

```
> { ghost var Elements: seq<T> ghost const N: nat ghost const
t ghost var s: set<int> predicate IsInitialized(i: int) requires
== b.Length == c.Length requires 0 <= n <= |Elements| reads
i] < n && c[b[i]] == i } ghost predicate Valid() reads this,
uctor (length: nat, initial: T) ensures Valid() && fresh(Repr)
&& forall i :: 0 <= i < N ==> Elements[i] == initial { N := len
th, _ => initial); default := initial; a, b, c := new T[length]
) then a[i] else default } method Update(i: int, x: T) requires
:= Elements[i := x]; } lemma ThereIsRoom(i: nat) requires Valid
ction Upto(k: nat): set<int> ensures forall i :: i in Upto(k)
{y}, U - {y}, x); } } class ExtensibleArray<T> { ghost var
&& fresh(Repr - o7) ensures Elements == old(Elements)[i
eases |Elements| { front := new T[256](_ =>
; } } d(a: array<int>, lo: nat, hi: nat) rec
} meth (a: array<int>, b: array<int>) returns (c: array
invariant c[..k] == a[..k] && Below(c[..k], b[j..]) invar
var multiset(a[..i]) == multiset(b[..j])
|b| |a| 0 { assert |multiset(a)| == |mult
y == x else { a0, a1 := Split(a[1..], x)
ule Imm = ListLibrary export provides Q
FQ(front: LL.List<A>) ghost function Elements
lc { Elem); == Elements(FQ(q.front, LL.Cons(x
s(q); } y(q: Queue) ensures IsEmpty(q) <==> c
- 1, x) x then 1 else 0 } lemma SplitCount<T
x) + Cou a, hi - lo { } predicate HasMajority<
od Search wil (==)>(a: seq<Candidate>, ghost hask
<= s.Length && (s[lo..hi] && hi - lo == n } ghost predi
Initializ(i) e n { var S := Upto(N); SetCardinaliti
st var Repr set<int> front: array?<T> var depot: Extens
1; calc { multiset == assert c == c0 + [x] + c1; } multiset
```



K. Rustan M. Leino

illustrated by Kaleb Leino

PROGRAM PROOFS

Program Proofs

Program Proofs

K. Rustan M. Leino

Illustrated by Kaleb Leino

The MIT Press
Cambridge, Massachusetts
London, England

© 2023 Massachusetts Institute of Technology

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

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 $\text{\TeX}$ Gyre Pagella, Bera Mono, and Noto Emoji by the author.

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

d_r0

Contents

[Preface](#)

[Notes for Teachers](#)

[0. Introduction](#)

[Part 0. Learning the Ropes](#)

[1. Basics](#)

[2. Making It Formal](#)

[3. Recursion and Termination](#)

[4. Inductive Datatypes](#)

[5. Lemmas and Proofs](#)

[Part 1. Functional Programs](#)

[6. Lists](#)

[7. Unary Numbers](#)

[8. Sorting](#)

[9. Abstraction](#)

[10. Data-Structure Invariants](#)

[Part 2. Imperative Programs](#)

[11. Loops](#)

[12. Recursive Specifications, Iterative Programs](#)

[13. Arrays and Searching](#)

[14. Modifying Arrays](#)

[15. In-situ Sorting](#)

[16. Objects](#)

[17. Dynamic Heap Data Structures](#)

[Reference Material](#)

[A. Dafny Syntax Cheat Sheet](#)

[B. Boolean Algebra](#)

[C. Answers to Select Exercises](#)

[References](#)

[Index](#)

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

$$2 \leq x \implies 10 \leq 4 * (x + 1)$$

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.

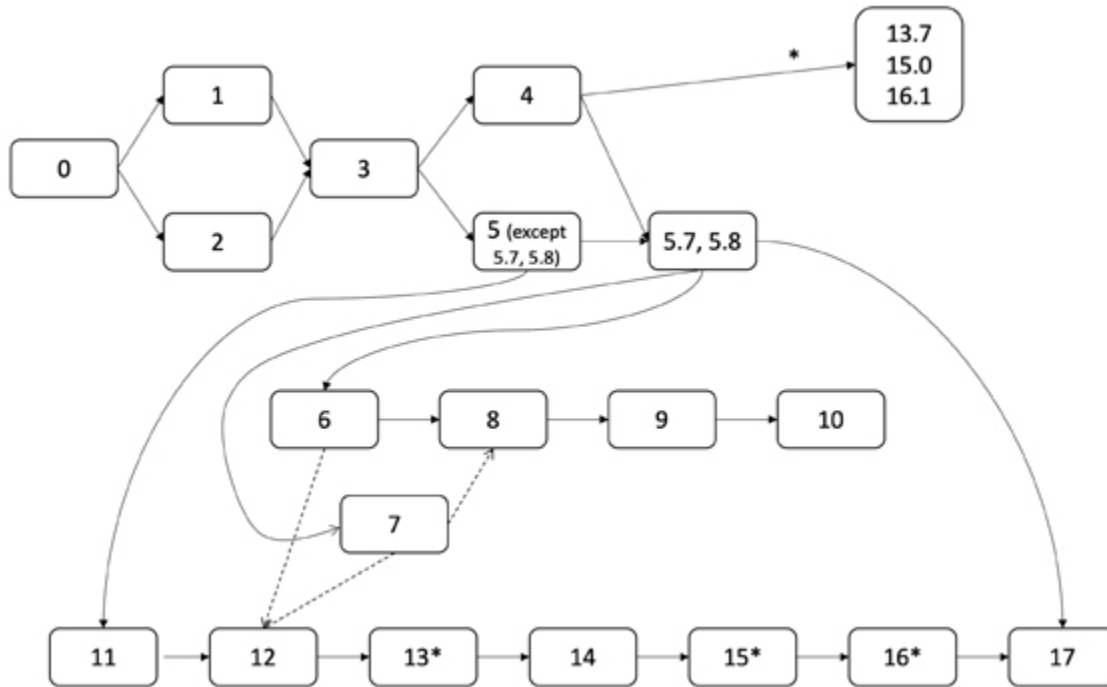
What the Book Is Not

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 $0 \leq x < y < 100$), implication ($\implies$), 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

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

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

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

and

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

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 $a \leq i < b$ 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

$$A \wedge B \quad A \vee B \quad \neg A$$

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

$$A \ \&\& \ B \quad A \ || \ B \quad !A$$

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 $A \Rightarrow B$ or $A \implies B$, 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 $\implies$ binds weaker than && and ||, as is suggested by the fact that $\implies$ 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 $A \implies B$. 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 $A \implies B$ 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 $A \ \&\& \ B \implies A$ always holds. Similarly, if we add a disjunct B to a condition A, then we say that we *weaken* A, because $A \ \|\| \ B \implies A$ 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].

Part 0

Learning the Ropes

Chapter 1

Basics



In this chapter, we review some basics of programming, consider what it means for a condition to always hold at a particular program point, and see how to write simple specifications.

1.0. Methods

A *method* is a program declaration that prescribes some behavior. For instance, here is the declaration of a method named `Triple`:

```
method Triple(x: int) returns (r: int) {  
    var y := 2 * x;  
    r := x + y;  
}
```

This method takes an *in-parameter* x of type integer and returns an *out-parameter* r , also of type integer. The *body* of a method, given in curly braces after the method signature, is a list of *statements* that give the method's implementation. Here, the body consists of two statements. The first statement declares a local variable y to which it assigns the value $2*x$. The second statement assigns the sum of the in-parameter x and the local variable y to out-parameter r .

Exercise 1.0.

If x is 10, what value does the method assign to r ?

In Dafny, methods can have any number of in-parameters and any number of out-parameters. In the body of the method, the out-parameters act as local variables and can be assigned and read. When the method body ends, whatever values the out-parameters have will be the values returned to the caller. The in-parameters can of course also be read, but they cannot be re-assigned in the method body.

Here is a statement that shows a call to the method:

```
var t := Triple(18); // sets t to 54
```

Next to the statement, I wrote a comment that the effect of this call is to set t to 54, which is 3 times 18. By the end of the next chapter, we will be able to turn comments like this into conditions that we can prove to be true.

1.1. Assert Statements

By inspecting method `Triple` above, you can see that it returns the value $3*x$. We can state this observation explicitly in the program using an **assert** statement (sometimes called an *inline assertion*, because it places an assertion at a particular point in the code):

```
method Triple(x: int) returns (r: int) {  
  var y := 2 * x;  
  r := x + y;  
  assert r == 3 * x;  
}
```

It is good software-engineering practice to use assertions (or in some other languages, comments) to write down essential and non-obvious conditions in the code. Writing down the condition helps our thinking and serves as documentation.

Throughout this book, these kinds of assertions are more than just documentation. They are *proof obligations*. For the program to be considered correct, we must *prove* that the asserted condition holds every time the program's control flow reaches the assertion. The proof should apply for all possible values of the enclosing method's parameters. This means that *running* the program is always going to be hopelessly insufficient as a form of proof—there are far too many inputs to try. Instead, we will construct a mathematical proof where we use symbolic *variables*, not specific values, to stand for the program's inputs.

When you write an assertion in a Dafny program, the verifier immediately tries to do the proof for you. For the assertion in `Triple`, the verifier easily constructs the proof, which confirms that the asserted condition always holds. In other cases, if the verifier cannot prove the assertion (for example, if we change the condition above to $3 * x + 1$), then it produces an error message.

Exercise 1.1.

Write the method `Triple` in the Dafny IDE. If you don't introduce any typos, you should see the indication “verified” at the bottom. Change the asserted condition to $3 * x + 1$. What happens?

1.2. Working with the Verifier

When receiving an error message from the program verifier, we need to figure out the problem (in other words, we need to debug the verification) and take some recourse. Both of these can be difficult.

If the problem isn't immediately obvious, we can try to figure out the problem by writing more assertions at various points in the program. For example, we might add another assertion between the two assignment statements above. Writing down an assertion amounts to asking the verifier, “do you know that this condition (always) holds at this program point?”.

In some cases, our analysis results in detecting an error in the code. In other cases, we realize that the condition we wrote down isn't correct, which sharpens our understanding of the code. In yet other cases, the condition may in fact always hold but there isn't enough information to conclude this. The typical recourse in such cases is to write some preconditions or to strengthen some invariant; we will see many examples of this in the next few chapters. Finally, in some cases, the problem may be that the verifier isn't “clever” enough to do the proof automatically, in which case we'll need to help it along. The verifier tends to do very well on the sorts of programs discussed in the next several chapters, but for more intricate programs, it will frequently be necessary to supply lemmas or proof steps.

Any error message received from the verifier should be understood as a *possible* error (in the code, in a specification, or in some auxiliary declaration required for the proof), rather than as evidence that the program will not work as intended. In general, think of the program verifier as a dedicated, detailed-oriented colleague who is constantly peering over your shoulder. Sometimes, it detects errors quickly. Other times, just like the best of our colleagues, the verifier isn't able to figure it out, and thus

we need to supply further information as hints that help explain why we think the program is correct.

A program trace that reaches a failing assertion (that is, an assertion whose condition evaluates to **false**) is erroneous and does not continue past the assertion. In other words, the only control flows that continue after an assertion are ones where the condition holds. For illustration, consider the program

```
method Triple(x: int) returns (r: int) {  
  var y := 2 * x;  
  r := x + y;  
  assert r == 10 * x; // error  
  assert r < 5;  
  assert false; // error  
}
```

The first assertion does not always hold, so the verifier issues an error message. For any trace where the first assertion does in fact hold (namely, when $3 \cdot x == 10 \cdot x$, which holds just when $x == 0$), the second assertion holds as well. Therefore, the verifier has no complaint about the second assertion. But even the traces that pass both of the first assertions do not pass the third, so the verifier will issue a complaint about it.

Exercise 1.2.

Try this program with the three **assert** statements for yourself in the Dafny IDE. Change the second assertion to make the verifier complain about the first two assertions but not about the third.

1.3. Control Paths

The simple code examples we've seen so far have a single control path containing a sequence of statements. With **if** statements and other conditional statements, a piece of

code can contain many control paths. A program is correct when the traces along *all* control paths are correct.

For example, here is method `Triple` again, this time using a conditional statement:[0](#)

```
method Triple(x: int) returns (r: int) {
  if x == 0 {
    r := 0;
  } else {
    var y := 2 * x;
    r := x + y;
  }
  assert r == 3 * x;
}
```

The `if` statement divides the program traces into those that flow through the “then” branch (when $x == 0$) and those that flow through the “else” branch (when $x \neq 0$). Program correctness requires the asserted condition to hold regardless of which branch is taken.

When we reason about the control path through the “then” branch, we can confine our attention to the case where x is 0, because only when x is 0 can the program take that path. Likewise, when we reason about the “else” branch, we can confine our attention to the case where x is non-0, because only when x is non-0 can the program take that path.

Control in the ordinary `if` statement is *deterministic*. This means that the program inputs uniquely determine the control path taken. Control flow can also be *nondeterministic*, which means that repeatedly running the program, even on the same input, may result in different traces.

An example of nondeterministic control flow is found in Dafny's `if-case` statement, where the guards are allowed to be overlapping. This is illustrated by the following, overly convoluted version of `Triple`:

```
method Triple(x: int) returns (r: int) {
  if {
```

```

    case x < 18 =>
      var a, b := 2 * x, 4 * x;
      r := (a + b) / 2;
    case 0 <= x =>
      var y := 2 * x;
      r := x + y;
  }
  assert r == 3 * x;
}

```

When reasoning about the first branch, we can assume $x < 18$, because that branch is taken only if x is less than 18. Similarly, when reasoning about the second branch, we can assume $0 \leq x$, because that branch is taken only if x is non-negative. If x is *both* less than 18 and non-negative, then either branch can be chosen; therefore, in order for the assertion to be correct, both control paths must be checked to be correct for such values of x . In this example, both branches compute the same value for r , but in general, different branches can have vastly different behaviors.

Two other points are worth making about this last example. One point is that the first branch makes use of a simultaneous assignment, where the right-hand sides are all evaluated before the variables on the left are assigned. The other point is that the scope of local variables declared in a **case** branch is limited to that branch, even though the branch is not enclosed in separate curly-brace pairs.

1.4. Method Contracts

A *client* of a method (or function or type or module) is a piece of code that wants to use the method (or function or type or module). Consider a client of the `Triple` method:

```

method Caller() {
  var t := Triple(18);
  assert t < 100;
}

```

By looking at the body of `Triple` in the previous sections, we can see that `t` will be assigned the value 54, and thus the caller's assertion will hold. However, by the good software-engineering principle of *information hiding*, the method body is considered to be private to the method and should not be looked at by callers. This allows the method to alter the details of its implementation without affecting callers.

Exercise 1.3.

Using the definition of `Triple` from Section 1.0 and the definition of `Caller` just given, how does the verifier respond?

So, if a caller is not allowed to peek into the method body, how can the caller know anything about what the method will do? The answer is that the method additionally is declared with a *specification*. The specification describes the behavior of the method, but abstracts from the details of the method body. More precisely, a method uses an “agreement”, or *contract*, between the caller and implementation that says what the caller can rely on. The contract thus gives the method author flexibility to describe the interface to the method without divulging the details of the current implementation. Since the method contract, not the method body, is used at call sites, we say that the methods are *opaque*.

A method contract has two fundamental parts: a *precondition* and a *postcondition*. The precondition says when it is legal for a caller to invoke the method. It is a proof obligation at every call site, and in exchange it can be assumed to hold at the start of the method body. The postcondition is a proof obligation at every return point from the method body, and in exchange it can be assumed to hold upon return from the invocation at the call site.

