Program Proofs

K. Rustan M. Leino

Illustrated by Kaleb Leino

The MIT Press
Cambridge, Massachusetts
London, England

The MIT Press would like to thank the anonymous peer reviewers who provided comments on drafts of this book. The generous work of academic experts is essential for establishing the authority and quality of our publications. We acknowledge with gratitude the contributions of these otherwise uncredited readers.

This book was set in TEX Gyre Pagella, Bera Mono, and Noto Emoji by the author. Printed and bound in the United States of America.

Illustrated by Kaleb Leino.

Library of Congress Cataloging-in-Publication Data is available.

ISBN: 978-0-262-54623-2

10 9 8 7 6 5 4 3 2 1

Contents

Preface ix
Notes for Teachers xv

0. Introduction 1
0.0. Prerequisites . . . 2
0.1. Outline of Topics . . . 4
0.2. Dafny . . . 5
0.3. Other Languages . . . 6

Part 0. Learning the Ropes

1. Basics 9
1.0. Methods . . . 9
1.1. Assert Statements . . . 10
1.2. Working with the Verifier . . . 11
1.3. Control Paths . . . 12
1.4. Method Contracts . . . 13
1.5. Functions . . . 17
1.6. Compiled versus Ghost . . . 19
1.7. Summary . . . 21
2. Making It Formal 25
2.0. Program State . . . 26
2.1. Floyd Logic . . . 28
2.2. Hoare Triples . . . 29
2.3. Strongest Postconditions and Weakest Preconditions . . . 32
2.4. $\mathcal{WP}$ and $\mathcal{SP}$ . . . 40
2.5. Conditional Control Flow . . . 41
2.6. Sequential Composition . . . 45
2.7. Method Calls and Postconditions . . . 46
2.8. Assert Statements . . . 50
2.9. Weakest Liberal Preconditions . . . 53
2.10. Method Calls with Preconditions . . . 55
2.11. Function Calls . . . 57

2.12. Partial Expressions . . . 58
2.13. Method Correctness . . . 60
2.14. Summary . . . 60
3. Recursion and Termination 63
3.0. The Endless Problem . . . 64
3.1. Avoiding Infinite Recursion . . . 66
3.2. Well-Founded Relations . . . 70
3.3. Lexicographic Tuples . . . 72
3.4. Default **`decreases`** in Dafny . . . 79
3.5. Summary . . . 80
4. Inductive Datatypes 83
4.0. Blue-Yellow Trees . . . 84
4.1. Matching on Datatypes . . . 85
4.2. Discriminators and Destructors . . . 86
4.3. Structural Inclusion . . . 88
4.4. Enumerations . . . 89
4.5. Type Parameters . . . 89
4.6. Abstract Syntax Trees for Expressions . . . 90
4.7. Summary . . . 93
5. Lemmas and Proofs 95
5.0. Declaring a Lemma . . . 96
5.1. Using a Lemma . . . 96
5.2. Proving a Lemma . . . 99
5.3. Back to Basics . . . 102
5.4. Proof Calculations . . . 106
5.5. Example: Reduce . . . 110
5.6. Example: Commutativity of Multiplication . . . 115
5.7. Example: Mirroring a Tree . . . 118
5.8. Example: Working on Abstract Syntax Trees . . . 122
5.9. Summary . . . 130

Part 1. Functional Programs

6. Lists 137
6.0. List Definition . . . 137
6.1. `Length` . . . 138
6.2. Intrinsic versus Extrinsic Specifications . . . 139
6.3. `Take` and `Drop` . . . 142
6.4. `At` . . . 144
6.5. `Find` . . . 146
6.6. List Reversal . . . 147
6.7. Lemmas in Expressions . . . 151
6.8. Eliding Type Arguments . . . 157
6.9. Summary . . . 158

7. Unary Numbers 161
7.0. Basic Definitions . . . 162
7.1. Comparisons . . . 162
7.2. Addition and Subtraction . . . 165
7.3. Multiplication . . . 167
7.4. Division and Modulus . . . 167
7.5. Summary . . . 172
8. Sorting 175
8.0. Specification . . . 175
8.1. Insertion Sort . . . 179
8.2. Merge Sort . . . 181
8.3. Summary . . . 188
9. Abstraction 189
9.0. Grouping Declarations into Modules . . . 190
9.1. Module Imports . . . 190
9.2. Export Sets . . . 191
9.3. Modular Specification of a Queue . . . 194
9.4. Equality-Supporting Types . . . 201
9.5. Summary . . . 204
10. Data-Structure Invariants 207
10.0. Priority-Queue Specification . . . 208
10.1. Designing the Data Structure . . . 210
10.2. Implementation . . . 212
10.3. Making Intrinsic from Extrinsic . . . 224
10.4. Summary . . . 229

Part 2. Imperative Programs

11. Loops 235
11.0. Loop Specifications . . . 235
11.1. Loop Implementations . . . 241
11.2. Loop Termination . . . 247
11.3. Summarizing the Loop Rule . . . 250
11.4. Integer Square Root . . . 252
11.5. Summary . . . 255
12. Recursive Specifications, Iterative Programs 257
12.0. Iterative Fibonacci . . . 257
12.1. Fibonacci Squared . . . 260
12.2. Powers of 2 . . . 264
12.3. Sums . . . 267
12.4. Summary . . . 272
13. Arrays and Searching 275
13.0. About Arrays . . . 275
13.1. Linear Search . . . 280

