in computer science. Manna and Waldinger called it "the calculus of computer science," while Halpern et al.'s paper *On the Unusual Effectiveness of Logic in Computer Science* catalogs scores of ways in which logic offers critical tools and insights. Indeed, they observe that, "As a matter of fact, logic has turned out to be significiantly more effective in computer science than it has been in mathematics. This is quite remarkable, especially since much of the impetus for the development of logic during the past one hundred years came from mathematics."

In particular, the fundamental tools of *inductive proof* are ubiquitous in all of computer science. You have surely seen them before, perhaps in a course on discrete math or analysis of algorithms, but in this course we will examine them more deeply than you have probably done so far.

Proof Assistants

The flow of ideas between logic and computer science has not been unidirectional: CS has also made important contributions to logic. One of these has been the development of software tools for helping construct proofs of logical propositions. These tools fall into two broad categories:

- *Automated theorem provers* provide "push-button" operation: you give them a proposition and they return either *true* or *false* (or, sometimes, *don't know: ran*

out of time). Although their capabilities are still limited to specific domains, they have matured tremendously in recent years and are used now in a multitude of settings. Examples of such tools include SAT solvers, SMT solvers, and model checkers.

you state a claim and construct a value to prove that claim. so you proof check your program that is actually correct!

- *Proof assistants* are hybrid tools that automate the more routine aspects of building proofs while depending on human guidance for more difficult aspects. Widely used proof assistants include Isabelle, Agda, Twelf, ACL2, PVS, and Coq, among many others.

monkey at the keyboard strategy :D

This course is based around Coq, a proof assistant that has been under development since 1983 and that in recent years has attracted a large community of users in both research and industry. Coq provides a rich environment for interactive development of machine-checked formal reasoning. The kernel of the Coq system is a simple proof-checker, which guarantees that only correct deduction steps are ever performed. On top of this kernel, the Coq environment provides high-level facilities for proof development, including a large library of common definitions and lemmas, powerful tactics for constructing complex proofs semi-automatically, and a special-purpose programming language for defining new proof-automation tactics for specific situations.

Coq has been a critical enabler for a huge variety of work across computer science and mathematics:

- As a *platform for modeling programming languages*, it has become a standard tool for researchers who need to describe and reason about complex language definitions. It has been used, for example, to check the security of the JavaCard platform, obtaining the highest level of common criteria certification, and for formal specifications of the x_{86} and LLVM instruction sets and programming languages such as C.

- As an *environment for developing formally certified software and hardware*, Coq has been used, for example, to build CompCert, a fully-verified optimizing compiler for C, and CertiKos, a fully verified hypervisor, for proving the correctness of subtle algorithms involving floating point numbers, and as the basis for CertiCrypt, an environment for reasoning about the security of cryptographic algorithms. It is also being used to build verified implementations of the open-source RISC-V processor.

- As a *realistic environment for functional programming with dependent types*, it has inspired numerous innovations. For example, the Ynot system embeds "relational Hoare reasoning" (an extension of the *Hoare Logic* we will see later in this course) in Coq.

- As a *proof assistant for higher-order logic*, it has been used to validate a number of important results in mathematics. For example, its ability to include complex computations inside proofs made it possible to develop the first formally verified proof of the 4-color theorem. This proof had previously been

controversial among mathematicians because part of it included checking a large number of configurations using a program. In the Coq formalization, everything is checked, including the correctness of the computational part. More recently, an even more massive effort led to a Coq formalization of the Feit-Thompson Theorem — the first major step in the classification of finite simple groups.

By the way, in case you're wondering about the name, here's what the official Coq web site at INRIA (the French national research lab where Coq has mostly been developed) says about it: "Some French computer scientists have a tradition of naming their software as animal species: Caml, Elan, Foc or Phox are examples of this tacit convention. In French, 'coq' means rooster, and it sounds like the initials of the Calculus of Constructions (CoC) on which it is based." The rooster is also the national symbol of France, and C-o-q are the first three letters of the name of Thierry Coquand, one of Coq's early developers.

Functional Programming

The term *functional programming* refers both to a collection of programming idioms that can be used in almost any programming language and to a family of programming languages designed to emphasize these idioms, including Haskell, OCaml, Standard ML, F#, Scala, Scheme, Racket, Common Lisp, Clojure, Erlang, and Coq.

Functional programming has been developed over many decades — indeed, its roots go back to Church's lambda-calculus, which was invented in the 1930s, well before the first computers (at least the first electronic ones)! But since the early '90s it has enjoyed a surge of interest among industrial engineers and language designers, playing a key role in high-value systems at companies like Jane St. Capital, Microsoft, Facebook, and Ericsson.

The most basic tenet of functional programming is that, as much as possible, computation should be *pure*, in the sense that the only effect of execution should be to produce a result: it should be free from *side effects* such as I/O, assignments to mutable variables, redirecting pointers, etc. For example, whereas an *imperative* sorting function might take a list of numbers and rearrange its pointers to put the list in order, a pure sorting function would take the original list and return a *new* list containing the same numbers in sorted order.

A significant benefit of this style of programming is that it makes programs easier to understand and reason about. If every operation on a data structure yields a new data structure, leaving the old one intact, then there is no need to worry about how that structure is being shared and whether a change by one part of the program might break an invariant that another part of the program relies on. These considerations are particularly critical in concurrent systems, where every piece of mutable state that is shared between threads is a potential source of pernicious bugs. Indeed, a large part of the recent interest in functional programming in industry is due to its simpler behavior in the presence of concurrency.

