SOFTWARE FOUNDATIONS

VOLUME 2: PROGRAMMING LANGUAGE FOUNDATIONS

TABLE OF CONTENTS INDEX ROADMAP

PREFACE

Welcome

This electronic book is a survey of basic concepts in the mathematical study of programs and programming languages. Topics include advanced use of the Coq proof assistant, operational semantics, Hoare logic, 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.

As with all of the books in the *Software Foundations* series, this one is one hundred percent formalized and machine-checked: the entire text is literally a script for Coq. It is 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 are organized into a sequence of core chapters, covering about one half 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.

The book builds on the material from *Logical Foundations* (*Software Foundations*, volume 1). It can be used together with that book for a one-semester course on the theory of programming languages. Or, for classes where students who are already familiar with some or all of the material in *Logical Foundations*, there is plenty of additional material to fill most of a semester from this book alone.

Overview

The book develops two main conceptual threads:

(1) formal techniques for *reasoning about the properties of specific programs* (e.g., the fact that a sorting function or a compiler obeys some formal specification); and

(2) the use of *type systems* for establishing well-behavedness guarantees for *all* programs in a given programming language (e.g., the fact that well-typed Java programs cannot be subverted at runtime).

Each of these is easily rich enough to fill a whole course in its own right, and tackling both together naturally means that much will be left unsaid. Nevertheless, we hope readers will find that these themes illuminate and amplify each other and that bringing them together creates a good foundation for digging into any of them more deeply. 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.

Program Verification

In the first part of the book, we introduce two broad topics of critical importance in building reliable software (and hardware): techniques for proving specific properties of particular *programs* and for proving general properties of whole programming *languages*.

For both of these, the first thing we need is a way of representing programs as mathematical objects, so we can talk about them precisely, plus ways of describing their behavior in terms of mathematical functions or relations. Our main tools for these tasks are *abstract syntax* and *operational semantics*, a method of specifying programming languages by writing abstract interpreters. At the beginning, we work with operational semantics in the so-called "big-step" style, which leads to simple and readable definitions when it is applicable. Later on, we switch to a lower-level "small-step" style, which helps make some useful distinctions (e.g., between different sorts of nonterminating program behaviors) and which is applicable to a broader range of language features, including concurrency.

The first programming language we consider in detail is *Imp*, a tiny toy language capturing the core features of conventional imperative programming: variables, assignment, conditionals, and loops.

We study two different ways of reasoning about the properties of Imp programs. First, we consider what it means to say that two Imp programs are *equivalent* in the intuitive sense that they exhibit the same behavior when started in any initial memory state. This notion of equivalence then becomes a criterion for judging the correctness of *metaprograms* — programs that manipulate other programs, such as compilers and optimizers. We build a simple optimizer for Imp and prove that it is correct.

Second, we develop a methodology for proving that a given Imp program satisfies some formal specifications of its behavior. We introduce the notion of *Hoare triples* — Imp programs annotated with pre- and post-conditions describing what they expect to be true about the memory in which they are started and what they promise to make true about the memory in which they terminate — and the reasoning principles of

Hoare Logic, a domain-specific logic specialized for convenient compositional reasoning about imperative programs, with concepts like "loop invariant" built in.

This part of the course is intended to give readers a taste of the key ideas and mathematical tools used in a wide variety of real-world software and hardware verification tasks.

Type Systems

Our other major topic, covering approximately the second half of the book, is *type systems* — powerful tools for establishing properties of *all* programs in a given language.

Type systems are the best established and most popular example of a highly successful class of formal verification techniques known as *lightweight formal methods*. These are reasoning techniques of modest power — modest enough that automatic checkers can be built into compilers, linkers, or program analyzers and thus be applied even by programmers unfamiliar with the underlying theories. Other examples of lightweight formal methods include hardware and software model checkers, contract checkers, and run-time monitoring techniques.

This also completes a full circle with the beginning of the book: the language whose properties we study in this part, the *simply typed lambda-calculus*, is essentially a simplified model of the core of Coq itself!

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.

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! Please see the Preface to *Logical Foundations* for instructions.

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*.