13.2. Binary Search . . . 288
13.3. Minimum . . . 292
13.4. Coincidence Count . . . 294
13.5. Slope Search . . . 301
13.6. Canyon Search . . . 304
13.7. Majority Vote . . . 309
13.8. Summary . . . 318
14. Modifying Arrays 321
14.0. Simple Frames . . . 321
14.1. Basic Array Modification . . . 326
14.2. Summary . . . 336
15. In-situ Sorting 337
15.0. Dutch National Flag . . . 337
15.1. Selection Sort . . . 341
15.2. Quicksort . . . 343
15.3. Summary . . . 347
16. Objects 351
16.0. Checksums . . . 352
16.1. Tokenizer . . . 359
16.2. Simple Aggregate Objects . . . 364
16.3. Full Aggregate Objects . . . 374
16.4. Summary . . . 382
17. Dynamic Heap Data Structures 387
17.0. Lazily Initialized Arrays . . . 387
17.1. Extensible Array . . . 396
17.2. Binary Search Tree for a Map . . . 403
17.3. Iterator for the Map . . . 413
17.4. Summary . . . 423

A. Dafny Syntax Cheat Sheet 427
B. Boolean Algebra 433
B.0. Boolean Values and Negation . . . 433
B.1. Conjunction . . . 433
B.2. Predicates and Well-Definedness . . . 434
B.3. Disjunction and Proof Format . . . 435
B.4. Implication . . . 437
B.5. Proving Implications . . . 438
B.6. Free Variables and Substitution . . . 439
B.7. Universal Quantification . . . 441
B.8. Existential Quantification . . . 442
C. Answers to Select Exercises 445
References 459
Index 467

Preface

Welcome to Program Proofs!

I've designed this book to teach a practical understanding of what it means to write specifications for code and what it means for code to satisfy the specifications. In this preface, I want to tell you about the book itself and how to use it.

Programs and Proofs

When I first learned about program verification, all program developments and proofs were done by hand. I loved it. But I think I was the only one in the class who did. Even if you do love it, it's not clear how to connect the activity you have mastered on paper with the activity of sitting in front of a computer trying to get a program to work. And if you didn't love the proofs in the first place and didn't get enough practice to master them, it's not clear you make any connection whatsoever between these two activities.

To bring the two activities closer together, you need to get experience in seeing the proofs at work in a programming language that the computer recognizes. And playing out the activity of writing specifications and proofs together with programs has the additional benefit that the computer can check the proofs for you. This way, you get instant feedback that helps you understand what the proofs are all about. Instead of turning in your handwritten homework and getting it back from the teaching assistant a week later (when you have forgotten what the exercises were about and the teaching assistant's comments on your paper seem less important than next week's looming assignment), you can *interact* with the automated verifier dozens of times in a short sitting, all in the context of the program you're writing!

Trying to teach program-proof concepts in the setting of an actual programming language may seem like madness. Most languages were not designed for verification, and trying to bolt specification and proof-authoring features onto such a language is at best clumsy. Moreover, if you'd have to learn a separate notation for writing proofs or interacting with the automated verifier, the burden on the learner becomes even much greater. To really connect the program and proof activities, I argue you want to teach verification in terms of software-engineering concepts (like preconditions, invariants, and assertions), not in terms of induction schemas, semantics-mapping transforms,

and prover directives.

Luckily, there are several programming languages designed to support specifications and proofs (so-called *verification-aware languages*), and there are integrated development environments (IDEs) that run the automated verifiers (sometimes known as *auto-active* verifiers: automated tooling that offers interaction via the program text [82]). Among these are the functional languages WhyML [20] and F* [53], the Ada-based SPARK language [43, 117], the object-oriented language Eiffel [89, 44, 121], the imperative languages GRASShopper [126] and Whiley [109], and—what I use in this book—Dafny [76, 78, 35]. In a similar spirit, but with annotation languages that have been added to existing programming languages are ACL2 (for Applicative Common LISP) [71], VeriFast (for C and Java) [64], the KeY toolset (for Java) [2], OpenJML (for Java with JML annotations) [105, 26, 66], the Frama-C toolset (for C) [14], Stainless (for Scala) [118], Prusti (for Rust) [5], Nagini (for Python) [45], Gobra (for Go) [4], and LiquidHaskell (for Haskell) [86]. In the notes at the end of chapters, I occasionally point out some alternative notation or other differences with these other tools, so as to make the concepts and experiences taught in this book readily applicable to those language settings as well.

Material

I have written this book to support the level of a second-year university course in computer science. It can also be used as a comprehensive introduction for industrial software engineers who are new to specification and verification and want to apply such techniques in their work.

The book assumes basic knowledge of programs and programming. The style of this prior programming (functional, imperative) and the particular prior language used are not so important, but it is helpful if the prior programming has not completely ignored the concept of types.

The book also assumes some basics of logic. The "and", "or", and "not" operators from programming will go a long way, but some fluency with implication (logical consequence) is also important. For example, a reader is expected to feel comfortable with the meaning of a formula like

```
2 <= x  ==>  10 <= 4 * (x + 1)
```

The book's Appendix B reviews some useful logic rules, but is hardly suitable as a first introduction to logic. For that, I would recommend a semester course in logic.

Beyond the basics of logic, concepts like mathematical induction and well-founded orderings play a role in program proofs. The book explains these concepts as needed.

The book is divided into three parts. Part 0 covers some foundations, leading up to writing proofs. After that, Part 1 focuses on (specifications and proofs of) functional programs and Part 2 on imperative programs. Other than occasional references between these parts, Parts 1 and 2 are independent of each other.