In our example, we can write a postcondition for `Triple` that lets `Caller` prove the assertion in its body. The

postcondition is introduced with the **ensures** keyword:

```
method Triple(x: int) returns (r: int)
  ensures r == 3 * x
{
  var y := 2 * x;
  r := x + y;
}
```

Method caller now verifies, and so does Triple.

Our example does not warrant a precondition, but let us nevertheless declare one for illustration purposes. A precondition is introduced with the **requires** keyword:

```
method Triple(x: int) returns (r: int)
  requires x % 2 == 0
  ensures r == 3 * x
{
  var y := x / 2;
  r := 6 * y;
}
```

This version of Triple requires the given x to be even. Indeed, the new implementation body relies on the evenness of x —without the precondition, the value assigned to r would be less than $3x$ for odd values of x (since integer division performs some rounding) and the verifier would issue an error message that Triple may fail to establish its postcondition.

Exercise 1.4.

Remove (or comment out) the precondition of Triple. What error does the verifier give you?

Exercise 1.5.

Write two stronger alternatives to the precondition $x \% 2 == 0$ that also make the method Triple verify.

Let's take a look at some more examples.

1.4.0. Underspecification

Consider the following method specification:

```
method Index(n: int) returns (i: int)  
  requires 1 <= n  
  ensures 0 <= i < n
```

The precondition says that `Index` can be called only on positive integers. The postcondition says that `Index(n)` will return `i` as some number between 0 and less than `n`.

Sidebar 1.0

In this book, I'll be careful to say “0 to `n`” to mean the possible values of `i` in the half-open interval $0 \leq i < n$, and I will say “0 *through* `n`” to mean the inclusive interval $0 \leq i \leq n$.

So, *which* number from 0 to `n` will be returned? The specification does not say. Therefore, when we reason about calls to `Index`, we consider all of these values. Viewed by the caller, it is as if this method were nondeterministic.

Here is one way to implement the method:

```
method Index(n: int) returns (i: int)  
  requires 1 <= n  
  ensures 0 <= i < n  
{  
  i := n / 2;  
}
```

This implementation is correct: for every possible input value `n` that satisfies the precondition, the method body establishes the postcondition.

Here is another way to implement `Index`:

```
method Index(n: int) returns (i: int)  
  requires 1 <= n  
  ensures 0 <= i < n  
{  
  i := 0;  
}
```

This implementation is also correct. Because methods are opaque—that is, callers only get to see the specification of a method, not its body—it is fine to change the body of `Index` from `i := n / 2;` to `i := 0;` without affecting the correctness of any callers.

Each of the two implementations we just considered is deterministic. That is, for a given input, the method body computes the output in the same way. But since methods are opaque, this is not something a caller can rely on. For example, in

```
var x := Index(50);  
var y := Index(50);  
assert x == y; // error
```

the assertion cannot be proved, because all we know from the specification of `Index` is

```
0 <= x < 50 && 0 <= y < 50
```

but we know of no connection between `x` and `y`.

While the specification allows any value, the implementation can be—and most often is—deterministic. This important idea is called *underspecification*. It precisely specifies the freedom entailed by the caller-implementation contract.

1.4.1. Multiple postconditions

Consider a method that computes the smaller of two given values:

```
method Min(x: int, y: int) returns (m: int)  
  ensures m <= x && m <= y
```

This specification says that the value returned is no greater than either `x` or `y`. But it allows the value returned to be significantly smaller than both.

Exercise 1.6.

Write an implementation for `Min` that satisfies the postcondition above but is not always the minimum of `x` and `y`.

To make the specification be the expected one for minimum, we also need to say that the value returned is one of the two inputs:

```
method Min(x: int, y: int) returns (m: int)  
  ensures m <= x && m <= y  
  ensures m == x || m == y
```

Exercise 1.7.

Here is the type signature of a method to compute `s` to be the sum of `x` and `y` and `m` to be the maximum of `x` and `y`:

```
method MaxSum(x: int, y: int) returns (s: int, m: int)
```

(a) Specify the intended postcondition of this method. (b) Write a method that calls `MaxSum` with the input arguments 1928 and 1. Follow the call with an assertion about what you expect the two out-parameters to be. If the verifier complains that the assertion may be violated, go back to (a) to improve the specification. This is a useful way to “test the specification”. Note, you're testing the specification by using the verifier, not by running the program. (c) Write an implementation for `MaxSum`.

Exercise 1.8.

Consider a method that attempts to reconstruct the arguments `x` and `y` from the return values of `MaxSum` in Exercise 1.7. In other words, consider a method with the following type signature and postcondition:

```
method ReconstructFromMaxSum(s: int, m: int) returns (x: int,  
y: int)  
  ensures s == x + y  
  ensures (m == x || m == y) && x <= m && y <= m
```

(a) Try to write the body for this method. You will find you cannot. Write an appropriate precondition for the

method that allows you to implement the method. (b) Write the following test harness to test the method's specification:

```
method TestMaxSum(x: int, y: int) {  
    var s, m := MaxSum(x, y);  
    var xx, yy := ReconstructFromMaxSum(s, m);  
    assert (xx == x && yy == y) || (xx == y && yy == x);  
}
```

How can you change the specification of `ReconstructFromMaxSum` to allow the assertion in the test harness to succeed?

1.5. Functions

There's one more central declaration construct we'll see often: functions. As you know from mathematics, a *function* denotes a value computed from given arguments. The key property of a function is that it is *deterministic*, that is, any two invocations of the function with the same arguments result in the same value.

Here is an example function declaration in Dafny:

```
function Average(a: int, b: int): int {  
    (a + b) / 2  
}
```

Note that, whereas a method is declared to have some number of out-parameters, a function instead declares a result type, and whereas a method body is a statement, the body of a function is an expression.

Functions can be used in expressions, so we can write a specification like this:

```
method Triple'(x: int) returns (r: int)  
    ensures Average(r, 3 * x) == 3 * x
```

Sidebar 1.1

Identifiers in Dafny can contain the ' character. Here, I declared a method named `Triple'`, which you can pronounce as “triple prime”.

Exercise 1.9.

The specification of `Triple'` is not the same as that of `Triple`. (a) Write a correct body for `Triple'` that does not meet the specification of `Triple`. (b) How can you strengthen the specification of `Triple'` with minimal changes to make it equivalent to the specification of `Triple`? (c) How can you use Dafny to prove that the specifications in (b) are indeed equivalent?

The example above highlights another important difference between methods and functions in Dafny: whereas methods are opaque, functions are *transparent*. This is necessary, because if callers had to understand a function only from its specification, then the specification of functions could never make use of functions, which would severely limit what can be said about a function.

Nevertheless, a function can have specifications. Of special importance is the function's precondition, which says under which circumstances the function is allowed to be invoked. For example, we can restrict uses of `Average` to non-negative arguments:

```
function Average(a: int, b: int): int
  requires 0 <= a && 0 <= b
{
  (a + b) / 2
}

method Triple(x: int) returns (r: int)
  ensures r == 3 * x
{
  if 0 <= x {
    r := Average(2 * x, 4 * x);
```

```

    } else {
      r := -Average(-2 * x, -4 * x);
    }
  }
}

```

Function preconditions are enforced at call sites, just as for method preconditions.

Since assertions, preconditions, and postconditions use boolean conditions, it frequently happens that we declare boolean functions (that is, a function with result type **bool**) for use in specifications. A boolean function is also known as a *predicate* and Dafny reserves a keyword for this purpose: the meaning of

```

predicate IsEven(x: int) {
  x % 2 == 0
}

```

is identical to

```

function IsEven(x: int): bool {
  x % 2 == 0
}

```

1.6. Compiled versus Ghost

I'll end our introduction of basics with one more important concept. When reasoning about a program, it is frequently necessary to make use of more information than is needed by the compiler or at run time. The Dafny language integrates many features for this purpose. A declaration, variable, statement, etc., that is used only for specification purposes is called a *ghost*. The verifier takes all ghosts into account, whereas the compiler erases all ghosts when it emits executable code. Program constructs that make it into the executable code are referred to as *compiled* or *non-ghost*.

We have seen several ghost constructs already. Pre- and postconditions (declared by **requires** and **ensures** clauses) are ghost. They are used to specify the behavior of the

program and establish a contract between callers and implementations. We have also seen a ghost statement: **assert**.

Pre- and postconditions and assertions are checked by the verifier when you write the program. Since they are ghost, they are erased by the compiler and thus they bear no run-time cost. Even if we were not worried about run-time cost, there would be no point in checking assertions at run-time, because once they pass the verifier, the assertions are known to hold in every trace of the program.

Several kinds of declarations in Dafny exist in both a ghost form and a compiled form. They look mostly the same and are treated the same by the verifier, but the purpose for using them is often different. To declare a variable, (in- or out-)parameter, method, function, or predicate as a ghost, simply precede its declaration with the keyword **ghost**. To make the erasure of ghosts possible, Dafny checks that compiled code does not rely on ghost constructs. For example, as illustrated here:

```
method IllegalAssignment() returns (y: int) {  
  ghost var x := 10;  
  y := 2 * x; // error: cannot assign to compiled  
             // variable using a ghost  
}
```

a program is not allowed to use a ghost variable in the right-hand side of an assignment to a compiled variable.

Exercise 1.10.

The function and method in the following code snippet are both declared as compiled.

```
function Average(a: int, b: int): int {  
  (a + b) / 2  
}  
  
method Triple(x: int) returns (r: int)  
  ensures r == 3 * x  
{
```

```

    r := Average(2 * x, 4 * x);
}

```

(a) Change this example to declare `Average` as `ghost`. What error message do you get? (b) Declare both `Average` and `Triple` as `ghost`. Is that legal? (c) Declare `Average` as `compiled` and `Triple` as `ghost`. Is that legal?

1.6.0. Ghost-method example

Let me illustrate a (silly) use of ghost constructs by considering another version of our running example:

```

method Triple(x: int) returns (r: int)
  ensures r == 3 * x
{
  var y := 2 * x;
  r := x + y;
  ghost var a, b := DoubleQuadruple(x);
  assert a <= r <= b || b <= r <= a;
}

```

```

ghost method DoubleQuadruple(x: int) returns (a: int, b: int)
  ensures a == 2 * x && b == 4 * x
{
  a := 2 * x;
  b := 2 * a;
}

```

This ghost method computes $2 \cdot x$ and $4 \cdot x$ into the method's two out-parameters. As you can see, other than the `ghost` keyword in front of the method declaration, `Double-Quadruple` looks like an ordinary, compiled method.

This version of `Triple` calls the ghost method, storing its out-parameter in two local ghost variables, `a` and `b`. The subsequent assertion mentions both ghost and compiled variables to express a condition that is expected to hold. When this code is compiled, the postcondition of `Triple`, `Triple`'s local variables `a` and `b`, the call to `DoubleQuadruple`, the assertion, and the entire `DoubleQuadruple` method are erased.

Sidebar 1.2

The example above (and also the `Index` example of Section 1.4.0) illustrates a notational feature that we will make frequent use of: the *chaining* of comparison operators. For example, the operators `<`, `<=`, and `=` can be chained together without repeating the shared operands. The meaning is that of conjoining the pairwise comparisons. For example, the expression

```
0 <= i < j < a.Length == N
```

has the same meaning as

```
0 <= i && i < j && j < a.Length && a.Length == N
```

but is easier to read. The **`assert`** statement in our last version of `Triple` above could also have been written

```
assert (a <= r && r <= b) || (b <= r && r <= a);
```

1.7. Summary

In this chapter, I have introduced methods, functions, pre- and postconditions, simple statements and control flow, and the distinction between ghost and compiled.

A method has any number of in- and out-parameters. It is specified with pre- and postconditions, and its body is a list of statements. The body is opaque, which says that callers reason about method invocations in terms of the specification alone. The behavior of a method can be nondeterministic.

A function has any number of in-parameters and exactly one result value. It can have a specification, and its body is an expression. The body is transparent, which says that callers can look into the function body when reasoning about function invocations. The behavior of a function is deterministic. A function with a boolean result type is called a predicate.

An **assert** statement points out a condition that a programmer expects to hold. It gives rise to a proof obligation that must be proved before you can compile and run your program. Thus, in a verified program, you can rely on every asserted condition to hold.

Pre- and postconditions describe the behavior of methods. The precondition limits the uses of a method by declaring a condition that must be proved at every call site. The postcondition limits the behaviors of the method implementation by declaring a condition that must be proved for every control path through the method body.

A ghost is a construct that serves to explain the intended behavior of a program. It is used by the verifier, but erased by the compiler. Ghost constructs include pre- and postconditions and **assert** statements.

Exercise 1.11.

(a) What does the verifier have to say about the two assertions in this program?

```
function F(): int {  
    29  
}  
  
method M() returns (r: int) {  
    r := 29;  
}  
  
method Caller() {  
    var a := F();  
    var b := M();  
    assert a == 29;
```



```
    assert b == 29;  
}
```

(b) Explain why. (c) How can you change the program to make both assertions verify?

Notes

Programming languages do not agree on what to call procedural routines. For example, C and Python refer to a body of code as a “function”, and Rust and Go use the word “method” for a “function” that takes a receiver parameter (like `this`). In this book, a “method” has code for its body, and a “function” has an expression for its body. Importantly, a “function” in this book (and, indeed, a **function** in Dafny) behaves like a mathematical function. Whether or not a method or function has a receiver parameter depends on whether or not it's declared as a member of a type (we'll have to wait until Chapter 16 before we see a receiver parameter).

A method with side effects is inappropriate for use in specifications. Since Java only has methods, the Java Modeling Language (JML) [31] allows a method to be marked as “pure”, which makes it usable in specifications, approximately like a mathematical function. The OpenJML tool also supports a “function” mark-up for a method, which declares it to be like a function [32].

In WhyML [20] and F* [53], there is no distinction between statements and expressions. Instead, these languages rely on other mechanisms (regions and effect types, respectively) to keep track of the state mutations that are possible in their program functions.

The idea of writing pre- and postconditions and proof assertions as part of a program text is old. For example, they are featured as comments in early program proofs (e.g., [51, 88]). The standard documentation style of the

influential language CLU [87] includes pre- and postconditions (and a form of the **modifies** clauses we'll see in Part 2). The Euclid language for verifiable programs supports pre- and postconditions as part of its syntax [74]. The role of specifications in the design of programs is emphasized by VDM [67] and Larch [57]. For additional historical notes, I refer you to a survey paper on behavioral specification languages [59].

It was Bertrand Meyer's Eiffel language and the accompanying *Design by Contract* methodology that popularized the idea of writing pre- and postconditions in the program text [89]. These contracts between callers and implementations are written using the ordinary language syntax for expressions, an idea that influenced the designs of JML [66] and Spec# [13].

The use of ghost variables and ghost code has been part of specification folklore for decades. They sometimes go by different names. For example, Owicki and Gries called them “auxiliary” variables [106]. In SPARK [43] and WhyML [20], ghost functions are known as “logical functions”. For a wider perspective on ghost declarations, see [48, 59].

Lastly, an important remark about integers. Dafny's type `int`, which I use throughout the book, is *unbounded*. This means that the numbers behave as in mathematics, without any maximum or minimum value. So, if your Dafny program computes a large number (perhaps using the Ackermann function in Section 3.3.1), adds up numbers (like in Section 12.3), or does something as innocent as tripling the value of a given parameter (like method `Triple` in this chapter), you don't have to worry that the magnitude of the result will be too large. (Well, if numbers become so large that your computer doesn't have enough memory to represent them 🙄, your program will halt. Dafny does not check that you have enough resources to run your programs.) Dafny

additionally supports *bounded integers*, like those you can represent using 32 bits, but I won't use them in this book. Some programming languages support only unbounded integers (e.g., Python), some support only bounded integers (e.g., C and Java), and some verification languages support a mix of the two (e.g., SPARK and JML).