Another reason for the current excitement about functional programming is related to the first: functional programs are often much easier to parallelize than their imperative counterparts. If running a computation has no effect other than producing a result, then it does not matter *where* it is run. Similarly, if a data structure is never modified destructively, then it can be copied freely, across cores or across the network. Indeed, the "Map-Reduce" idiom, which lies at the heart of massively distributed query processors like Hadoop and is used by Google to index the entire web is a classic example of functional programming.

For purposes of this course, functional programming has yet another significant attraction: it serves as a bridge between logic and computer science. Indeed, Coq itself can be viewed as a combination of a small but extremely expressive functional programming language plus a set of tools for stating and proving logical assertions. Moreover, when we come to look more closely, we find that these two sides of Coq are actually aspects of the very same underlying machinery — i.e., *proofs are programs*.

Further Reading

This text is intended to be self contained, but readers looking for a deeper treatment of particular topics will find some suggestions for further reading in the Postscript chapter.

Practicalities

Chapter Dependencies

A diagram of the dependencies between chapters and some suggested paths through the material can be found in the file `deps.html`.

System Requirements

Coq runs on Windows, Linux, and macOS. You will need:

- A current installation of Coq, available from the Coq home page. These files have been tested with Coq 8.7.1.
- An IDE for interacting with Coq. Currently, there are two choices:
    - Proof General is an Emacs-based IDE. It tends to be preferred by users who are already comfortable with Emacs. It requires a separate installation (google "Proof General").

      Adventurous users of Coq within Emacs may also want to check out extensions such as `company-coq` and `control-lock`.
    - CoqIDE is a simpler stand-alone IDE. It is distributed with Coq, so it should be available once you have Coq installed. It can also be compiled from scratch, but on some platforms this may involve installing additional packages for GUI libraries and such.

Exercises

Each chapter includes numerous exercises. Each is marked with a "star rating," which can be interpreted as follows:

- One star: easy exercises that underscore points in the text and that, for most readers, should take only a minute or two. Get in the habit of working these as you reach them.
- Two stars: straightforward exercises (five or ten minutes).
- Three stars: exercises requiring a bit of thought (ten minutes to half an hour).
- Four and five stars: more difficult exercises (half an hour and up).

Also, some exercises are marked "advanced," and some are marked "optional." Doing just the non-optional, non-advanced exercises should provide good coverage of the core material. Optional exercises provide a bit of extra practice with key concepts and introduce secondary themes that may be of interest to some readers. Advanced exercises are for readers who want an extra challenge and a deeper cut at the material.

Please do not post solutions to the exercises in a public place. Software Foundations is widely used both for self-study and for university courses. Having solutions easily available makes it much less useful for courses, which typically have graded homework assignments. We especially request that readers not post solutions to the exercises anyplace where they can be found by search engines.

Downloading the Coq Files

A tar file containing the full sources for the "release version" of this book (as a collection of Coq scripts and HTML files) is available at http://www.cis.upenn.edu/~bcpierce/sf.

(If you are using the book as part of a class, your professor may give you access to a locally modified version of the files, which you should use instead of the release version.)

Lecture Videos

Lectures for an intensive summer course based on *Logical Foundations* (part of the DeepSpec summer school in 2017) can be found at https://deepspec.org/event/dsss17/coq_intensive.html. The video quality is poor at the beginning but gets better in the later lectures.

Note for Instructors

If you plan to use these materials in your own course, you will undoubtedly find things you'd like to change, improve, or add. Your contributions are welcome!

In order to keep the legalities simple and to have a single point of responsibility in case the need should ever arise to adjust the license terms, sublicense, etc., we ask all contributors (i.e., everyone with access to the developers' repository) to assign copyright in their contributions to the appropriate "author of record," as follows:

- I hereby assign copyright in my past and future contributions to the Software Foundations project to the Author of Record of each volume or component, to be licensed under the same terms as the rest of Software Foundations. I understand that, at present, the Authors of Record are as follows: For Volumes 1 and 2, known until 2016 as "Software Foundations" and from 2016 as (respectively) "Logical Foundations" and "Programming Foundations," the Author of Record is Benjamin Pierce. For Volume 3, "Verified Functional Algorithms", the Author of Record is Andrew W. Appel. For components outside of designated Volumes (e.g., typesetting and grading tools and other software infrastructure), the Author of Record is Benjamin Pierce.

To get started, please send an email to Benjamin Pierce, describing yourself and how you plan to use the materials and including (1) the above copyright transfer text and (2) the result of doing "htpasswd -s -n NAME" where NAME is your preferred user name.

We'll set you up with access to the subversion repository and developers' mailing lists. In the repository you'll find a file `INSTRUCTORS` with further instructions.

Translations

Thanks to the efforts of a team of volunteer translators, *Software Foundations* can be enjoyed in Japanese at http://proofcafe.org/sf. A Chinese translation is also underway; you can preview it at https://coq-zh.github.io/SF-zh/.

Thanks

Development of the *Software Foundations* series has been supported, in part, by the National Science Foundation under the NSF Expeditions grant 1521523, *The Science of Deep Specification*.