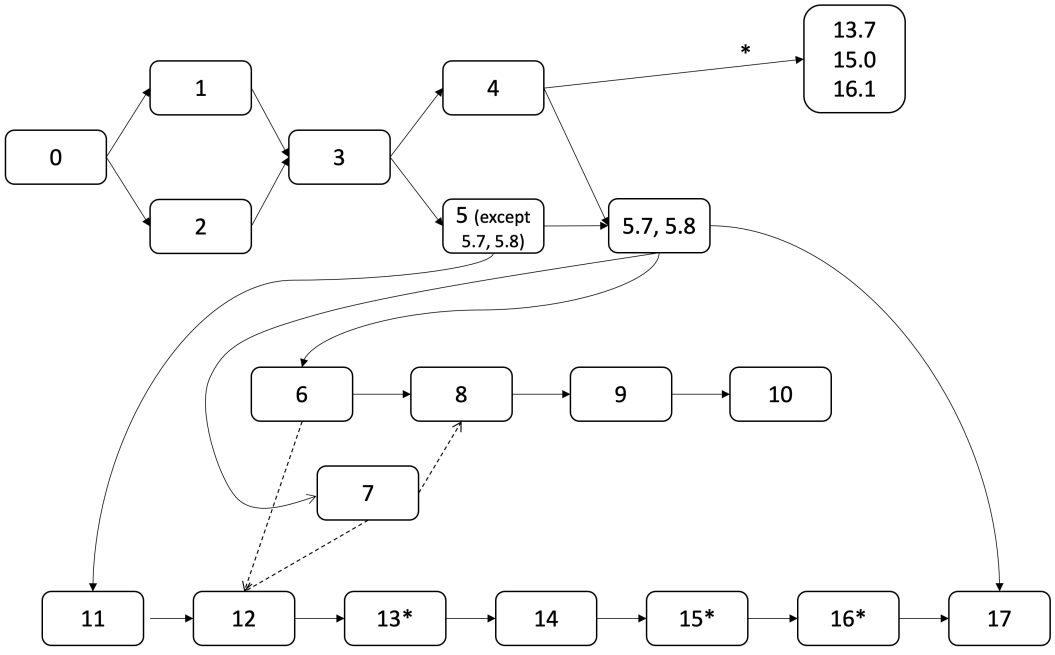
What the Book Is Not

Here are some things this book is not:

- It is not a beginner's guide to programming. The book assumes the reader has written (and compiled and run) basic programs in either a functional or imperative language. This seems like a reasonable assumption for a second-year university course in computer science.
- It is not a beginner's guide to logic, but see Appendix B for a review of some useful logic rules and some exercises.
- It is not a Dafny language guide or reference manual. The focus is on teaching program proofs. The book explains the Dafny constructs in the way they are used to support this learning, and Appendix A provides a cheat sheet for the language.
- It is not a research survey. There are many (mature or under-development) program-reasoning techniques that are not covered. There are also many useful programming paradigms that are not covered. The mathematics or motivations behind those advanced techniques are outside the scope of this book. Instead, this book focuses on teaching basic concepts and includes best practices for doing so.
- The book does not teach how to *build* a program verifier. Indeed, throughout this book, I treat the verifier as a black box. A recurring theme is the process of building proofs manually, which is good practice for interacting with any verifier.

How to Read This Book

Here is a rough chapter dependency graph:



Sections 13.7, 15.0, and 16.1 depend on Chapter 4, but the rest of their enclosing chapters do not. The dotted lines show recommended dependencies—it would be beneficial, but not absolutely required, to study Chapter 7 before Chapters 8 and 12, and likewise to study Chapter 6 before Chapter 12.

Dafny

All specifications, programs, and program proofs in the book use the Dafny programming language and can be checked in the Dafny verification system. Broadly speaking, the constructs of the Dafny language support four kinds of activities.

- There are constructs for imperative programming, such as assignment statements, loops, arrays, and dynamically allocated objects. The simpler of these are the bread and butter of many classic treatments of program proofs.
- There are constructs for functional programming, such as recursive functions and algebraic datatypes. In Dafny, these behave like in mathematics; for example, functions are deterministic and cannot change the program state.
- There are constructs for writing specifications, such as preconditions, loop invariants, and termination metrics. The way these are integrated into the language has been influenced by the pioneering Eiffel language and the Java Modeling Language (JML). Specifications can use any of the functional-language features, which makes them quite expressive.
- Lastly, there are constructs for proof authoring, such as lemmas and proof calculations.

These various features blend together. For example, all the constructs use the same expression language; these expressions include chaining expressions (like `0 <= x < y < 100`), implication (`==>`), quantifications (**`forall`**, **`exists`**), and sets (like `{2, 3, 5}`), which are often found in specifications and math, but can also be used in programs; methods, functions, and proofs bind values to local variables in the same way; in a method, an **`if`** statement divides up control flow, and in a lemma, it divides up proof obligations; variables can be marked as **`ghost`**, which makes them suitable for abstraction, but otherwise behave as ordinary compiled variables; and induction is achieved simply by calling a lemma recursively, where termination is specified and checked in the same way as for methods and functions.

Not only is the Dafny language versatile, but so are its uses. The Dafny development tools are quick to install and are available on Windows, MacOS, and Linux. The verifier runs automatically in the VS Code integrated development environment. Dafny programs compile to executable code for several language platforms, including .NET, Java, JavaScript, and Go. The toolset itself is available as open source at

```
github.com/dafny-lang/dafny
```

Even before this book, Dafny has been used in teaching for over a decade. It has also been used in several impressive research projects (for example, at Microsoft Research,

VMware Research, ConsenSys R&D, CMU, U. Michigan, and MIT) and is currently in industrial use (for example, at Amazon Web Services).

Online Information

Some additional information about this book is available online at

`www.program-proofs.com`

Acknowledgments

I have many to thank for helping make this book possible.

I extend my deep gratitude to Rajeev Joshi, Rosemary Monahan, Bryan Parno, Cesare Tinelli, and especially Graeme Smith, who used earlier drafts of this book in teaching their university courses. The book has greatly benefited from their feedback, and from feedback of their students.

The detailed comments from Rajeev Joshi, Yannick Moy, Jean-Christophe Filliâtre, Peter Müller, and Ran Ettinger were much beyond the call of duty and were really helpful! I've also received good feedback from Nada Amin, Nathan Chong, David Cok, Josh Cowper, Mikaël Mayer, Gaurav Parthasarathy, and Robin Salkeld.

I'm grateful for the encouragement of Byron Cook and Reto Kramer in the Automated Reasoning Group where I work at Amazon Web Services.

The term "program proofs" as a rubric for the kind of science and engineering that this book is about was suggested by Nik Swamy.

To write and typeset this book, I used the Madoko system, and I thank Daan Leijen for creating Madoko and for helping me with customizations.