⁰Note that, unlike some other popular languages, Dafny does not require the guard condition of `if` to be surrounded in parentheses. The program text looks cleaner without them.

Chapter 2

Making It Formal



This chapter gives the formal underpinnings of what the previous chapter described informally. The meaning of programs is given precisely with mathematical formulas. These are said to give the *semantics* of the program statements.

Throughout this book, the reasoning is based on what is called *Floyd logic*. It uses boolean formulas to describe what is known before and after each statement, which allows us to break down the reasoning about programs into

reasoning about individual statements of the program. Along the way, I will define *Hoare triples*, which give a good guide to understanding semantics, and *weakest preconditions*, which give convenient ways to automate program reasoning.

The study of program semantics may seem overly concerned with picky details. Think of it as long multiplication—it prescribes steps for computing something in detail. Once you've applied long multiplication on many problems, you develop a deeper understanding of multiplication. You then realize that some steps can be replaced by shortcuts. More importantly, you can then put multiplication to use in daily problems. It is not the tiny steps of long multiplication that let you solve problems, but it is your familiarity with multiplication that does. Because you can carry out the steps, you know of a way to check something in detail, but in practice you use a calculator or spreadsheet to perform multiplication for you.

Program proving is like that. Carrying out the small steps lets you see how program reasoning is done. If you're unsure of something, you can return to the foundational steps and convince yourself of your program's correctness (or find mistakes). In practice, however, you will typically use an automated program verifier that performs the semantic computations for you.

Let's get on with the details. Program semantics, here we come!

2.0. Program State

A program has several kinds of *variables*. These include the in- and out-parameters of methods and the local variables declared in a method body. The variables that can be used at a point in a program are said to be *in scope* at that

program point. The *state* at a program point is a valuation of the variables in scope at that program point.

For illustration, consider the following method:

```
method MyMethod(x: int) returns (y: int)
  requires 10 <= x
  ensures 25 <= y
  ①
  { ①
    var a, b;
    ②
    a := x + 3;
    ③
    if x < 20 {
      ④
      b := 32 - x;
      ⑤
    } else {
      ⑥
      b := 16;
      ⑦
    }
    ⑧
    y := a + b;
    ⑨
  } ⑩
```

On entry to this method, that is, at the program point labeled ①, only the method's in-parameter (x) is in scope. Just inside the method body, at program point ①, both in- and out-parameters (x and y) are in scope. At program point ②, after the statement that declares local variables a and b , the scope consists of x , y , a , and b . The same is true at program points ③ through ⑨. On exit from the method, at program point ⑩, the local variables have gone out of scope so the scope once again consists of just the parameters (x and y).

The state at program point ①, which is called the *initial state* of the method, is a valuation of x . For example, x may be 18 or 37. If it is 37, the state at program point ⑨ is 37 for

x , 56 for y , 40 for a , and 16 for b . Moreover, the state at ⑩, which is called the *final state* of the method, is then 37 for x and 56 for y . You can determine these states if you trace the program execution through the method body.

Exercise 2.0.

If the initial program state is 18 for x , what is the program state at ⑨ and what is the final program state?

As we saw in the previous chapter, callers of a method are responsible for establishing the method's precondition. Consequently, we know the initial state satisfies the precondition. For example, the initial state cannot be 2 for x , because $10 \leq x$ does not hold if x were 2.

As we also saw in the previous chapter, a method implementation is responsible for establishing the method's postcondition. Consequently, to declare `MyMethod` to be correctly implemented, we must check that $25 \leq y$ holds in state ⑩ for every possible trace through the method body, for every possible initial state.

We cannot reasonably determine this if we had to try out traces (like those starting from x being 18 or 37) one at a time. Instead, we need a way to reason about many states at a time. This is done using boolean expressions. More precisely, we use a boolean expression to denote those states that make the boolean expression evaluate to **true**. For example, the boolean expression $15 \leq x \leq 40$ represents all initial states where the value of x lies in the range 15 through 40.

Another name for a boolean expression is *logical formula* (or just *formula*) or *predicate*, which simply means an expression that for a given state evaluates to either **false** or **true**. You can also think of a predicate as *characterizing a set of states*.

Instead of tracing a state through the program, we are going to trace a *predicate* through the program. The main

purpose of this chapter is to teach you how that is done. Here are some examples of what we'd like to be able to figure out:

- If $12 \leq x \ \&\& \ a \% 2 == 0$ holds at ③, what can we say about the possible states at ⑨?
- If we want to be sure that the state satisfies $a > b$ whenever control reaches ⑧, what needs to hold at program point ②?
- Suppose we write down a predicate at each program point ① through ⑨, write the method precondition at ①, write the method postcondition at ⑩, and then check that each such predicate includes the states reachable from the prior predicates. Does this mean the method implemented correctly?

The first bullet is answered by what's called strongest postconditions, the second bullet by what's called weakest preconditions, and the third bullet is the idea behind Floyd logic. Weakest preconditions are also the basis for what in Part 2 of this book I will often refer to as “working backward”.

2.1. Floyd Logic

Consider this simple program:

```
method MyMethod(x: int) returns (y: int)
  requires 10 <= x
  ensures 25 <= y
①
{ ①
  var a := x + 3;
  ②
  var b := 12;
  ③
  y := a + b;
  ④
```


} ⑤

where I have labeled the the various program points. Let's write a predicate at each program point. In the initial state, the only thing we know is the method precondition, so let's write it at ①. In the final state, the only thing we care about is the method postcondition, so let's write it at ⑤.

① $10 \leq x$

⑤ $25 \leq y$

So, ① shows what we're given, and ⑤ shows our proof goal.

Let's trace through the program from the initial state. Whatever is known to hold at ① is also known to hold at ②, namely $10 \leq x$. Then, what we *know* in each subsequent state is this:

① $10 \leq x$

② $10 \leq x \ \&\& \ a == x + 3$

③ $10 \leq x \ \&\& \ a == x + 3 \ \&\& \ b == 12$

④ $10 \leq x \ \&\& \ a == x + 3 \ \&\& \ b == 12 \ \&\& \ y == a + b$

I didn't yet tell you how I came up with these predicates, but I hope they look plausible to you. Now, to show that the method body correctly meets the method specification, we must show that the formula at ④ logically implies the formula at ⑤. In other words, we must show that

$$10 \leq x \ \&\& \ a == x + 3 \ \&\& \ b == 12 \ \&\& \ y == a + b \implies 25 \leq y$$

is a *valid* formula, that is, a formula that evaluates to **true** for any values of its variables.

Exercise 2.1.

Argue—as rigorously as you can—that the formula is indeed valid.

We filled in the formulas ① through ④ in the forward direction. In this way, each formula describes the states

that can reach that point. Each formula thus speaks of “this is what we've got” or “this is what is known to hold here”.

It is also possible to fill in the formulas in the opposite order. 😬 To do that, we start with the *desired* final state and work our way back to the initial state. In this way, each formula speaks of “this is what we want” or “this is what we need to hold here”.

Let's try it. Whatever we want to hold at ⑤ is also what we want to hold at ④, namely $25 \leq y$. Then, what we *need* in each preceding state is this:

④ $25 \leq y$

③ $25 \leq a + b$

② $25 \leq a + 12$

① $25 \leq x + 3 + 12$

Now, to show that the method body correctly meets the specification, we must show that the formula at ⑥ logically implies the formula at ①. That is, we must show that

$10 \leq x \implies 25 \leq x + 3 + 12$

is a valid formula.

Exercise 2.2.

Rigorously argue that the formula is indeed valid.

The direction in which we fill in the formulas does not matter. What matters is that we decorate each program point with a formula. With the formulas in place, we check, for each statement s , that starting in any state satisfying the formula before s leads to a state satisfying the formula written after s . We also check that the method precondition implies the formula used to decorate the initial state (⑥ $\implies$ ①) and check that the formula used to decorate the final state implies the method postcondition (④ $\implies$ ⑤).

This program-reasoning technique was captured by Robert W. Floyd in his ground-breaking paper “Assigning

Meanings to Programs” [50] and is therefore known as *Floyd logic*.

2.2. Hoare Triples

We need a notation for the central building block of Floyd logic, which we can summarize as “a statement takes some pre-states to some post-states”. We’ll use the triples introduced for this purpose by C. A. R. Hoare [63]. For P a predicate on the pre-state of a program S and Q a predicate on the post-state of S , the *Hoare triple*

$$\{ \{ P \} \} S \{ \{ Q \} \}$$

says that if S is started in any state that satisfies P , then S will not crash (or do other bad things) and will terminate in some state satisfying Q .¹

For example, the Hoare triple

$$\{ \{ x == 12 \} \} x := x + 8 \{ \{ x == 20 \} \}$$

expresses the idea that if the program $x := x + 8$ is started in a state where x is 12, then the program is guaranteed to terminate in a state where x has the value 20. This is true about the program $x := x + 8$. That is, we say that this Hoare triple *holds*, or that the Hoare triple is *valid*.

Sidebar 2.0

When discussing program semantics and in various other places of the book, I may omit punctuation required in the Dafny programming language. For example, in the previous paragraph, I spoke of the assignment statement $x := x + 8$. Writing it in Dafny requires a terminating semi-colon, as in $x := x + 8;$.

As another example, the triple

$\{\{ x < 18 \} \} S \{\{ 0 \leq y \} \}$

is notation for saying that whenever you start program S in a state where x is less than 18, then S will terminate in a state where y is non-negative. This triple holds if S is the program $y := 5$ or the program $y := 18 - x$, but it does not hold if S is the program $y := x$ or the program $y := 2 * (x + 3)$, because these programs are not guaranteed to terminate with a non-negative y when started with x being less than 18. Case in point: a state where x is -5 satisfies $x < 18$, and yet the execution of $y := x$ or $y := 2 * (x + 3)$ from that state leads to a state where y is negative.

For brevity, and without any risk of confusion, we usually talk about a program “starting in P ” or “ending in Q ” when we mean “the program starts in a state satisfying predicate P ” or “the program is guaranteed to be crash-free and to terminate in a state satisfying the predicate Q ”.

Exercise 2.3.

Explain rigorously why each of these triples holds:

- a) $\{\{ x == y \} \} z := x - y \{\{ z == 0 \} \}$
- b) $\{\{ \text{true} \} \} x := 100 \{\{ x == 100 \} \}$
- c) $\{\{ \text{true} \} \} x := 2 * y \{\{ x \text{ is even} \} \}$
- d) $\{\{ x == 89 \} \} y := x - 34 \{\{ x == 89 \} \}$
- e) $\{\{ x == 3 \} \} x := x + 1 \{\{ x == 4 \} \}$
- f) $\{\{ 0 \leq x < 100 \} \} x := x + 1 \{\{ 0 < x \leq 100 \} \}$

Exercise 2.4.

For each of the following triples, find initial values of x and y that demonstrate that the triple does not hold.

- a) $\{\{ \text{true} \}\} x := 2 * y \{\{ y \leq x \}\}$
- b) $\{\{ x == 3 \}\} x := x + 1 \{\{ y == 4 \}\}$
- c) $\{\{ \text{true} \}\} x := 100 \{\{ \text{false} \}\}$
- d) $\{\{ 0 \leq x \}\} x := x - 1 \{\{ 0 \leq x \}\}$

Exercise 2.5.

For each of the following triples, come up with some predicate to replace the question mark to make it a Hoare triple that holds. Make your conditions as precise as possible.

- a) $\{\{ 0 \leq x < 100 \}\} x := 2 * x \{\{ ? \}\}$
- b) $\{\{ 0 \leq x \leq y < 100 \}\} z := y - x \{\{ ? \}\}$
- c) $\{\{ 0 \leq x < N \}\} x := x + 1 \{\{ ? \}\}$

Exercise 2.6.

For each of the following triples, come up with some predicate to replace the question mark to make it a Hoare triple that holds. Make your conditions as precise as possible.

- a) $\{\{ -128 \leq x < 0 \}\} x := 1 - x \{\{ ? \}\}$
- b) $\{\{ 0 \leq x \leq y < 100 \}\} y := y - x \{\{ ? \}\}$
- c) $\{\{ x \text{ is even} \ \&\& \ y < 100 \}\} x, y := y, x \{\{ ? \}\}$

Exercise 2.7.

For each of the following triples, come up with some predicate to replace the question mark to make it a Hoare triple that holds. Make your conditions as general as possible.

- a) $\{\{ ? \}\} x := 400 \{\{ x == 400 \}\}$
- b) $\{\{ ? \}\} x := x + 3 \{\{ x \text{ is even} \}\}$
- c) $\{\{ ? \}\} x := 65 \{\{ y \leq x \}\}$

Exercise 2.8.

For each of the following triples, come up with some predicate to replace the question mark to make it a Hoare triple that holds. Make your conditions as general as possible.

- a) $\{\{ ? \} \} b := y < 10 \{\{ b \implies x < y \} \}$
- b) $\{\{ ? \} \} x, y := 2*x, x+y \{\{ 0 \leq x \leq 100 \ \&\& \ y \leq x \} \}$
- c) $\{\{ ? \} \} x := 2*y \{\{ 10 \leq x \leq y \} \}$

2.3. Strongest Postconditions and Weakest Preconditions

In Section 2.1, I sketched two approaches for how to fill in, or *derive*, the decorations needed in Floyd logic to prove a program correct. One of these derivations proceeds in the forward direction, which you may think of as the what-we-have approach. The other derivation proceeds in the backward direction, which you may think of as the goal-oriented approach. It's time to make these two approaches more precise and systematic.

In the forward derivation, we construct the postcondition of a statement from a given precondition. There are many predicates that hold after a statement. For example, all of the following Hoare triples are valid:

```
{ { x == 0 } } y := x + 3 { { y < 100 } }  
{ { x == 0 } } y := x + 3 { { x == 0 } }  
{ { x == 0 } } y := x + 3 { { 0 <= x && y == 3 } }  
{ { x == 0 } } y := x + 3 { { 3 <= y } }  
{ { x == 0 } } y := x + 3 { { true } }
```

As it turns out, if the two triples $\{\{ P \} \} S \{\{ Q_0 \} \}$ and $\{\{ P \} \} S \{\{ Q_1 \} \}$ are both valid, then so is the triple $\{\{ P \} \} S \{\{ Q_0 \ \&\& \ Q_1 \} \}$. In a forward derivation, it is then natural to want to compute the strongest—that is, most precise—post-state

predicate. The forward derivation that does that computes what is referred to as the *strongest postcondition* of a statement (with respect to a given predicate on the statement's pre-state).

Similarly, there are many predicates on the pre-state of a statement that guarantee that the statement establishes a given post-state predicate. For example, all of the following Hoare triples are valid:

```

      {{ x <= 70 }} y := x + 3 {{ y <= 80 }}
{{ x == 65 && y < 21 }} y := x + 3 {{ y <= 80 }}
      {{ x <= 77 }} y := x + 3 {{ y <= 80 }}
{{ x*x + y*y <= 2500 }} y := x + 3 {{ y <= 80 }}
      {{ false }} y := x + 3 {{ y <= 80 }}

```

If the two triples $\{\{ P_0 \} \} S \{\{ Q \} \}$ and $\{\{ P_1 \} \} S \{\{ Q \} \}$ are both valid, then so is the triple $\{\{ P_0 \mid P_1 \} \} S \{\{ Q \} \}$, which suggests that we want to compute the weakest—that is, most general—pre-state predicate. The backward derivation that does that computes what is referred to as the *weakest precondition* of a statement (with respect to a given predicate on the statement's post-state).