A big shout-out to Kaleb, who drew the cheerful chapter illustrations.

Lastly, thank you, Gwen, for your loving support and the countless weekends we spent at coffee shops while I was writing.

Thank you all!

K.R.M.L.

Notes for Teachers

Much thought goes into the selection and order of material in a book. Here, I describe the purpose of and motivation for chapters in greater detail. If you're a learner and just want to get started with the book, skip ahead to Chapter 0. If you're a teacher and want to plan a course outline, this is for you.

Part 0

If you want to go beyond that fun put-your-feet-into-the-water experience, I strongly recommend learning how to swim by paying attention to Chapters 3, 4, and 5 from Part 0.

Dafny provides a considerable amount of automation. This allows you to write the loop and array programs in Chapter 13 mostly by just supplying the necessary pre- and postconditions and loop invariants, without the need to pain yourself with the details of the proofs. In practice, when you leave those simpler programs, you will always encounter situations where a tool's automation runs out. A program-proof practitioner needs to know how to deal with those situations, and Part 0 is aimed at providing the necessary foundations.

Chapter 0

Chapter 0 sets the stage for the book and gives instructions for how to install the Dafny integrated development environment (IDE).

Chapters 1 and 2

Chapter 1 introduces some programming concepts like methods and functions, as well as fundamental specification concepts like pre- and postconditions.

If you want to start with the formal program-semantics underpinnings, then Chapter 2 (after reviewing Appendix B) is your friend. If you're a pro at understanding and dealing with formal equations, then your dream version of Chapter 2 would be a single page with just definitions. Most new learners are not equipped to be illuminated by equations alone, so I present the material in Chapter 2 more gradually and using aids like flow diagrams. The diagrams echo Floyd's seminal work, and I also incorporate

Hoare's way of explaining program behavior using triples, as well as Dijkstra's way of computing the first or last component of such triples given the other two.

If you're less interested in formal equations, then getting just a taste of Chapter 2 is enough for the rest of the book, with one exception. In Part 2, especially in Chapters 13 and 14, I often calculate necessary correctness conditions by applying weakest preconditions to a loop invariant and a loop-index update. I refer to this as "working backward". Those details are built up in Chapter 2 and I think that forms an important part of understanding the correctness of imperative programs.

Because I want readers to be able to "think as a programmer" as much as possible, I have preceded Chapter 2 with a lighter, more informal view of what it means to reason about what is known at various program points. A learner who feels comfortable writing programs but less comfortable with math formulas may find Chapter 1 to be a good preparation for Chapter 2. That's what I recommend, but this book has also been used by skipping Chapter 1 and going straight to Chapter 2, or by starting with the more formal Chapter 2 and then using Chapter 1 as a first guide to some Dafny notation.

Chapters 3

Discussing termination, Chapter 3 centers around the concept of a well-founded order and how that applies to recursive calls (termination for loops is covered in Chapter 11, which introduces reasoning about loops). It may seem odd to place a chapter on termination so early in a book. Indeed, many programs can be written in Dafny without any distraction from concerns about termination. This is because Dafny handles a large proportion of termination concerns completely automatically. What I have found, however, is that the day when a user first encounters a program that requires manual intervention in a termination proof, the surprise and additional learning necessary to understand what to do next require a large detour. Therefore, I have found termination to be an easier topic to cover early, before other concerns have become complicated. Besides, doing some termination proofs provides a learner with good opportunities to practice human-and-verifier interactions.

There is one more important reason to cover termination early. Without understanding termination, it is difficult and highly mysterious to explain mathematical induction. Chapter 5, all of Part 1, and Chapter 12 rely heavily on inductive proofs, so coming into those chapters with a good understanding of termination is helpful.

I do want to point out that I think of induction in a different way than many math texts. Many treatments of induction are very strict about the format of base case/induction hypothesis/induction step. These are often known as *induction schemas*. The strict format gives better "side bumpers" for high school introductions to induction, and induction schemas play an important role in logic or type theory where one wants to give formal *justifications* of why induction works. But the way I present it, mathematical induction is just about calling lemmas recursively. When recursion is taught to programmers, you don't talk about some strict syntactic format for how recursive calls must be done, or some pre-declared "recursion schema" that precedes the body

of a recursive method. No, programmers are used to making a recursive (or mutually recursive) call whenever they have a need to obtain the method's behavior again on a smaller problem size. Now, a key ingredient to the *correctness* of such recursive calls is their termination. That is how I teach induction in this book—feel free to call any lemma recursively, but when you do, make sure the recursive call terminates. In other words, termination is not built into some kind of recursion schema, but is instead a good-hygiene thing that you prove of any call.

For the most part, inductive proofs in this book do follow simple induction schemas using tried-and-true idiomatic syntactic formats. So, don't get all worried if that's the only way you've used induction before. (But see Section 5.6 for an example that highlights the difference.)

Anyhow, that's why termination is covered already in Chapter 3.

Chapter 4

Chapter 4 introduces algebraic datatypes. It is a simple chapter and its material will be familiar to those with a functional-programming background. This chapter is placed here because algebraic datatypes are great for teaching proofs, which is the subject of the subsequent chapters. Datatypes are used heavily throughout Part 1, and they are also used in some sections of Part 2.

Chapter 5

Being able to write manual proofs is essential to any nontrivial program development. The role of Chapter 5 is to teach how this is done. The focus is on the formulation of theorems and proofs, so I have chosen the subject of the theorems to be as familiar as possible—arithmetic and the algebraic datatypes introduced in the preceding chapter. I recommend Chapter 4 before Chapter 5, but it is possible to skip Chapter 4 if you also skip Sections 5.7 and 5.8.

If you want more proof practice even after Chapter 5, then I recommend Chapters 6 and 7, in either order.

In mathematics, there are two roles of a proof. One role is to communicate a proof to other mathematicians. The other is as a thinking tool during the development of a theorem. For programs, proofs have the same two roles. When a proof is machine checked, the communication I mentioned is between the user and the automated verifier. It is therefore crucial that proofs be practiced interactively with the verifier, just as reaching fluency in a foreign language cannot be gained just by reading books—you need to practice using it as a way of communication.

So, don't be satisfied by just *reading* Chapter 5. Redo the steps of the proofs yourself so you get to experience the interaction with the verifier firsthand. Also, do exercises, where the "answer" to what to do next is not right in front of you on the page. As eager as I'm sure the learner will be at this point to dig into programs, learning how to write proofs and communicate with the automated verifier will be well worth the time so invested.

Part 1

Part 1 teaches specifications and proofs in the setting of functional programs. With regard to program proofs, this setting has two main advantages. One is that data structures are immutable, so there is no need to keep track of changes to the program state. The other advantage is that data and operations tend to be defined recursively, which gives a consistent and natural way to structure proofs. Even if most of the programs you write are imperative, you will use functional program fragments in specifications. And if the programming language offers both imperative and functional constructs (like Dafny does), you will find many good uses of functional features in those parts of your imperative programs that are in fact immutable.

Chapters 6, 7, and 8

Chapter 6 introduces the basic ways to specify and reason about inductively defined data structures, and in particular lists. Chapter 7 follows that up with an inductive representation of unary numbers. Although not so interesting by themselves, unary numbers give ample opportunity to practice proof skills. Chapter 8 specifies and verifies two algorithms, both for sorting.

Chapters 9 and 10

Chapters 9 and 10 look at the structure of larger programs, paying attention to abstraction and information hiding. These concepts are at the core of good computer science and also play a crucial role for program proofs. A module that provides a high level of abstraction is often easier to both use and verify than a module that reveals too many implementation details to its clients.

Modularity, abstraction, and information hiding apply equally well to imperative programs, but the book first covers these topics in Part 1. Still, Part 2 can be read without first reading Part 1.

Chapter 9 introduces some mechanics of structuring code into modules. It touches on many small design decisions about what to reveal outside the module and what to hide inside the module.

Chapter 10 introduces another cornerstone of computer science: invariants. Here, the invariants talk about the properties of immutable data structures, and in Part 2 the invariants talk about the state before loop iterations (Chapter 11) and the steady state of mutable data structures (Chapter 16).

Chapters 10 and 16 both capture the data-structure invariants in a predicate by convention named `Valid()`. This gives a more uniform treatment of the functional and imperative settings, reduces the number of concepts needed to understand invariants, and always makes it clear *what* properties hold and *where* they hold. The downside of the explicit `Valid()` predicates is that they can make specifications verbose. Various languages (including Dafny) provide *predicate subtypes* (aka *subset types*, *refinement types*, or *dependent types*) that in effect incorporate the `Valid()` predicate into a type.

With two more chapters in Part 1, I would have covered them, too, but in choosing between them, I decided on keeping types and other invariants separate.

Part 2

Part 2 introduces ways to reason about imperative programs. While Part 1 is not a prerequisite of Part 2, imperative programs do use functions and modules to organize code and write modular specifications. Part 2 speaks about these as needed, but refers to Part 1 for a fuller treatment. As I've mentioned before, if you want to learn to do proofs well for imperative programs (beyond a quick tour of Chapter 11), I strongly recommend first learning the concepts of termination and proofs from Part 0.

Chapter 11

One of the most conspicuous programming constructs in imperative programming is the loop. Reasoning about loops using loop invariants is the subject of Chapter 11. Most beginners struggle with loop invariants. From a teaching perspective, I have two recommendations about loops, both of which are reflected in the book.

The first recommendation is to treat loop invariants as specifications for loops, rather than as an afterthought that seeks to explain what the loop body does. One way to do this is to hide the loop body from view. In a live demonstration, you can get this effect by collapsing the loop body using the IDE's outlining features. In Dafny, you can also do this by omitting the body of a loop altogether! For example, the Dafny verifier will accept and prove correct the program

```
method BodylessLoop() {
  var s, n := 0, 0;
  while n < 100
    invariant 0 <= n <= 100 && s == 4 * n
  assert s + n == 500;
}
```

despite the fact that the body of the loop is omitted. The point that needs to come across is that one reasons about the use of the loop from the invariant alone, *without* peering into its body, and one reasons about the correctness of the body without considering where the loop is used. That is, the loop invariant is like a contract between the context that uses the loop and the implementation of the loop. This is the same idea as reasoning about methods in terms of their specifications, not their implementations (as explained in Chapter 1). Yet, I have found that this idea is harder to get across for loops than for methods, which I suspect is because the loop body is "right there" and it's impossible for beginners to resist the temptation to look at the loop body.

So, when learning about loop invariants, my recommendation is to try, as much as possible, to separate the loop specification from its body. (Others sources that stress this include Hehner [61] and Morgan [93].)

My other recommendation when teaching about loops is to avoid **`for`** loops. Once you're comfortable with loop invariants, **`for`** loops are convenient and concise. But before you understand invariants, the fact that the update of the loop index (e.g., `i := i + 1;`) is implicit makes it much harder to understand what value of the loop index the invariant is supposed to hold for. Also, the helpful step of "working backward" from the loop invariant becomes hard to explain if you cannot see the loop-index update at the end of the loop body.

So, my recommendation is to stick with the more verbose **`while`** loops when teaching (at least until after Chapter 13).