The first time you see this goal-oriented, backward derivation, you may find it more difficult to understand than the what-we-have, forward derivation. As it turns out, the backward derivation of formulas is actually easier to construct. Here's how you do it. If you have an assignment $x := E$ that you want to establish a formula Q , then the formula that needs to hold before the assignment is: Q in which you replace all occurrences of x with E . In other words, the most general condition you can write for the question mark in $\{\{ ? \} \} x := E \{\{ Q \} \}$ is Q in which you replace all occurrences of x with E , a formula that I will write using the notation $Q[x := E]$.

For example, if the assignment $y := a + b$ in the program needs to establish the condition $25 \leq y$ (see the program in Section 2.1), then the formula that must hold before the assignment is $25 \leq y$ in which you replace y by $a + b$, that

is, $25 \leq a + b$. In the notation of Hoare triples, the most general condition you can write for the question mark in

$\{\{ ? \}\} y := a + b \{\{ 25 \leq y \}\}$

is $25 \leq a + b$. Easy, right? Similarly, to work out the most general condition you can write for the question mark in

$\{\{ ? \}\} a := x + 3 \{\{ 25 \leq a + 12 \}\}$

you take the desired condition, $25 \leq a + 12$, and replace a with $x + 3$, which gives you $25 \leq x + 3 + 12$.

In a functional program (that is, a program written in a functional programming language), a variable is assigned once when the variable is introduced. (In functional programming, this is known as *let binding*.) In an imperative program, the assignment statement allows you to change the value of a variable. Not only that, but the right-hand side of the assignment can refer to the variable itself, which gives a way to update the variable. For example, the statement

$x := x + 1$

increments the value of x by 1.

The recipe for computing the weakest precondition also works if the right-hand side of the assignment mentions the variable being updated. For example, we follow the same recipe as before for assignments like $x := x + 1$ and $x := 2*x + y$ (it is best to read the following triples from right to left):

$\{\{ x+1 \leq y \}\} x := x + 1 \{\{ x \leq y \}\}$
 $\{\{ 3 * (2*x + y) + 5*y < 100 \}\} x := 2*x + y \{\{ 3*x + 5*y < 100 \}\}$

2.3.0. Swap

Let's compute weakest preconditions to verify the correctness of a program that swaps the values stored in x and y :


```

var tmp := x;
x := y;
y := tmp;
{ { x == X && y == Y } }
{ { y == Y && x == X } }
var tmp := x;
{ { y == Y && tmp == X } }
x := y;
{ { x == Y && tmp == X } }
y := tmp;
{ { x == Y && y == X } }

```

Figure 2.0. To prove that this program fragment swaps the values of x and y , the reasoning is shown backwards from the desired postcondition. One then has to show that the first annotation implies the second.

To specify what we expect this program to do, we need a way in the postcondition to refer to the initial values of x and y . When we use Hoare triples, this feat is accomplished by introducing *logical variables*, that is, variables that stand for some values in our proof, but that cannot be used in the program. Using X and Y as logical variables, we write

```

{ { x == X && y == Y } }
var tmp := x;
x := y;
y := tmp;
{ { x == Y && y == X } }

```

This program consists of three statements, sequentially composed. To compute weakest preconditions for a sequence of statements, we simply do them one at the time, as is suggested by the following chaining sequence of Hoare triples:

```

{ { x == X && y == Y } }
{ { ? } }
var tmp := x;
{ { ? } }
x := y;

```

```

{{ ? }}
y := tmp;
{{ x == Y && y == X }}

```

In this sequence, we intend to use weakest preconditions to compute predicates for the question marks. When we are done, the first question mark will have been replaced by the weakest precondition of the sequence of statements. We then need to prove that the given precondition implies the weakest precondition. Here we go—one at a time, from back to front, see Figure 2.0.

```

{{ x == X && y == Y }}
{{ y - (y - x) + (y - x) == Y && y - (y - x) == X }}
x := y - x;
{{ y - x + x == Y && y - x == X }}
y := y - x;
{{ y + x == Y && y == X }}
x := y + x;
{{ x == Y && y == X }}

```

The diagram illustrates backward reasoning for proving the correctness of a swapping program. It shows a sequence of formulas from top to bottom, connected by arrows indicating the flow of reasoning from the final goal back to the initial precondition.

- The top formula is the goal: $\{ \{ x == X \ \&\& \ y == Y \} \}$.
- The second formula is an intermediate step: $\{ \{ y - (y - x) + (y - x) == Y \ \&\& \ y - (y - x) == X \} \}$.
- The third formula is another intermediate step: $\{ \{ y - x + x == Y \ \&\& \ y - x == X \} \}$.
- The fourth formula is another intermediate step: $\{ \{ y + x == Y \ \&\& \ y == X \} \}$.
- The bottom formula is the final precondition: $\{ \{ x == Y \ \&\& \ y == X \} \}$.

Arrows indicate the backward reasoning process: from the bottom formula to the fourth, from the fourth to the third, from the third to the second, and from the second to the top formula.

Figure 2.1. Backward reasoning for proving the correctness of this program for swapping integers.

To come up with these intermediate formulas, we just apply the recipe for computing weakest preconditions. It's like when we do long multiplication—we systematically apply the steps we've learned. Next, we need to do a proof of the logical implication

$$x == X \ \&\& \ y == Y \implies y == Y \ \&\& \ x == X$$

In this case, the implication holds trivially. So, we have proved that the program above does swap x and y .

2.3.1. A notation for program-proof bookkeeping

Let's repeat the exercise of verifying that a program correctly swaps, but for a different program. This time, we assume x and y to hold integer values:

```
{ { x == X && y == Y } }
x := y - x;
y := y - x;
x := y + x;
{ { x == Y && y == X } }
```

Constructing weakest preconditions, we get what is shown in Figure 2.1.

Filling in the missing parts in the order suggested by the arrows, all we did was perform substitutions according to the recipe for constructing weakest preconditions. Now, we have a proof obligation:

```
x == X && y == Y
==>
y - (y - x) + (y - x) == Y && y - (y - x) == X
```

After some arithmetic simplifications, we see that this implication holds, so the program correctly swaps.

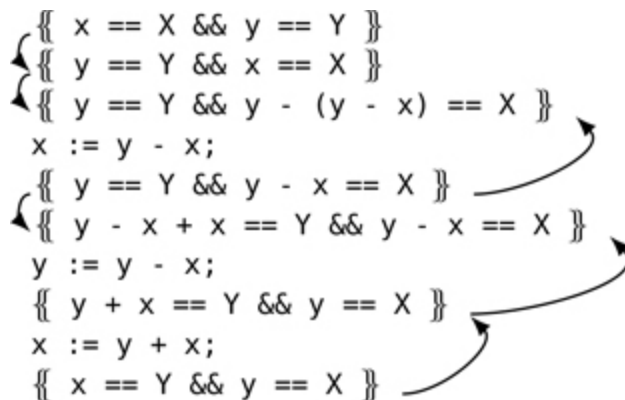


Figure 2.2. Consecutive annotations in this program fragment show simplifications of the formulas. The formula in one such annotation implies the formula in the next annotation. I find such simplifications easiest to read in the forward direction (connected by implication). In contrast, annotations around a program statement are constructed backwards (using weakest preconditions) as shown by the arrows.

When we construct weakest preconditions like we did, it helps to simplify our formulas as we go along, rather than doing so only at the end. A convenient notation for this is to, working backward, write the simplified formula on the line above the one we just wrote. If we perform several simplifications, we can write several formulas above each other. If we want to, we are also allowed to strengthen the formula as we continue working backward. (Beware! A common mistake is to *weaken* the formula as you go backward. This is not allowed, because it will not give you a proof.) In the end, our derivation shows a sequence of formulas of program statements. For every pair of consecutive formulas, we need to prove that the former implies the latter, and every program statement between two formulas must be a valid Hoare triple.

Figure 2.2 shows the previous program with intermediate simplifications, as just described. The arrows on the right show constructions of weakest preconditions, thus forming valid Hoare triples. The arrows on the left connect consecutive annotations. These may be constructed in the backward direction as strengthenings, but many people are more conditioned to check them as implications in the forward direction. Therefore, I'm showing the arrows on the left in the forward direction.

Exercise 2.9.

Verify that the following program correctly swaps, where $\wedge$ denotes xor:

```
x := x ^ y;
y := x ^ y;
x := x ^ y;
```

You can use the commutativity and associativity of xor, as well as the properties $x \wedge x == 0$ and $x \wedge 0 == x$.

Exercise 2.10.

Find the error in the following proof attempt:

```

{{ x == 0 }}
{{ x == 0 && y == 6 }}
x := x + 2
{{ x == 2 && y == 6 }}
{{ x + y == 8 }}
y := x + y
{{ y == 8 }}

```

2.3.2. Simultaneous assignments

Some languages allow several variables to be assigned in one statement. This is called *simultaneous assignment*. For example,

```
x, y := 10, 3;
```

sets x to 10 and, at the same time, sets y to 3. As another example,

```
x, y := x + y, x - y;
```

computes the sum of x and y and the difference between x and y , and then updates x and y to these, respectively. Note that all of the right-hand sides are computed before any variable is assigned. This is what makes the assignment “simultaneous”, and it is what distinguishes it from a *sequence* of two assignments:

```
x := x + y; y := x - y;
```

The semantics of simultaneous assignment is no more difficult than an assignment to a single variable. All we need is for the weakest-precondition substitution to happen simultaneously as well. More precisely, for an assignment $x, y := E, F$ to establish a condition Q , the condition we need before the assignment is $Q[x, y := E, F]$, that is, Q in which we simultaneously substitute E for x and F for y .

Simultaneous assignment gives us a ridiculously easy way to write a program for swapping the values of two variables: $x, y := y, x$. Here is its proof:

```

{{ x == X && y == Y }}
{{ y == Y && x == X }}

```

```
x, y := y, x
{{ x == Y && y == X }}
```

Exercise 2.11.

Fill in the question marks in the following Hoare triples. Simplify the formulas you obtain by computing weakest preconditions.

- a) $\{\{ ? \}\} x, y := 6, 7 \{\{ x < 10 \ \&\& \ y \leq z \}\}$
- b) $\{\{ ? \}\} x, y := x + 1, 2 * x \{\{ y - x == 3 \}\}$
- c) $\{\{ ? \}\} x := x + 1; y := 2 * x \{\{ y - x == 3 \}\}$

2.3.3. Variable introduction

Local variables are introduced with the **var** statement. Most often, we use it together with an assignment, like **var** tmp := x in Section 2.3.0. So far in treating the semantics, I have ignored the actual variable introduction (that is, the “**var**” part) and focused on the assignment. But variable introduction also has semantics. To talk about it, we should think of **var** as being separate from any initial assignment that shares the syntax. That is, **var** tmp := x is really two statements:

```
var tmp;
tmp := x;
```

Every variable has a type, but the type of a local variable is usually inferred, so you may not see it explicitly in the program.

The statement **var** x introduces x, but does not promise anything about the initial value of x (other than that x will have some value of its type). Thus, in order for **var** x to establish a predicate q, what we need before the variable introduction is that q hold for all values of x. Written as a Hoare triple, we have

```
{{ forall x :: Q }} var x {{ Q }}
```

For example, for an integer variable x , suppose Q is $0 \leq x < 100$. Then, the following is a valid Hoare triple:

```
{{ forall x :: 0 <= x < 100 }} var x {{ 0 <= x < 100 }}
```

The condition **forall** $x :: 0 \leq x < 100$ is just **false**, because the condition $0 \leq x < 100$ does not hold universally of all integers (case in point: 102). So, this says that **var** x in no way guarantees that x will end up in the range from 0 to 100.

Generally, if Q mentions x in any nontrivial way, then there's no way to prove that **var** x will establish Q . But don't go thinking that variable introduction is never useful—the semantics just properly captures the fact that uninitialized variables can have any value. If we want to depend on what value the variable has, then we had better first assign to it.

As an example, let's prove that we can introduce a local variable to temporarily store a value we intend to assign to another variable. We know from the weakest precondition of assignment that $x := E$ will establish Q provided the statement is started in $Q[x := E]$. When will **var** $tmp := E; x := tmp$ establish Q , where tmp is a variable that is not used in either E or Q ? The following chain of Hoare triples show the answer to be $Q[x := E]$, just as for the direct assignment $x := E$.

```
{{ Q[x := E] }}
{{ forall tmp :: Q[x := E] }}
var tmp
{{ Q[x := E] }}
{{ Q[x := tmp][tmp := E] }}
tmp := E
{{ Q[x := tmp] }}
x := tmp
{{ Q }}
```

As usual, the lines above are best read from bottom to top. The simplification from $Q[x := tmp][tmp := E]$ to $Q[x := E]$ is justified by the fact that tmp does not occur in Q . The simplification from

```
forall tmp :: Q[x := E]
```

to $Q[x := E]$ is justified by the fact that `tmp` does not occur in the expression $Q[x := E]$.

Exercise 2.12.

Here is an example that does not depend on the initial value of x . Prove:

```
{{ true }} var x; x := x * x {{ 0 <= x }}
```

2.3.4. The complicated one

Okay, it wouldn't be fair not to show you how to compute the strongest postcondition for an assignment, even if we won't use it much. It is not the simple substitution we had for weakest preconditions. It's complicated. Brace yourself.

To motivate the recipe for strongest postconditions, consider the following, innocent-looking Hoare triple:

```
{{ w < x < y }} x := 100 {{ ? }}
```

It's clear that $x == 100$ is a postcondition, but is it the strongest? The condition $w < x < y$ no longer holds, because the assignment overwrites the previous value of x . However, in the post-state, there exists some value x_0 (namely, the value of x in the pre-state) for which $w < x_0 < y$ holds. So, the most precise condition we can write for the question mark is:

```
exists  $x_0 :: w < x_0 < y \ \&\& \ x == 100$ 
```

Since the variables involved are integers, this formula is logically equivalent to

```
 $w + 1 < y \ \&\& \ x == 100$ 
```

In general, the right-hand side of the assignment may also mention x , so we'll need to use the x_0 there, too. So here it is:

```
{{ P }} x := E {{ exists  $x_0 :: P[x := x_0] \ \&\& \ x == E[x := x_0]$  }}
```

Aren't you glad you asked? 🙄

Exercise 2.13.

Replace the ? in the following Hoare triples by computing strongest postconditions. In each case, simplify the formula, if possible.

- a) $\{\{ y == 10 \}\} x := 12 \{\{ ? \}\}$
- b) $\{\{ 98 \leq y \}\} x := x + 1 \{\{ ? \}\}$
- c) $\{\{ 98 \leq x \}\} x := x + 1 \{\{ ? \}\}$
- d) $\{\{ 98 \leq y < x \}\} x := 3 * y + x \{\{ ? \}\}$

Exercise 2.14.

Verify the correctness of the various swap programs in Section 2.3.0 using strongest postconditions.

2.4. $\mathcal{WP}$ and $\mathcal{SP}$

We can save many words by introducing some notation. Suppose s is a statement. Then, for any predicate P on the pre-state of s , let's define $\mathcal{SP}[[s, P]]$ to denote the strongest postcondition of s with respect to P . Similarly, for any predicate Q on the post-state of s , let's define $\mathcal{WP}[[s, Q]]$ to denote the weakest precondition of s with respect to Q .

2.4.0. Working backward

Using this notation, the semantics of assignment is defined as follows:

$$\begin{aligned}\mathcal{WP}[[x := E, Q]] &= Q[x := E] \\ \mathcal{SP}[[x := E, P]] &= \textbf{exists } x_0 :: P[x := x_0] \ \&\& \ x == E[x := x_0]\end{aligned}$$

These $\mathcal{WP}$ and $\mathcal{SP}$ equations apply for assignments to one variable as well as simultaneous assignment to several variables, as long as x , x_0 , and E are lists of the same length.

It is very useful to *work backward* over an assignment statement, because it lets you convert a condition Q that

you want to hold after the assignment into a condition that needs to hold prior to the assignment. I will make frequent use of this step in Part 2.

Exercise 2.15.

Practice working backward: Compute the weakest precondition of the following programs with respect to the postcondition $x + y \leq 100$.

- a) $x := 20$ b) $x := x + 1$
- c) $x := 2 * x$ d) $x := -x$
- e) $x := y$ f) $x := x + y$
- g) $x := y - x$ h) $x := x - y$
- i) $z := x + y$

In each case, simplify your answer.

Exercise 2.16.

Compute the strongest postcondition of the following programs with respect to the precondition $x + y \leq 100$.

- a) $x := 5$ b) $x := x + 1$
- c) $x := 2 * y$ d) $z := x + y$

In each case, simplify your answer.

2.4.1. Local variables

In Section 2.3.3, I presented the semantics of `var` in terms of weakest preconditions. Here is its semantics in terms of both $\mathcal{WP}$ and $\mathcal{SP}$:

$$\begin{aligned}\mathcal{WP}[\text{var } x, Q] &= \text{forall } x :: Q \\ \mathcal{SP}[\text{var } x, P] &= \text{exists } x :: P\end{aligned}$$

Exercise 2.17.

Compute

a) $\mathcal{WP}[\text{var } x, x \leq 100]$

b) $\mathcal{SP}[\text{var } x, x \leq 100]$

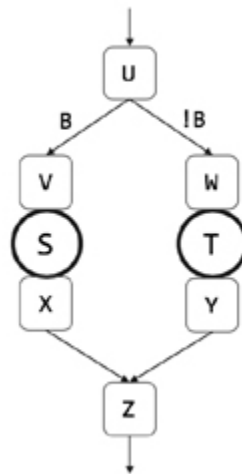
In each case, simplify your answer.

2.5. Conditional Control Flow

To learn what Floyd logic says about conditional statements, like

if B { S } **else** { T }

it is instructive to draw the control-flow graph. Here, I'm using U, V, W, X, Y, and Z to denote predicates at the beginning and end of the conditional statement and of its two substatements, S and T:



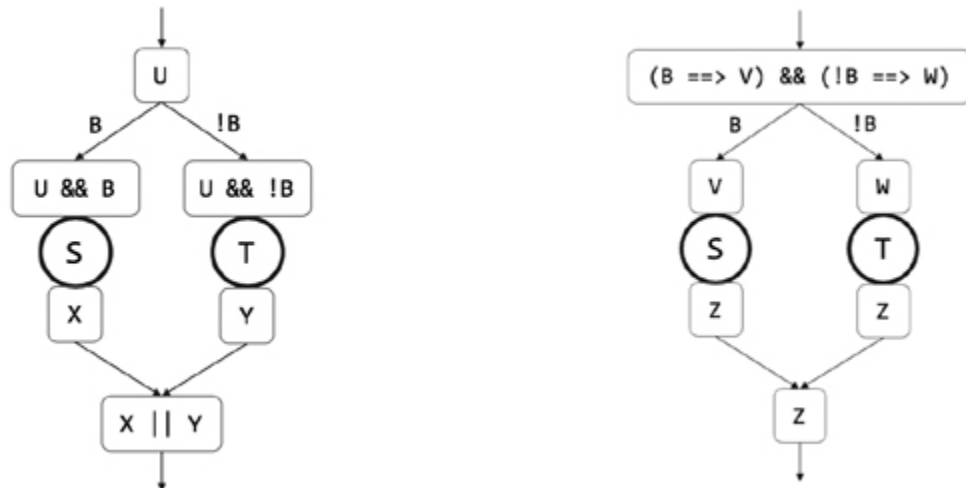
Floyd logic tells us that this decorated program is correct if the following are valid formulas and valid Hoare triples:

- $U \ \&\& \ B \implies V$
- $U \ \&\& \ !B \implies W$
- $\{\{ V \}\} S \ \{\{ X \}\}$
- $\{\{ W \}\} T \ \{\{ Y \}\}$
- $X \implies Z$
- $Y \implies Z$

If we're computing strongest postconditions, then we

- set V to $U \ \&\& \ B$ and set W to $U \ \&\& \ !B$,
- compute X as the strongest postcondition for V, S and compute Y as the strongest postcondition for W, T , and
- set Z to $X \ || \ Y$.

Pictorially, we have the diagram shown here to the left:



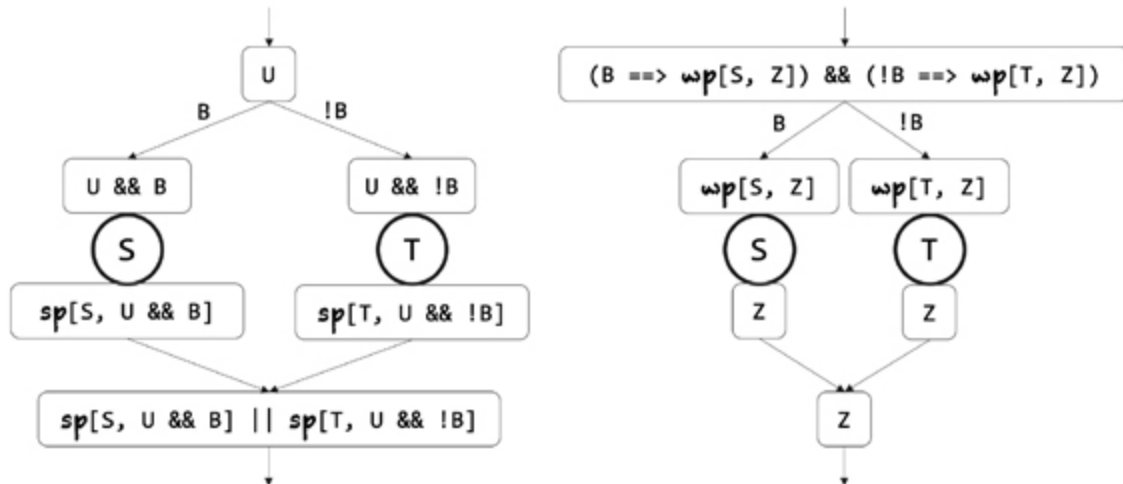
To compute weakest preconditions, we instead

- set both X and Y to Z ,
- compute V as the weakest precondition of S, X and compute W as the weakest pre-condition of T, Y , and
- set U to $(B \ ==> \ V) \ \&\& \ (!B \ ==> \ W)$.

Pictorially, we have the diagram shown above to the right.

Regardless of how you come up with the intermediate predicates in these diagrams, what you should observe is how the predicates are transformed as you follow—either forwards or backwards—the edges of the control-flow graph.

Here are the same flow diagrams, but showing $\mathcal{SP}$ and $\mathcal{WP}$ in the diagrams:



Exercise 2.18.

(a) Draw a decorated flow diagram as the ones above for the conditional statement

if $x < 3$ **{** $x, y := x + 1, 10;$ **}** **else** **{** $y := x;$ **}**

(b) In the forward direction and using $x + y == 100$ as the precondition (that is, as U), fill in the formulas in the remaining boxes. (c) In the backward direction and using $x + y == 100$ as the postcondition (that is, as Z), fill in the formulas in the remaining boxes.

2.5.0. Just the formulas, ma'am

Written with just formulas—no diagrams—the meaning of **if** statements in terms of SP and WP is:

$$\begin{aligned}
 SP[\text{if } B \{ S \} \text{ else } \{ T \}, P] &= \\
 &SP[S, P \ \&\& \ B] \ || \ SP[T, P \ \&\& \ !B] \\
 WP[\text{if } B \{ S \} \text{ else } \{ T \}, Q] &= \\
 &(B \implies WP[S, Q]) \ \&\& \ (!B \implies WP[T, Q])
 \end{aligned}$$

Exercise 2.19.

Suppose $x < 100$ is known to hold before the statement

if $x < 20$ **{** $y := 3;$ **}** **else** **{** $y := 2;$ **}**

then what is the strongest condition you know after the statement? In other words, compute the strongest postcondition of the statement with respect to $x < 100$. Simplify the condition after you have computed it.

Exercise 2.20.

Suppose you want $x + y == 22$ to hold after the statement

```
if  $x < 20$  {  $y := 3$ ; } else {  $y := 2$ ; }
```

then in which states can you start the statement? In other words, compute the weakest precondition of the statement with respect to $x + y == 22$. Simplify the condition after you have computed it.

Exercise 2.21.

Compute the weakest precondition for the following statement with respect to $y < 10$. Simplify the condition.

```
if  $x < 8$  {  
  if  $x == 5$  {  
     $y := 10$ ;  
  } else {  
     $y := 2$ ;  
  }  
} else {  
   $y := 0$ ;  
}
```

Exercise 2.22.

Compute the weakest precondition for the following statement with respect to $y \% 2 == 0$ (that is, “y is even”). Simplify the condition.

```
if  $x < 10$  {  
  if  $x < 20$  {  $y := 1$ ; } else {  $y := 2$ ; }  
} else {  
   $y := 4$ ;  
}
```

Exercise 2.23.

Compute the weakest precondition for the following statement with respect to $y \% 2 == 0$ (that is, “y is even”). Simplify the condition.

```
if x < 8 {  
  if x < 4 { x := x + 1; } else { y := 2; }  
} else {  
  if x < 32 { y := 1; } else { }  
}
```

Exercise 2.24.

Determine under which circumstances the following program establishes $0 \leq y < 100$. If you're starting to get a hang of how to compute weakest preconditions, try to do the computation in your head. Write down the answer you come up with, and then write out the full $\mathcal{WP}$ computations to check that you got the right answer.

```
if x < 34 {  
  if x == 2 { y := x + 1; } else { y := 233; }  
} else {  
  if x < 55 { y := 21; } else { y := 144; }  
}
```

2.6. Sequential Composition

Another way to compose two statements is to follow one by the other. This simple idea goes by the multi-syllable name *sequential composition*. For statements S and T , we can convey the Floyd logic semantics of $S;T$ by writing two overlapping Hoare triples

$$\{\{ P \}\} S \{\{ Q \}\} T \{\{ R \}\}$$

This is valid if the following two individual Hoare triples are valid:

- $\{\{ P \}\} S \{\{ Q \}\}$
- $\{\{ Q \}\} T \{\{ R \}\}$

There's no reason we have to involve the predicate Q , because there is no loss in generality by setting Q to either $\mathcal{SP}[S, P]$ or $\mathcal{WP}[T, R]$. In other words, we can directly define the validity of the Hoare triple

$$\{\{ P \}\} S;T \{\{ R \}\}$$

in terms of strongest postconditions or weakest preconditions. In formulas, we have

$$\begin{aligned}\mathcal{SP}[S;T, P] &= \mathcal{SP}[T, \mathcal{SP}[S, P]] \\ \mathcal{WP}[S;T, R] &= \mathcal{WP}[S, \mathcal{WP}[T, R]]\end{aligned}$$

Notice that the compositions in these two right-hand sides go in opposite directions— $\mathcal{SP}$ is computed forwards from P and $\mathcal{WP}$ is computed backward from R .

Exercise 2.25.

Which of the following Hoare-triple combinations are valid?

- a) $\{\{ 0 \leq x \}\} x := x + 1 \{\{ -2 \leq x \}\} y := 0 \{\{ -10 \leq x \}\}$
- b) $\{\{ 0 \leq x \}\} x := x + 1 \{\{ \text{true} \}\} x := x + 1 \{\{ 2 \leq x \}\}$
- c) $\{\{ 0 \leq x \}\} x := x + 1; x := x + 1 \{\{ 2 \leq x \}\}$
- d) $\{\{ 0 \leq x \}\} x := 3 * x; x := x + 1 \{\{ 3 \leq x \}\}$
- e) $\{\{ x < 2 \}\} y := x + 5; x := 2 * x \{\{ x < y \}\}$

Exercise 2.26.

Practice working backward over several assignment statements. Compute $\mathcal{WP}$ with respect to $x + y \leq 100$ for the following programs:

- a) $x := x + 1; y := x + y$
- b) $y := x + y; x := x + 1$
- c) $x, y := x + 1, x + y$

Exercise 2.27.

Compute $\mathcal{SP}$ with respect to $x + y \leq 100$ for the following programs:

- a) $x := x + 1; y := x + y$
- b) $y := x + y; x := x + 1$
- c) $x, y := x + 1, x + y$

Exercise 2.28.

Compute the strongest postcondition and weakest precondition of the following statements, each with respect to the predicate $x + y < 100$. Simplify the predicates.

- a) $x := 32; y := 40$
- b) $x := x + 2; y := y - 3 * x$

Exercise 2.29.

Compute

- a) $\mathcal{WP}[\text{var } x; x := 10, x \leq 100]$
- b) $\mathcal{SP}[\text{var } x; x := 10, x \leq 100]$

Exercise 2.30.

Compute the strongest postcondition and weakest precondition of the following statements, each with respect to the predicate $x < 10$:

```
if x % 2 == 0 { y := y + 3; } else { y := 4; }  
if y < 10 { y := x + y; } else { x := 8; }
```

2.7. Method Calls and Postconditions

Methods are opaque. This means we reason about them in terms of their specifications, not their implementations (their bodies). Consider our Triple method from Section 1.4 again:

```
method Triple(x: int) returns (y: int)
```

ensures $y == 3 * x$

The basic idea is that we expect to be able to prove Hoare triples like

`{{ true }} t := Triple(u + 3) {{ t == 3 * (u + 3) }}`

(This *is* what you'd expect, right?) To make this precise, we need to consider several details. Let's start with the parameters.

2.7.0. Parameters

For illustration, let's use the call

`t := Triple(u + 3)`

The specification uses the formal parameters x and y , whereas the call site uses the actual parameters $u + 3$ and t . Somehow, we must connect these. Because of various possible name clashes, including any overlap in the names of the formal and actual parameters, we start by giving the formal parameters *fresh* names. That is, we rename the formal parameters to names that are not already used in our verification. Stated in yet one more way, the names are fresh if they are unique to this call site.

For this step, I will rename x to x' and y to y' . With these fresh names, the method and its specification look like this:

method Triple(x' : **int**) **returns** (y' : **int**)
 ensures $y' == 3 * x'$

The meaning of the call can now be explained as the sequential composition of first assigning the actual in-parameters to the freshly named formal in-parameters and then assigning the freshly named formal out-parameters to the actual out-parameters, where—importantly—we get to assume the relation between the freshly named variables entailed by the postconditions. For our example, these two assignments are $x' := u + 3$ and $t := y'$, and the relation we get to assume is $y' == 3 * x'$.

2.7.1. Assumptions

To make things more precise, it would be convenient to have a program statement that introduced an assumption. For this purpose, let me define a statement **assume** E for a boolean expression E . For any pre-state predicate P and post-state predicate Q , the semantics of **assume** is given by the following two equations:

$$\begin{aligned} SP[\text{assume } E, P] &= P \ \&\& \ E \\ WP[\text{assume } E, Q] &= E \implies Q \end{aligned}$$

The first of these equations says that if you start **assume** E in a state satisfying P , then you get to assume that both P and E hold afterwards. The second equation says that if you want to prove Q to hold after **assume** E , then it suffices to prove, before the statement, that Q holds under the assumption E .

The **assume** statement is fictitious. That is, it is not a statement that a compiler could handle. After all, it introduces an assumption, much like making a wish come true. Nevertheless, if you just think of the fictitious statement as “oh, at this point, I apparently get to assume that condition E holds”, then this **assume** statement is quite handy as a primitive in defining the semantics of non-fictitious statements. Like the call statement.