Chapter 12

After the introduction of loops and loop invariants in Chapter 11, Chapter 12 introduces a practically important topic: going from recursively defined specifications to iteratively defined implementations. I think this topic was omitted from many of the program-verification books that expected proofs to be done by hand. When you do this work with paper and pen, you are free to invent convenient notations where you "know" what they say without having to be entirely formal about them. For example, you may notate the number of integers from a to b that satisfy a predicate P by

$$(\#i :: \ a \le i < b \ \wedge \ P(i))$$

This notation is beautifully agnostic about which "end" you remove an element from. For example, if we write $\|P(a)\|$ to denote 1 if $P(a)$ holds and 0 otherwise, then, for $a < b$, the properties

$$(\#i :: \ a \le i < b \ \wedge \ P(i)) \ \ = \ \ \|P(a)\| + (\#i :: \ a + 1 \le i < b \ \wedge \ P(i))$$

and

$$(\#i :: \ a \le i < b \ \wedge \ P(i)) \ \ = \ \ (\#i :: \ a \le i < b - 1 \ \wedge \ P(i)) + \|P(b-1)\|$$

are equally obvious. But if the # comprehension is defined recursively (inductively), which is most likely in a computer-aided verification system (unless # is built in, see e.g. [81]), then you have to choose at which end of the range $a \le i < b$ the recursion (induction) happens. This has an effect on what you have to do when verifying a loop for computing these things, since a loop also has to choose which direction (up or down) to evolve the loop index. I have seen many people (not just beginners) get stuck on this point, which is why I have devoted Chapter 12 to this topic, before doing more interesting algorithms in Chapter 13 and beyond.

Chapter 13

The 1980s brought not just a lot of good music but also several excellent books on program proofs. The classic gems by Backhouse [12], Cohen [29], Dijkstra and Feijen [41],

Gries [55], Hehner [61], Kaldewaij [68], Morgan [93], Reynolds [113], and Van de Snepscheut [122] often covered some Boolean algebra, then some formal program semantics (typically Hoare triples or weakest preconditions), and finally some example applications of what you might call "loop and array programs". These are wonderful textbooks. It's too bad the 1980s didn't bring the CPU speeds and tools needed to carry out these program proofs on a computer.

If you're familiar with these classic books, you'll feel right at home with Chapter 13, which covers several fun and instructive algorithms—many of which have been directly inspired by examples in the classic textbooks. If you carefully pick a subset of the sections in Chapter 13, you could get away with skipping Chapter 12. However, for someone wanting to learn program proofs well, I think the considerations in Chapter 12 are at least as instructive as proving the algorithms in Chapter 13.

A fine introductory course on program proofs would be to cover or review the Boolean algebra in Appendix B, then cover (some of) the program semantics in Chapter 2, followed by the loop and array programs in Chapters 11, 12, and 13. Although this wouldn't give the same depth and practical fluency as starting with all of Part 0, it gives the learner a quicker path to proving properties of small, interesting programs.

Chapters 14 and 15

Chapter 15 continues with more algorithms on arrays, but since those algorithms (unlike the ones in Chapter 13) *modify* the arrays, I first introduce the topic of state modifications in Chapter 14. This topic is part of what more generally is called *framing*. The Chapter 14 introduction of framing is gentle enough that it justifies being a prerequisite for Chapter 15.

I imagine that many university classes that want to cover all the basics of program proofs for imperative programs will find Chapter 15 to be a good chapter to end with.

Chapters 16 and 17

The final two chapters of the book introduce more difficult mutations of dynamically allocated data structures. Some programmers who come from C- or Java-like languages may feel an urgent need to conquer these chapters. That's all good, but I feel compelled to point out that many programming tasks can be performed equally well using immutable data structures like those in Part 1 of the book. I might have said "compelled to issue the reminder that" in the previous sentence instead of "compelled to point out that", but in my experience, this point is not always apparent to those whose primary programming language does not offer support for such types. If your problem can be solved with datatypes rather than with classes, then your proof effort will be both smaller and more pleasant (note, for example, that the class in Exercise 16.4 uses a datatype `List` rather than an imperative linked list stitched together via pointers).

Chapters 16 and 17 explain how to specify and verify mutable, heap-allocated data structures that may evolve over time. The main difference that makes this more difficult than specifying the immutable data structures of Chapters 9 and 10 is framing.

Essentially, framing comes down to keeping track of (or, in some cases, separating) all the objects that are used as part of a mutable data structure. This was introduced for simple cases in Chapter 14, so the main ingredient that is added in Chapters 16 and 17 is abstraction, that is, the ability to specify a frame to be a particular set of objects without having to divulge the exact identities of those objects to clients.

It's interesting that the only facilities needed in a language to handle framing and abstraction are **`modifies`**/**`reads`** clauses, sets, and ghost variables. Hence, these two last chapters build nicely on previous material in the book. Some languages and verifiers instead provide ways to restrict the use of object references. This can streamline some patterns of specifications, but requires explaining the restrictions imposed (like systems of object ownership à la Spec# [13] or Rust [115]) or the more complicated underlying logics (like separation logic [64] or implicit dynamic frames [98]). The issues and concerns are the same, so what is learnt from using the specifications in this book carries over to other formalisms.

On some material omitted

Some kinds of programs are trickier to get past a mechanical verifier than others. I've tried to avoid such complications when possible. For example, the book uses only a limited amount of multiplication, because multiplication of several variables is an area where automation is typically weaker. If you design your own exercises, then I suggest not going rampant with the use of multiplication (and make sure you first try the exercises yourself).

Other expressions that require some finesse are quantifiers and comprehensions. There are plenty of quantifiers in the book, but I've tried to stay clear of nested or otherwise complicated quantifiers. (And the only program that uses quantifiers before Chapter 13 is the `IsMin` predicate in Chapter 10.) Quantifiers are important, but it's both unfair and unnecessary to subject students to the ways of taming quantifiers until after they acquire a good understanding of program proofs and a fluency in interacting with the verifier.

One tricky area that I was not able to avoid is that of *extensionality* for collection types (in particular, multisets and sequences). In Sections 10.2.4 and 13.4, I include some detours to explain what is needed.

Chapter 0

Introduction

For many of us, programming started with some simple scripts that print messages on a terminal, display rectangles of various colors on the screen, or maybe even send chat messages between friends on mobile devices. The process of writing or modifying such a program is to add a couple of lines of code and then run the program to see what effect your change had. When we finally see the program print "Hello, Rustan!", show rectangles that are aqua *and* magenta, or automatically insert emojis when chat messages contain certain keywords, then we feel a sense of accomplishment that our program behaves like we want. The program *works*!