2.7.2. Semantics of method call with a postcondition

Using renaming and an assumption, we can define the semantics of our illustrative call. Above, we already freshly renamed the formal parameters of `Triple`. Next, the semantics of the call is given by the following program snippet:

```
var x', y';  
x' := u + 3;  
assume y' == 3 * x';
```

$t := y'$

In other words, I'm saying that you reason about the call

$t := \text{Triple}(u + 3)$

in the same way as you reason about these four statements.

In terms of weakest preconditions, we thus have, for any predicate Q ,

```

$$\begin{aligned} & \mathcal{WP}[\![t := \text{Triple}(u + 3), Q]\!] \\ = & \quad \{ \text{definition of a call in terms of the four statements} \} \\ & \mathcal{WP}[\![\text{var } x', y'; x' := u + 3; \text{assume } y' == 3 * x'; t := y', Q]\!] \\ = & \quad \{ \text{definition of } \mathcal{WP} \text{ for } ; \} \\ & \mathcal{WP}[\![\text{var } x', y', \mathcal{WP}[\![x' := u + 3, \\ & \quad \mathcal{WP}[\![\text{assume } y' == 3 * x', \mathcal{WP}[\![t := y', Q]\!] \ ] \ ] \ ] \\ = & \quad \{ \text{definition of } \mathcal{WP} \text{ for } := \text{ and } \text{assume} \text{ and } \text{var} \} \\ & \text{forall } x', y' :: (y' == 3 * x' \implies Q[t := y'])[x' := u + 3] \\ = & \quad \{ \text{apply the substitution for } x' \} \\ & \text{forall } x', y' :: y' == 3 * (u + 3) \implies Q[t := y'] \\ = & \quad \{ x' \text{ is not used} \} \\ & \text{forall } y' :: y' == 3 * (u + 3) \implies Q[t := y'] \\ = & \quad \{ \text{logic} \} \\ & Q[t := y'][y' := 3 * (u + 3)] \\ = & \quad \{ \text{apply substitution for } y' \} \\ & Q[t := 3 * (u + 3)] \end{aligned}$$

```

Let's try this out. As an example, take Q to be $t == 54$. That is, if we want the call $t := \text{Triple}(u + 3)$ to establish $t == 54$, then what needs to hold before we make the call?

```

$$\begin{aligned} & \mathcal{WP}[\![t := \text{Triple}(u + 3), t == 54]\!] \\ = & \quad \{ \mathcal{WP} \text{ of call, see above} \} \\ & (t == 54)[t := 3 * (u + 3)] \\ = & \quad \{ \text{substitutions} \} \\ & 3 * (u + 3) == 54 \\ = & \quad \{ \text{arithmetic} \} \\ & u == 15 \end{aligned}$$

```

So, if we want $t := \text{Triple}(u + 3)$ to establish $t == 54$, we must make sure to make the call when $u == 15$. This matches our understanding of the call and its specification, because if u is 15, then $u + 3$ is 18, and then `Triple` returns 54.

More generally, for a method

```
method M(x: X) returns (y: Y)
ensures R
```

the semantics of a call to M is

$$\mathcal{WP}[\![t := M(E), Q]\!] =$$

$$\text{forall } x', y' :: (R[x, y := x', y'] \implies Q[t := y'])[x' := E]$$

where x' and y' are fresh names and Q is any predicate on the post-state of the call. Notice that I included the renaming in this equation, rather than first doing a renaming step. Since x' is fresh, it does not occur in Q , so we can simplify the equation by distributing the substitutions:

$$\mathcal{WP}[\![t := M(E), Q]\!] =$$

$$\text{forall } y' :: R[x, y := E, y'] \implies Q[t := y']$$

What I wrote applies to any number of in- and out-parameters. That is, it applies equally well if x , y , and t are lists of variables and E is a list of expressions (provided x and E have the same lengths, y and t have the same lengths, and there are no duplicates among the variables in the list t).

Exercise 2.31.

Compute the weakest precondition for a call $t := \text{Abs}(7 * u)$ with respect to the postcondition $u < t$, where Abs is defined as follows:

```
method Abs(x: int) returns (y: int)
ensures 0 <= y && (x == y || x == -y)
```

Simplify your answer.

Exercise 2.32.

Compute the weakest precondition for a call $t := \text{Max}(2 * u, u + 7)$ with respect to the postcondition $t \% 2 == 0$, where Max is defined as follows:

```
method Max(x: int, y: int) returns (m: int)
ensures m == x || m == y
ensures x <= m && y <= m
```

| Simplify your answer.

Since we have a way to give the semantics of a call in terms of the four more primitive statements, the strongest-postcondition semantics of a call follows. For any predicate P on the pre-state of the illustrative call to `Triple`, we have:

$$\begin{aligned} & \mathcal{SP}[\text{t} := \text{Triple}(u + 3), P] \\ = & \quad \{ \text{definition of a call in terms of the four statements} \} \\ & \mathcal{SP}[\text{var } x', y'; x' := u + 3; \text{assume } y' == 3 * x'; t := y', P] \\ = & \quad \{ \text{definition of } \mathcal{SP} \text{ for } ; \} \\ & \mathcal{SP}[\text{var } x', y', \mathcal{SP}[t := y', \\ & \quad \mathcal{SP}[\text{assume } y' == 3 * x', \mathcal{SP}[x' := u + 3, P]]]] \\ = & \quad \{ \text{definition of } \mathcal{SP} \text{ for } :=, \text{ simplifying since } x' \text{ is} \\ & \text{fresh} \} \\ & \mathcal{SP}[\text{var } x', y', \mathcal{SP}[t := y', \\ & \quad \mathcal{SP}[\text{assume } y' == 3 * x', P \ \&\& \ x' == u + 3]]]] \\ = & \quad \{ \text{definition of } \mathcal{SP} \text{ for } \text{assume} \} \\ & \mathcal{SP}[\text{var } x', y', \mathcal{SP}[t := y', \\ & \quad P \ \&\& \ x' == u + 3 \ \&\& \ y' == 3 * x']]] \\ = & \quad \{ \text{definition of } \mathcal{SP} \text{ for } :=, \text{ where } t_0 \text{ is fresh} \} \\ & \mathcal{SP}[\text{var } x', y', \\ & \quad \text{exists } t_0 :: (P \ \&\& \ x' == u + 3 \ \&\& \ y' == 3 * x')[t := t_0] \ \&\& \\ & \quad t == y']] \\ = & \quad \{ \text{simplification} \} \\ & \mathcal{SP}[\text{var } x', y', \\ & \quad \text{exists } t_0 :: P[t := t_0] \ \&\& \ x' == u + 3 \ \&\& \ y' == 3 * x' \ \&\& \ t \\ = & \quad y']] \\ = & \quad \{ \text{definition of } \mathcal{SP} \text{ for } \text{var} \} \\ & \text{exists } x', y' :: \text{exists } t_0 :: \\ & \quad P[t := t_0] \ \&\& \ x' == u + 3 \ \&\& \ y' == 3 * x' \ \&\& \ t == y' \\ = & \quad \{ \text{logic simplification for } x' \} \\ & \text{exists } y' :: \text{exists } t_0 :: \\ & \quad (P[t := t_0] \ \&\& \ y' == 3 * x' \ \&\& \ t == y')[x' := u + 3] \\ = & \quad \{ \text{apply substitution for } x' \} \\ & \text{exists } y' :: \text{exists } t_0 :: \\ & \quad P[t := t_0] \ \&\& \ y' == 3 * (u + 3) \ \&\& \ t == y' \\ = & \quad \{ \text{logic simplification for } y' \} \\ & (\text{exists } t_0 :: P[t := t_0] \ \&\& \ y' == 3 * (u + 3))[y' := t] \end{aligned}$$

$$= \quad \{ \text{apply substitution for } y' \} \\ \text{exists } t_0 :: P[t := t_0] \ \&\& \ t == 3 * (u + 3)$$

In the general case where M is the method above with the postcondition R , we have

$$\mathcal{SP}[t := M(E), P] = \\ \text{exists } t_0 :: P[t := t_0] \ \&\& \ R[x := E[t := t_0]][y := t]$$

where t_0 is fresh. (I don't know about you, but I find the weakest-precondition formulation easier to understand than this strongest-postcondition formulation.)

2.8. Assert Statements

In what I described above for method calls, I only showed postconditions. Before I cover preconditions, I will define the semantics of **assert** statements.

The statement **assert** E is a no-op if E holds; otherwise, it crashes your program. So if you want to prove that a condition Q holds after the statement, then you must prove that both E and Q hold before it— E to prevent the assertion from crashing, and Q because the assertion doesn't change the state. This is captured in the following $\mathcal{WP}$ equation:

$$\mathcal{WP}[\text{assert } E, Q] = E \ \&\& \ Q$$

We're always interested in proving our programs to be crash-free, so the effect of the **assert** statement is to introduce a proof obligation.

Exercise 2.33.

Compute the weakest precondition of the following statements with respect to the postcondition $x < 100$. Simplify each answer.

- a) **assert** $y == 25$
- b) **assert** $0 \leq x$
- c) **assert** $x < 200$
- d) **assert** $x \leq 100$
- e) **assert** $0 \leq x < 100$

2.8.0. The real difference between $\mathcal{SP}$ and $\mathcal{WP}$

Here is the $\mathcal{SP}$ equation for **assert**:

$$\mathcal{SP}[\text{assert } E, P] = P \ \&\& \ E$$

It says that the most you can be sure of after a crash-free execution of the statement is that both P and E hold.

Until now, the choice of using $\mathcal{SP}$ versus $\mathcal{WP}$ has seemed rather arbitrary, or perhaps simply a matter of taste. Okay, for assignment statements, $\mathcal{SP}$ is more complicated than $\mathcal{WP}$, but is there any deeper difference between $\mathcal{SP}$ and $\mathcal{WP}$? Yes, there is, and the **assert** statement brings out this difference.

If you would consider defining the formal meaning of statements using just $\mathcal{SP}$, then you would find yourself with two questions about the $\mathcal{SP}$ equation for **assert** above:

0. Under what circumstances does **assert** E crash? For example, will the statement always crash if E does not hold? Of greater interest, can we be assured that the statement does *not* crash if E holds?
1. An assertion is supposed to give us a way to introduce a proof obligation, so how does $\mathcal{SP}$ capture the fact that E is a proof obligation?

The answer to question 0 is that $\mathcal{SP}$ does *not* say when a statement can crash. It only says that *if* the statement does *not* crash, then we know what holds afterwards.

The answer to question 1 is that $\mathcal{SP}$ does not. That is, $\mathcal{SP}$ is incognizant of proof obligations. If you look back to

Section 2.7.1, you may be alarmed to find that $\mathcal{SP}$ is the same for **assert** E as it is for **assume** E. In that section, I had defined **assume** E as a fictitious statement that introduces an assumption. This assumption comes for free, like a wish or a pipe dream, and there is no proof obligation involved. In contrast, the intended meaning of **assert** E is to introduce a proof obligation—we're not just assuming the condition E to hold, but we also want to prove that it does hold. Yet, $\mathcal{SP}[\text{assert } E, P]$ is the same as $\mathcal{SP}[\text{assume } E, P]$. Both of these say that *in the event that* the statement terminates without crashing, then $P \ \&\& \ E$ holds afterwards.

Exercise 2.34.

Suppose we didn't use $\mathcal{WP}$, but we defined the semantics of a statement in terms of $\mathcal{SP}$ alone. Then, according to the $\mathcal{SP}$ equation, would you say that an **assert** statement could lower your Candy Crush score?

In conclusion, $\mathcal{SP}$ tells us the strongest thing we can say upon crash-free execution of a statement, but it does not tell us under which circumstances a statement *is* crash-free. In contrast, $\mathcal{WP}$ tells us the pre-states from which a statement is guaranteed to be crash-free and terminate in a state we'd like. The **assert** and **assume** statements highlight this difference:

$$\begin{aligned}\mathcal{WP}[\text{assert } E, Q] &= E \ \&\& \ Q \\ \mathcal{WP}[\text{assume } E, Q] &= E \implies Q \\ \mathcal{SP}[\text{assert } E, P] &= P \ \&\& \ E \\ \mathcal{SP}[\text{assume } E, P] &= P \ \&\& \ E\end{aligned}$$

These $\mathcal{WP}$ equations say that if we want to establish Q after the statement, then for **assert** E, we must prove that E holds *and* prove that Q holds, whereas for **assume** E, we never need to prove that E holds and we still get to assume E when proving Q . The $\mathcal{SP}$ equations just say that if the statements terminate without crashing, then $P \ \&\& \ E$ holds.

2.8.1. An informal reading

The names of the statements **assert** and **assume** are not accidental. They have been chosen to let us internalize what the statements mean. You can forget the mathematical definitions for these statements if (or: there's no reason to get caught up in the formal details as long as) you understand the statements like this:

The statement **assert** E says “at this program point, we assert that E holds; that is, we expect that E holds and we're going to prove that it does”. Alternatively, “we expect that whenever control reaches this program point, E holds, and we're going to prove that this is indeed the case for all possible executions”.

The statement **assume** E says “at this program point, we have no idea if E holds or not; nevertheless, for the purpose of proving the correctness of our program, we are going to assume that E does hold here”. Alternatively, “when you run the program and control reaches this point, E may or may not hold, but as far as our proof is concerned, we are going to ignore any execution that gets here and E does not hold”.

2.8.2. Using **assume** to reason about calls

How come I dare use **assume** when defining the meaning of calls, as I did in Section 2.7.2? Are we really proving the correctness of our program if the **assume** causes the proof to ignore certain executions? In this case, yes, we are still proving the correctness of our program, because we are going to make sure that no executions are ignored. For a call to a method M , I used the **assume** statement at the call site to assume the post-condition of M . Sooner or later in the reasoning about our program, we're going to prove the

correctness of the implementation of M . In that proof, we are going to ensure that the method implementation establishes the postcondition. In this way, it will always be the case that, when a call returns, the postcondition of the method called holds. So, no executions are ignored after all.

This justifies the use of **assume** in the definition of a call.

2.9. Weakest Liberal Preconditions

My aim in this book is to equip you with an understanding of how to reason about your programs, so that you can rigorously explain (to yourself or to a fellow software engineer, or even to a machine) why a program is correct. There is a beautiful aspect of the underlying semantics that is not strictly needed to prove your programs correct. I'm going to give you a taste of that beautiful aspect in this section. It also explains some of the mystery surrounding the difference between $\mathcal{WP}$ and $\mathcal{SP}$ that I talked about above in Section 2.8.0. If you don't care for such discussions about the underlying semantics, you can skip this section and instead continue with Section 2.10.

2.9.0. Turning capturing proof obligations into a separate concern

In Section 2.8.0, I pointed out that $\mathcal{SP}$ does not capture the proof obligations that ensure crash freedom, whereas $\mathcal{WP}$ does. It has also been clear from the beginning that $\mathcal{SP}$ proceeds in the forward direction of a program, whereas $\mathcal{WP}$ proceeds in the backward direction. Does going forward mean that we cannot speak about (crash-freedom) proof obligations?