Not all programs are like that. Typically, we can't just run a program once and determine that it always works—that the program is *correct*. In fact, we may run the program many times and get the feeling it is correct, but then someone (maybe a customer of our service-oriented software-powered business) finds a way to use the program that

causes it to behave in a way we did not intend. The program crashes. The program corrupts customer data. Worse, the program enables unauthorized access to personal information. We'd like to know our programs do not suffer from such problems. But how can we tell if a program is correct? And what does it even mean for such a program to be correct?

Reasoning about the behavior of programs is the subject of this book. The subject is best approached after you have done at least a semester of programming, maybe even two. This means you're accustomed to basic programming constructs. You've written some programs that you've struggled with. You've tried your own hand at determining if your programs are correct, how to test and debug them, and how to try—repeatedly—to correct them.

This book teaches you how to think about programs and how to *prove* them correct, that is, how to construct precise arguments that show the programs behave as intended. Such rigorous arguments also help sharpen your thinking, so that you more readily understand how to write programs that *are* correct—after all, the only programs that you can *prove* to be correct are those that *are* correct. The process of proving a program's correctness often discovers bugs (errors, defects, omissions, typos, think-o's, call them what you may), and fixing the bugs is part of the road to correctness.

Techniques that reason precisely about programs fall under the rubric of *formal methods*. This name implies the use of precise mathematics, logic, and formal proofs. Indeed, throughout the book, I will make use of rigorous proofs and detailed logical justifications—in fact, so detailed that the justifications can be mechanically checked by a computer. The basic concepts that go into this formal process (for instance, the concept of a *precondition* or *invariant*) are useful also when describing a program's behavior informally. The formal, machine-assisted process helps you make sure you don't forget or miss any details, and it can be reapplied automatically when a program undergoes changes.

0.0. Prerequisites

As part of teaching you how to reason about programs in this book, I will teach you some things about specifications, abstraction, lemmas, and (a lot of) proofs. I do not assume you have any prior knowledge of these topics. If you did some proofs by induction in high school and remember what that was all about, that may be a plus, but it's not necessary. (In fact, if you have a lot of familiarity with topics like induction, you may find that I teach them and think about them in a way that differs from some common accounts. I attribute this to the fact that I'm a programmer, not a logician.)

What I do assume is that you are familiar with writing programs. Specifically, I assume you have a working knowledge of variables, bindings and mutable assignments, **`if`** statements, loops, and recursion. I assume you understand the idea that a program execution is a trace through the program's statements (the control flow) and that the state of a program consists of the values of the program's variables (the program data).

The part of logic that one unavoidably learns when writing programs is booleans (**`false`** and **`true`**) and the common operations on these values. I assume you've encountered expressions like "`A` and `B`", "`A` or `B`", and "not `A`". In logic, these are often notated, respectively, as

$$A \wedge B \qquad A \vee B \qquad \neg A$$

whereas in programming notation (and in this book), they are more often written as

```
A && B        A || B        !A
```

Logical "and" is also called *conjunction*, so we refer to the operands of `&&` as *conjuncts*. Similarly, logical "or" is called *disjunction*, so we refer to the operands of `||` as *disjuncts*. The negation operator binds stronger than "and" and "or", so `!A && B` is the same as `(!A) && B`, and you have to leave the parentheses in `!(A && B)` if you intend to express that at most one of `A` and `B` holds.

In specifications, the logic expression "`A` implies `B`" is often used. Notated as $A \Longrightarrow B$ or `A ==> B`, this expression says that "if `A` is **`true`**, then so is `B`". It can be written in terms of the other operators as `!A || B`, but the arrow many times improves the intuitive understanding of how we use implication in specifications. For example, suppose we want to write down "if I'm at a coffee shop, then I drink espresso" and "if I'm at the gym, then I drink water". These are nicely formulated as

```
(AtCoffeeShop ==> DrinkEspresso) && (AtGym ==> DrinkWater)
```

The operator `==>` binds weaker than `&&` and `||`, as is suggested by the fact that `==>` is 3 characters wide, whereas the others are only 2 characters wide.

Exercise 0.0.
Write this drink implication using the form `!A || B` instead of the form `A ==> B`. Be sure to use parentheses in the right places. Which formulation do you find easier to understand?

The arrow notation for implication also suggests an ordering between `A` and `B`. If `A ==> B` is a condition that always holds, then we say that `A` is *stronger* than `B` and that `B` is *weaker* than `A`. For example, `AtCoffeeShop` is stronger than `DrinkEspresso`, and `DrinkWater` is weaker than `AtGym`. If we're talking about some condition `A` and we add a conjunct `B` to it, then we say that we *strengthen* `A` with `B`, because `A && B` is stronger than `A` (that is, `A && B ==> A` always holds). Similarly, if we add a disjunct `B` to a condition `A`, then we say that we *weaken* `A`, because `A || B` is weaker than `A` (that is, `A ==> A || B` always holds). The strongest of all conditions is **`false`**, and the weakest of all conditions is **`true`**.

Here and throughout, when I use the comparison words "stronger", "weaker", "below", or "above", then I'm also allowing the possibility that the things being compared are equal. If it becomes necessary to exclude equality, I will say *strictly* stronger, weaker, below, or above.

As for program notation, it's good if you are able to at least read the syntax of a C- or Java-like program, meaning a program where variables and procedure signatures

contain types, where curly braces surround blocks of code, and where operators like == (equality) and && (conjunction, "and") feel familiar. In short, if you feel comfortable in your understanding of a program like

```
int sum(int[] a, int n) {
  int s = 0;
  for (int i = 0; i < n; i++) {
    s += a[i];
  }
  return s;
}
```

which computes the sum of the integer elements of an array `a`, then you are ready for this book. It's also fine if your background in programming comes from a functional language like OCaml or Haskell.