No, not entirely. We could define a second semantic function in the forward direction, one that would keep track

of all proof obligations. (See the upcoming Exercise 2.35.) In the backward direction, $\mathcal{WP}$ already does this. That is, as $\mathcal{WP}$ proceeds in the backward direction, it carries with it any proof obligations it comes across. For example,

$$\begin{aligned}
 & \mathcal{WP}[\![x := x + 1; \text{assert } x \leq 10, \text{true}]\!] \\
 = & \quad \{ \text{definition of } \mathcal{WP} \text{ for } ; \} \\
 & \mathcal{WP}[\![x := x + 1, \mathcal{WP}[\![\text{assert } x \leq 10, \text{true}]\!]\!] \] \\
 = & \quad \{ \text{definition of } \mathcal{WP} \text{ for } \text{assert} \} \\
 & \mathcal{WP}[\![x := x + 1, x \leq 10]\!] \\
 = & \quad \{ \text{definition of } \mathcal{WP} \text{ for } := \} \\
 & x + 1 \leq 10
 \end{aligned}$$

Here, we started by computing the weakest precondition necessary to establish **true**, and we end up with something that says $x + 1 \leq 10$ needs to hold in the pre-state. The predicate $x + 1 \leq 10$ is thus a proof obligation.

So, then, if $\mathcal{WP}$ somehow plays double duty, is there a way to split $\mathcal{WP}$ into more primitive semantic functions? Yes. The $\mathcal{WP}$ function we've seen is sometimes known as the *weakest conservative precondition*. We can define it in terms of a *weakest liberal precondition* that ignores crash executions and a semantic function that accumulates proof obligations for avoiding crashes. Let's define these two.

For any statement s and any predicate q on the post-state of s , the weakest liberal precondition of s with respect to q , written $\mathcal{WLP}[\![s, q]\!]$, denotes those pre-states from which every crash-free execution of s terminates in a state satisfying q . That is, if an execution starts in a state that satisfies $\mathcal{WLP}[\![s, q]\!]$, then either the execution crashes or the execution terminates in a state satisfying q . Note, if s is a statement that always terminates without crashing, then $\mathcal{WLP}[\![s, q]\!]$ is the same as $\mathcal{WP}[\![s, q]\!]$.

For any statement s , we need a semantic function that tells us the proof obligations of s , that is, that tells us from which states s is guaranteed not to crash. We already have such a function, namely $\mathcal{WP}[\![s, \text{true}]\!]$. Recalling the

definition of $\mathcal{WP}$, we have that $\mathcal{WP}[s, \text{true}]$ describes those pre-states from which execution of s terminates in a state satisfying **true** (easy-peasy!) and *does not crash*. Since every state satisfies **true**, $\mathcal{WP}[s, \text{true}]$ is what we're looking for.

Putting these two pieces together, we have the following important equation, for every s and q :

$$\mathcal{WP}[s, q] = \mathcal{WLP}[s, q] \ \&\& \ \mathcal{WP}[s, \text{true}]$$

2.9.1. The connection between $\mathcal{WLP}$ and $\mathcal{SP}$

I argued in Section 2.8.0 that $\mathcal{WP}$ and $\mathcal{SP}$ are not the backward and forward formulations of the same function. But there is a way that $\mathcal{WLP}$ and $\mathcal{SP}$ are. It turns out that, for every s , p , and q , the following holds:

$$\mathcal{SP}[s, p] \implies q \text{ if and only if } p \implies \mathcal{WLP}[s, q]$$

I have written $\mathcal{SP}$ and $\mathcal{WLP}$ as taking two arguments, a program statement and a predicate. For a fixed s , we can think of $\mathcal{SP}[s, \_]$ and $\mathcal{WLP}[s, \_]$ as functions of a predicate. Whenever such functions satisfy the if-and-only-if relation above, they are said to form a *Galois connection* [19]. Function $\mathcal{SP}[s, \_]$ is called the *lower adjoint* and function $\mathcal{WLP}[s, \_]$ is called the *upper adjoint*.

Only certain functions can form a Galois connection. If a function has a corresponding upper adjoint, then that upper adjoint is unique, but the function may not have an upper adjoint at all. Similarly, if a function has a corresponding lower adjoint, then that lower adjoint is unique, but the function may not have a lower adjoint at all.

The functions of a Galois connection have many nice properties. One of these pertains to how the functions distribute over disjunction and conjunction. Every lower adjoint is *universally disjunctive* and every upper adjoint is

universally conjunctive. For our functions $\mathcal{SP}$ and $\mathcal{WLP}$, this means:

$$\mathcal{SP}[S, P_0 \ || \ P_1 \ || \ P_2 \ || \ \dots] = \\ \mathcal{SP}[S, P_0] \ || \ \mathcal{SP}[S, P_1] \ || \ \mathcal{SP}[S, P_2] \ || \ \dots$$

and

$$\mathcal{WLP}[S, Q_0 \ \&\& \ Q_1 \ \&\& \ Q_2 \ \&\& \ \dots] = \\ \mathcal{WLP}[S, Q_0] \ \&\& \ \mathcal{WLP}[S, Q_1] \ \&\& \ \mathcal{WLP}[S, Q_2] \ \&\& \ \dots$$

where I have used the notations $P_0, P_1, P_2, \dots$ and $Q_0, Q_1, Q_2, \dots$ to refer to any collections of predicates. I really mean *any*. The collection can be infinite or finite. Indeed, it can be an empty collection, whose disjunction is **false** and whose conjunction is **true**. So, the following holds, for any statement S :

$$\mathcal{SP}[S, \text{false}] = \text{false} \\ \mathcal{WLP}[S, \text{true}] = \text{true}$$

2.9.2. Strongest conservative postconditions? No such thing!

Is there a semantic function that we can pair with $\mathcal{WP}$ to form a Galois connection? If so, $\mathcal{WP}[S, \_]$ would be an upper adjoint and would satisfy:

$$\mathcal{WP}[S, \text{true}] = \text{true}$$

But this does not hold for a statement like **assert false**. Thus, we conclude that there is no version of $\mathcal{SP}$ that can be paired up with $\mathcal{WP}$.

Exercise 2.35.

For any assignment statement, **assert** statement, sequential composition, or conditional statement S , define a semantic function

$$\mathcal{NOCRASH}[S, P]$$

that gives the condition that needs to be checked to ensure that no execution of s from a state satisfying P crashes (that is, all executions of s from P are crash free). In other words, $NOCRASH[s, P]$ should give a predicate that is equivalent to $P \implies \mathcal{WP}[s, \text{true}]$. But instead of defining the semantic function backwards, as is done for $\mathcal{WP}$, define $NOCRASH[s, P]$ in the forward direction.

2.10. Method Calls with Preconditions

Back to method calls. In Section 2.7, I discussed the semantics of a call to a method with parameters and a postcondition. Using the **assert** statement from Section 2.8, we can now extend the method-call treatment to include a precondition, which induces a proof obligation for the caller.

As a general template for what a method specification looks like, consider

```
method M(x: X) returns (y: Y)
  requires P
  ensures R
```

To define the semantics of a call

```
t := M(E)
```

we first find fresh names x' and y' for the parameters. Then, the semantics of the call is the same as the semantics for the following program snippet:

```
var x', y';
x' := E;
assert P[x := x'];
assume R[x, y := x', y'];
t := y'
```

In words, we assign the actual in-parameters E to the freshly named formal in-parameters x' , then we assert the precondition of the call (that is, we check that the

precondition holds at the call site), and then, assuming that the freshly named formal out-parameters satisfy the method's postcondition, we assign those to the actual out-parameters of the call.

From this, we can compute the weakest precondition of a call:

$$\begin{aligned}
& \mathcal{WP}[\text{t} := \text{M}(\text{E}), \text{Q}] \\
= & \quad \{ \text{semantics of call} \} \\
& \mathcal{WP}[\text{var } x', y'; x' := \text{E}; \text{assert } P[x := x']; \\
& \quad \text{assume } R[x, y := x', y']; \text{t} := y', \text{Q}] \\
= & \quad \{ \text{definition of } \mathcal{WP} \text{ for } ; \text{ and } := \} \\
& \mathcal{WP}[\text{var } x', y'; x' := \text{E}; \text{assert } P[x := x']; \\
& \quad \text{assume } R[x, y := x', y']; \text{Q}[\text{t} := y']] \\
= & \quad \{ \text{definition of } \mathcal{WP} \text{ for } ; \text{ and } \text{assume} \} \\
& \mathcal{WP}[\text{var } x', y'; x' := \text{E}; \text{assert } P[x := x'], \\
& \quad R[x, y := x', y'] \implies \text{Q}[\text{t} := y']] \\
= & \quad \{ \text{definition of } \mathcal{WP} \text{ for } ; \text{ and } \text{assert} \} \\
& \mathcal{WP}[\text{var } x', y'; x' := \text{E}, \\
& \quad P[x := x'] \ \&\& \ (R[x, y := x', y'] \implies \text{Q}[\text{t} := y'])] \\
= & \quad \{ \text{definition of } \mathcal{WP} \text{ for } ; \text{ and } := \} \\
& \mathcal{WP}[\text{var } x', y', (P[x := x'] \ \&\& \ (R[x, y := x', y'] \implies \text{Q}[\text{t} := \\
& \text{y'}]))[x' := \text{E}]] \\
= & \quad \{ \text{definition of } \mathcal{WP} \text{ for var} \} \\
& \text{forall } x', y' :: (P[x := x'] \ \&\& \ (R[x, y := x', y'] \implies \text{Q}[\text{t} := \\
& \text{y'}]))[x' := \text{E}] \\
= & \quad \{ \text{apply the substitution for } x' \} \\
& \text{forall } x', y' :: P[x := \text{E}] \ \&\& \ (R[x, y := \text{E}, y'] \implies \text{Q}[\text{t} := y']) \\
= & \quad \{ x' \text{ is not used} \} \\
& \text{forall } y' :: P[x := \text{E}] \ \&\& \ (R[x, y := \text{E}, y'] \implies \text{Q}[\text{t} := y'])
\end{aligned}$$

As we had it in Section 2.7.2, the variable y' is universally quantified in the last line, because y' stands for any values of the formal out-parameters that satisfy the postcondition. So, our final $\mathcal{WP}$ equation for a call, considering both the method's precondition and its postcondition, is:

$$\begin{aligned}
& \mathcal{WP}[\text{t} := \text{M}(\text{E}), \text{Q}] = \\
& \quad P[x := \text{E}] \ \&\& \ \text{forall } y' :: R[x, y := \text{E}, y'] \implies \text{Q}[\text{t} := y']
\end{aligned}$$

|

Exercise 2.36.

If x' is fresh, then prove that

$x' := E; \text{ assert } P[x := x']$

is the same as

$\text{ assert } P[x := E]$

by showing that the weakest preconditions of these two statements are the same.

2.11. Function Calls

We reason about a method call in terms of the specification of the method, because methods are opaque. Functions, on the other hand, are transparent, so we reason about them by simply unfolding the definition of the function.

For example, consider a function for computing the absolute value of a number:

```
function Abs(x: int): int {  
  if x < 0 then -x else x  
}
```

Since `Abs` is a function (not a method), its body is an expression (see Section 1.5). Here, I'm using an *if-then-else expression*, which is an expression form of the *if statement*. The **then** and **else** branches of this expression are themselves expressions (whereas the branches of an *if statement* are statements). Using function `Abs`, suppose we want to prove, at some program point and for some expression E , that $0 \leq \text{Abs}(E)$ always holds. We can do that by writing

```
assert 0 <= Abs(E)
```

at that program point and proving that this statement does not crash. To prove that the statement is crash free and establishes a postcondition Q , we proceed as follows:

$$\mathcal{WP}[\text{assert } 0 \leq \text{Abs}(E), Q]$$

```

=      {  $\mathcal{WP}$  of assert }
  0 <= Abs(E) && Q
=      { def. Abs }
  0 <= (if E < 0 then -E else E) && Q
=      { distribute <= over if-then-else }
  (if E < 0 then 0 <= -E else 0 <= E) && Q
=      { arithmetic }
  true && Q
=
  Q

```

Exercise 2.37.

Compute $\mathcal{WP} \llbracket m := \text{Max}(x, y), m \leq 100 \rrbracket$ where function `Max` is as defined in Exercise 2.32. Simplify your answer.

Exercise 2.38.

Compute $\mathcal{WP} \llbracket y := \text{Plus3}(x); x := \text{Plus3}(y), x < 10 \rrbracket$, given

```

function Plus3(x: int): int {
  x + 3
}

```

Simplify your answer.

2.12. Partial Expressions

An expression may not always be defined. For example, the division c / d makes sense only if d is non-0. We say that such an expression is *partial*. When an expression is defined in the context where it is used, we describe it using words like *well-formed* or *well-defined*, and if it is always defined, we say that it is *total*.

When we reason about statements, there is a proof obligation that all expressions in the statements be well-defined. Formally, such proof obligations are expressed by $\mathcal{WP}$. This looks something like this:

$$\mathcal{WP} \llbracket x := E, Q \rrbracket = \mathcal{DEFINED} \llbracket E \rrbracket \ \&\& \ Q[x := E]$$

where $\mathcal{DEFINED} \llbracket E \rrbracket$ denotes an expression that (itself is total and) evaluates to **true** precisely when E is well-defined.

For example,

$\mathcal{WP}[x := c / d, x == 100]$

is

$d \neq 0 \ \&\& \ c / d == 100$

I'm not going to give the definition of $\mathcal{DEFINED}$ for all the expressions used in this book, nor am I going to redo the $\mathcal{WP}$ equations we've seen so far to incorporate $\mathcal{DEFINED}$ for every subexpression. Instead, I will typically just speak of there being some proof obligation. For example, I will say that the statement $x := c / d$ gives rise to the proof obligation $d \neq 0$.

Rather than incorporating $\mathcal{DEFINED}$ into the definition of $\mathcal{WP}$, we can think of every statement as starting with an implicit **assert** that contains the necessary proof obligations of the statement. If you always apply $\mathcal{WP}$ to those implicit assertions, then you will get the desired predicates using $\mathcal{WP}$ as I've defined it in this chapter.

Let me show you how this will go. When I say “when you write c / d , there is a proof obligation that d be non-0”, what I'm saying is that the implicit assertion that precedes $x := c / d$ is

assert $d \neq 0$

And when I point out that a program snippet like

```
if c / d < u / v {  
    x := a[i];  
}
```

comes with the proof obligations that d and v be non-0 and that the index i be a proper index into the array a , what I'm really saying is that the program has the following implicit assertions:

```
assert d != 0 && v != 0;  
if c / d < u / v {  
    assert 0 <= i < a.Length; // this says that i is in range  
    x := a[i];  
}
```

In effect, this transformation uniformly turns all well-definedness proof obligations into assertions. So, the transformed program will crash when the original program would have divided by zero, indexed an array outside its bounds, or engaged in any other undefined behavior.

Not all partial expressions are built-in operations. User-defined functions can have preconditions. For example, consider this function, `MinusOne`:

```
function MinusOne(x: int): int
  requires 0 < x
```

Calls to `MinusOne` come with the proof obligation that the argument be positive. Thus, a statement like `z := MinusOne(y)` has the following implicit assertion:

```
assert 0 < y
```

Exercise 2.39.

What is $\mathcal{DEFINED}$ of the following expressions?