0.1. Outline of Topics

I start with some simple programs where we can talk about what it means for a property to *hold* at a particular program point (Chapter 1). The chapter uses parameterized procedures, introduces pre- and postconditions, and describes how to reason about variables and control flow. In Chapter 2, I present a formal treatment of the same material. This is the foundation for the entire book and is essential to understand. In the rest of the book, I will rely on the understanding of program reasoning that you get from Chapters 1 and 2, but I will then do the reasoning directly on the programs rather than continuing to show the steps in the underlying semantics.

Next, my goal is to prepare you for writing proofs. An issue that comes up throughout is that of *termination*. When the topic of termination comes up for induction, recursion, or loops, it gets entangled or easily confused with other concerns. Therefore, in Chapter 3, I cover the topic of termination by itself.

If I'm going to show you how to write proofs and teach you about induction, we need to have something to write proofs about. Some data structures from functional languages are perfect here, because they are defined recursively and they are mathematical in nature. Therefore, I cover inductive datatypes in Chapter 4. If you've done functional programming before, that chapter will be a breeze.

In Chapter 5, I introduce *lemmas*, which are claims that require proof. I show how to write calculational proofs and how to use induction.

Those initial chapters are gathered into Part 0 and cover the foundations for the programming in the rest of the book. Part 1 then considers functional programs and Part 2 considers imperative programs. Parts 1 and 2 are mostly independent of each other, so you can go directly from Part 0 to Part 2 if you're mostly interested in imperative programs.

In Part 1, Chapters 6, 7, and 8 treat standard ideas in functional programming: lists, unary numbers, and some sorting routines. Chapters 9 and 10 cover the concepts of

modules, abstraction functions, and data-structure invariants, which are considered for larger examples. Although I introduce them in Part 1, abstraction and data-structure invariants are equally important for imperative programs.

Part 2 is about imperative programs with mutable state, that is, variables whose values you can change. A programming construct that's specific to imperative programs is the loop, which I discuss in Chapter 11. This introduces *loop invariants*, another instance of the general concept of invariants, which are a cornerstone in reasoning about all programs. Chapter 12 blends recursive function definitions and iterative loops. Loops, arrays, and quantifiers are like three peas in a pod, and Chapter 13 discusses these with numerous examples. To get into programs that modify arrays and objects in the heap (that is, in the dynamic-storage area of a program), Chapter 14 gently introduces the concept of a *frame*, which lets a specification focus on a part of the heap. It is followed by Chapter 15, which presents many more examples that modify the contents of arrays. I continue the treatment of frames in Chapter 16, which focuses on classes and objects, class invariants, abstraction (again, like in Chapters 9 and 10), and *dynamic frames*, which are used to specify modifications among objects. Chapter 17 then uses these techniques in four more programs that use dynamic, mutable object structures.

0.2. Dafny

Throughout this book, I use the programming language Dafny to illustrate the learning points [78, 76]. Dafny is a great fit for this, because it was designed for reasoning, and it includes constructs for both imperative and functional programming, as well as for writing specifications, defining pieces of mathematics, stating lemmas, and writing proofs.

Importantly, Dafny has an associated program verifier that checks program correctness and all proof steps. The verifier offers a high degree of automation, so you don't need to provide many of the smaller proof steps. In fact, when I first teach about inductive proofs in Chapter 5, we will have to disable some of Dafny's automation, or else it will do most of the proofs for us and you won't learn anything.

Dafny has been used for over a decade at several dozens of universities worldwide. It has also been used in some systems-building and verification projects, as well as in industrial applications.

The curly-brace block-structure syntax of Dafny is Java-like, and so are its imperative loops and classes [54]. Dafny's functional constructs include functions and ML-like inductive datatypes [90] (and also Haskell-like coinductive datatypes, but I won't cover those in this book [110]). The Eiffel-like specification constructs ("contracts" [89]) are checked by Dafny's program verifier. A sketch of Dafny's syntax and a rundown on its constructs are given in Appendix A.

Dafny is an open-source project (github.com/dafny-lang/dafny) and is available on Windows, Mac, and Linux platforms. It has several integrated development environments (IDEs), including VS Code from

`code.visualstudio.com`

To get started, install VS Code onto your machine. Then, click the Extensions button, search for the Dafny extension from `dafny-lang`, and click to install it. This book was written to target Dafny version 4 (note that the VS Code extension uses a different versioning scheme, but it mentions the underlying version of Dafny). Dafny program files have extension `.dfy`, so after you select File->New Text File from the menu, save the file with a filename that ends with `.dfy` (for example, `MyProgram.dfy`).

The verifier will run automatically as you type in your program, so there are no additional commands you need to know.

Well, programs are only *verified* as you go. If you want to *run* your program, you have to first right-click the buffer and select *Compile* or *Compile and Run* (the entry point into your program is a method called `Main()`). As you go through this book, I will not be surprised if you often forget that programs can be run—most of your time will be spent specifying, writing, and proving the programs, and when the verifier finally has no more complaints, you know the program satisfies the specification (so why even run it, right? 😉).

If you want to write your own build scripts for a project, or if you for any other reason want to run the `dafny` tool from the command line, see the installation instructions at github.com/dafny-lang/dafny.

0.3. Other Languages

Though I use Dafny to teach the concepts of program proofs, you can apply these concepts in other languages. For example, Jean-Christophe Filliâtre has written WhyML versions of many programs in this book. They are available at

`github.com/backtracking/program-proofs-with-why3`

and can be verified using the Why3 tool. Similarly, Peter Müller has collected Viper, Prusti, and Gobra versions of many of the programs in this book, available from

`www.pm.inf.ethz.ch/program-proofs`

You will also find that tools like SPARK and F* provide similar verification experiences.

Notes

Being formal helps when you're trying to be precise. But if you understand how to reason about programs, you can be precise without always being fully formal. Carroll Morgan teaches a course he calls *Informal Methods*, which stresses the point that rigorous thinking about programs is crucial to program reasoning, whether or not you ever write down any mathematical formulas [94].