- a) `x / (y + z)`
- b) `a / b < c / d`
- c) `a[2 * i]`
- d) `MinusOne(MinusOne(y))`

The programming-language operators `&&` and `||`, and `==>` are called *short-circuit operators*, because their well-definedness is sensitive to the value of the left argument. We have

$$\begin{aligned} \mathcal{DEFINED}\llbracket E \ \&\& \ F \rrbracket &= \mathcal{DEFINED}\llbracket E \rrbracket \ \&\& \ (E \implies \mathcal{DEFINED}\llbracket F \rrbracket) \\ \mathcal{DEFINED}\llbracket E \ || \ F \rrbracket &= \mathcal{DEFINED}\llbracket E \rrbracket \ \&\& \ (E \ || \ \mathcal{DEFINED}\llbracket F \rrbracket) \\ \mathcal{DEFINED}\llbracket E \implies F \rrbracket &= \mathcal{DEFINED}\llbracket E \rrbracket \ \&\& \ (E \implies \mathcal{DEFINED}\llbracket F \rrbracket) \end{aligned}$$

As these definitions show, the well-definedness of each of the three expressions depends on the well-definedness of `F`

only if E (is well-defined and) evaluates to **true**, **false**, and **true**, respectively. In a similar way, the well-definedness of an **if-then-else** expression depends on the value of the guard:

$$\begin{aligned} \mathcal{DEFINED}[\text{if } B \text{ then } E \text{ else } F] = \\ \mathcal{DEFINED}[B] \ \&\& \ \text{if } B \text{ then } \mathcal{DEFINED}[E] \text{ else } \\ \mathcal{DEFINED}[F] \end{aligned}$$

Exercise 2.40.

What is $\mathcal{DEFINED}$ of the following expressions?

- a) $p \ \&\& \ c / d == 100$
- b) $a / b < 10 \ || \ c / d < 100$
- c) **if** $a == b$ **then** c / d **else** 10
- d) $\text{MinusOne}(y) == 8 \implies a[y] == 20$

Exercise 2.41.

Use an **if-then-else** expression that is equivalent (both in value and well-definedness) to (a) $E \ \&\& \ F$, (b) $E \ || \ F$, and (c) $E \implies F$.

Exercise 2.42.

Revisit Exercise 2.35, but this time, define *NOCRASH* to also consider the partial expressions for the expressions occurring in the program text.

2.13. Method Correctness

To prove that a method implementation is correct with respect to its specification, we need to show that the given precondition implies the weakest precondition of the method body with respect to the postcondition.

In symbols, to prove that the following method is correct:

```
method M(x: X) returns (y: Y)
  requires P
  ensures Q
```

```
{  
  Body  
}
```

we need to prove

$P \implies \mathcal{WP}[\text{Body}, Q]$

2.14. Summary

In this chapter, I defined the semantics of the most common statements we will encounter. In summary, Floyd logic comes down to writing formulas that characterize the program state and verifying that a program “takes” predicates to other predicates. We often use Hoare triples to notate the claim that a program statement takes a given pre-state predicate to a given post-state predicate. The weakest precondition of a statement gives a systematic way to compute a necessary pre-state predicate from a desired post-state predicate. Weakest preconditions give a way to write canonical Hoare triples, by computing the first component of the Hoare triple as the $\mathcal{WP}$ of the other two components. Using $\mathcal{WP}$ computations, the resulting pre-state predicate includes the proof obligations of the program.

Many of the hairier discussions in this chapter surround the use of strongest post-conditions. I have included these discussions because strongest postconditions are computed in the direction of program flow, which makes them seem more obvious at first. Also, it would be irresponsible of me to omit strongest postconditions in a textbook chapter on program semantics. However, as we have seen, weakest preconditions are not just slightly easier to compute, but they also collect proof obligations as you go along. You wouldn't be able to prove that your program really establishes what you want if these proof obligations are not met. Still, it is good to explicitly check proof obligations,

because it lets us pin responsibility on the use of various program statements, like making sure our programs don't divide by zero or call a method without establishing its declared precondition.

With the understanding of program semantics from this chapter, I encourage you to page through the informal arguments of Chapter 1 again. You will feel a sense of mastery of the material when you're able to back up the informal arguments in Chapter 1 with formal arguments in your head and on paper. You will then also appreciate the work automated program verifiers do to compute $\mathcal{WP}$ or $\mathcal{SP} + \mathcal{NOCRAH}$ behind the scenes for you.

Notes

The general technique in Floyd's logic targeted a flowchart language, which makes it applicable to programs with any kind of control flow [50]. Hoare's presentation of Floyd logic not only used Hoare's elegant triples but also restricted its attention to structured program statements [63]. Such a restriction more effectively suggests compositional design rules for using the constructs in a programming language. For example, Hoare's formulation emphasized the role of an *invariant* when reasoning about a loop. (We'll see loop invariants throughout Part 2.)

For a comprehensive comparison of early approaches to program proofs, see a survey paper on program logics [58].

In his seminal 1976 book, *A Discipline of Programming*, Edsger W. Dijkstra gave five *healthiness conditions* that statements of every reasonable programming language should satisfy [40]. One of these, dubbed the Law of the Excluded Miracle, says that every statement s should satisfy

$$\mathcal{WP}[\![s, \text{false}]\!] = \text{false}$$

That is, under no circumstance should a statement `s` guarantee establishing the post-condition **false**.²

In the 1980s, Back [7], Morgan [92], Morris [96], and Nelson [99] all argued that it is useful to allow constructs like **assume** statements in specifications and in the study of program semantics. Note that $\mathcal{WP}[\text{assume false, false}]$ is **true**, which according to Dijkstra's terminology makes the statement miraculous. Nelson called such statements *partial commands* and used the partiality in the definition of **if** statements and loops. Morgan defined a general *specification statement* that can be partial. Morgan called the partial form *coercions* and showed that coercions can be combined to make compilable programs [95].

Back and von Wright built a beautiful lattice-theoretic framework of programs and specifications, where **assert** and **assume** statements are each other's duals [8, 9]. One way they view these statements is as “moves” in a game between two “agents”, often referred to as a “demon” and an “angel”.

Defying Dijkstra, Burrows and Nelson incorporated **assume** statements into a useful programming language for string processing [101]. Playfully, they called it LIM—Language of the *Included* Miracle.

¹In Hoare's seminal paper where he introduced these triples, crash-freedom and termination were not considered [63]. What I'm using here is a common variation of the triples that does consider these “total correctness” concerns (cf. [3]).

Chapter 3

Recursion and Termination



Your friend Amelia wants to borrow your phone. You agree, so long as Amelia promises to get it back to you. Then, Benny comes along and asks Amelia to borrow your phone. Amelia gives Benny the phone, but makes Benny promise to give it back to you. While Benny still has the phone, Chantal asks to borrow it, and then Dominykas, then Edsger, then Felistas, etc. If this goes on forever, you'll never get your phone back. Yet, it seems that each of your friends lives up to the promise to give the phone back by making an agreement with someone else to take it back. If we allow endless repetition, even of some behavior that seems justifiable by itself, then we don't achieve the results we're hoping for.

This chapter explains how to prevent endless repetitions.

3.0. The Endless Problem

Here are five little programs that demonstrate the need for considering termination.

First up is a method that claims to return a value equal to twice its argument:

```
method BadDouble(x: int) returns (d: int)  
  ensures d == 2 * x  
{  
  var y := BadDouble(x - 1);  
  d := y + 2;  
}
```

Sure, if the recursive call sets y to twice $x - 1$, then $y + 2$ will be twice x . But each recursive call will result in yet another recursive call, so, akin to the activities of your phone-borrowing friends, this method will not terminate. This is called *infinite recursion*. We want to avoid infinite recursion, because it does not seem meaningful to say such a method correctly computes $2 * x$.

As a second example, here is a method that claims to return the value of its argument. What could be simpler?

```
method PartialId(x: int) returns (y: int)  
  ensures y == x  
{  
  if x % 2 == 0 {  
    y := x;  
  } else {  
    y := PartialId(x);  
  }  
}
```

If the given x is even, then this method correctly returns x . But if x is odd, then the method proceeds by calling a method (namely, itself) that claims to return its argument. What we observe here is that *if* `PartialId` terminates, then it gives the right result. This is called *partial correctness*,

which means that every *terminating* call is correct, but it does not say that all calls do indeed terminate.

BadDouble above also adheres to partial correctness. In fact, since BadDouble *never* terminates, it would satisfy partial correctness also if you changed its assignment `d := y + 2` to `d := y + 5`, or, for that matter, to `d := 1000`.

A program is said to be *totally correct* if it is partially correct *and* always terminates. Throughout this book, we will prove total correctness. We have seen these concepts before. If you view non-termination as a form of crashing, then the predicate $\mathcal{WLP}[s, q]$ from Section 2.9.0 says that `s` is partially correct with respect to `q`. Furthermore, $\mathcal{WP}[s, \text{true}]$ says that `s` terminates without crashing. So, the conjunction of these two shows total correctness, which is captured in the equation we saw in Section 2.9.0:

$$\mathcal{WP}[s, q] = \mathcal{WLP}[s, q] \ \&\& \ \mathcal{WP}[s, \text{true}]$$

As a third example, here is a nondeterministic method that claims to compute the actual square of a number, given a guess of the same:

```
method Squarish(x: int, guess: int) returns (y: int)
  ensures x * x == y
{
  if
  case guess == x * x => // good guess!
    y := guess;
  case true =>
    y := Squarish(x, guess - 1);
  case true =>
    y := Squarish(x, guess + 1);
}
```

The implementation is partially correct and it could happen that its aimless search finds the answer. However, there is no *guarantee* of termination. For correct programs, we want to provide that guarantee by giving a *proof* of termination.

The fourth example tries to establish something impossible. And it would have gotten away with it, too, if it weren't for these meddling kids—uh, I mean, if it weren't for having to prove termination.

```
method Impossible(x: int) returns (y: int)  
  ensures y % 2 == 0 && y == 10 * x - 3  
{  
  y := Impossible(x);  
}
```

As a final example, consider this dubious function definition:

```
function Dubious(): int {  
  1 + Dubious()  
}
```

Here, we're seeing not just a case of infinite recursion, but also a lurking mathematical inconsistency. If we admit a definition like this, then you could easily prove **false**, as is shown by the reasoning suggested by the assertions in this program snippet:

```
var a := Dubious();  
assert a == Dubious(); // this is what the previous  
                        // assignment establishes  
// by expanding Dubious() to its body, the following  
// assertion says the same thing as the previous assertion  
assert a == 1 + Dubious();  
// combine the previous two assertions  
assert a == 1 + a;  
// subtract a from both sides  
assert 0 == 1;  
// arithmetic  
assert false;
```

It should not be possible to prove **false**—that would be a sign of a mathematical *inconsistency*. We conclude from this example that we should not admit functions like Dubious() that are mathematically inconsistent. The most common way to avoid inconsistently defined functions is to prove that the functions terminate.

In summary, we want to avoid infinite recursion and other forms of infinite repetition, avoid non-guaranteed termination, and avoid mathematical inconsistencies. We want to go beyond partial correctness, proving freedom from crashes and proving termination to achieve total correctness. Next, let's see how we do that.

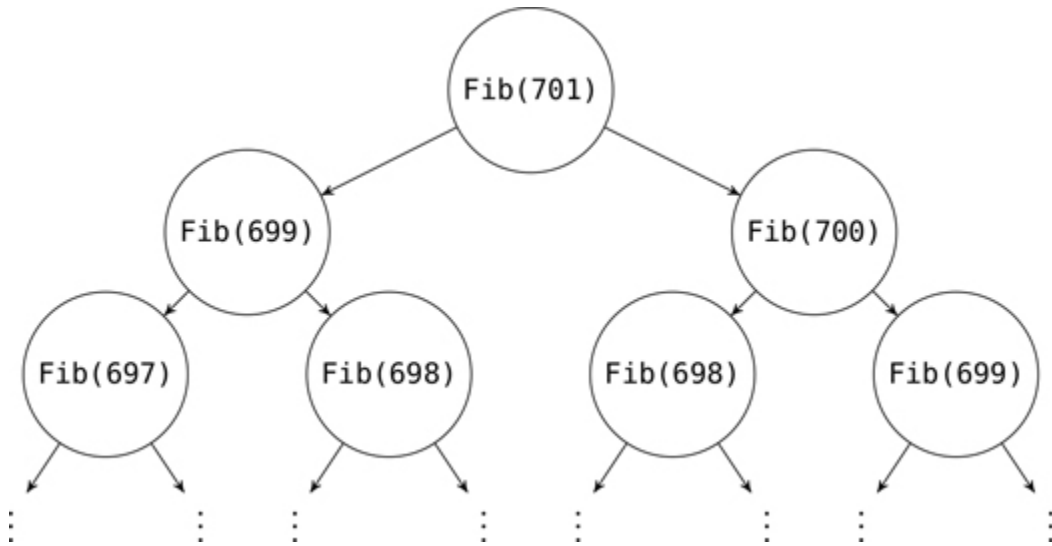
3.1. Avoiding Infinite Recursion

As an example, consider the well-known Fibonacci function:

```
function Fib(n: nat): nat {  
  if n < 2 then n else Fib(n - 2) + Fib(n - 1)  
}
```

In this example, I used a new type, **nat**. It denotes the natural numbers, that is, 0, 1, 2, . . . —the non-negative subset of the integers. (An alternative way to define `Fib` on the natural numbers would be to define `n` of type **int** and to add a precondition **requires** `0 <= n`. These are in effect the same, but using the type **nat** is shorter.)

Function `Fib` has two recursive calls, `Fib(n-2)` and `Fib(n-1)`. If `Fib` calls itself, how do we convince ourselves that the recursion will eventually end? Imagine a graph of the activation records that result from evaluating `Fib`. For example, here is a small portion of that graph starting with an activation record for `Fib(701)`:



Now, imagine decorating each node (that is, each activation record), with some natural number. A convenient way to pick such a natural number for each of these nodes is to pick the same number as the argument to the `Fib` function. If every node in the graph has the property that its decoration is strictly smaller than the decoration of the parent node, then every path in the graph is finite. Why? Because there is no infinite descending sequence of natural numbers. (The common phrase for this, which I'll use from now on, is infinite descending *chain*.)

This gives us a way to verify that a function terminates. To pull this off, all we need is a mechanism to prescribe a decoration for each activation record, and then we need to verify that the decoration of the callee is strictly smaller than the decoration of the caller. A decoration like this is called a *termination metric*.

In Dafny, the way to prescribe a termination metric for a function or method is to use a **decreases** clause. For `Fib`, we write:

```
function Fib(n: nat): nat
  decreases n
{
  if n < 2 then n else Fib(n - 2) + Fib(n - 1)
}
```

The expression that follows the **decreases** keyword (here, the expression n) will be evaluated and used as the decoration for the activation record of `Fib( $n$ )`. The decorations for the activation records that result from the two recursive calls are therefore $n-2$ and $n-1$, respectively. Dafny generates and checks the proof obligation $n-2 < n$ for the first call and $n-1 < n$ for the second call. These proof obligations imply that `Fib`'s recursion terminates.

In many cases, the **decreases** clause gives rise to an intuitive reading: “with each recursive call of `Fib`, the argument n decreases”. However, note that the keyword **decreases** just says how to decorate an activation record. Once the activation record gets decorated, according to the **decreases** clause evaluated on entry to the function or method, the decoration does not change. It is the *relation* between the caller's decoration and the callee's decoration that must show a decrease.

There are many other ways to decorate the activation records of `Fib` to prove termination. For example, we could have used

```
decreases 3*n + 2
```

in which case the termination proof obligation for the first recursive call would be $3*(n-1) + 2 < 3*n + 2$.

Let's consider another example. Here is a function that computes the sum of the elements of an integer sequence from index `lo` to `hi`:

```
function SeqSum(s: seq<int>, lo: int, hi: int): int
  requires 0 <= lo <= hi <= |s|
  decreases hi - lo
{
  if lo == hi then 0 else s[lo] + SeqSum(s, lo + 1, hi)
}
```

This example gives a taste of Dafny's **seq** type, which denotes immutable sequences of elements. The length of a sequence `s` is given by `|s|` and the element at index `i`,

where $0 \leq i < |s|$, is retrieved by the expression $s[i]$. Later chapters will make more use of sequences.

The `SeqSum` function needs a slightly more complicated way of computing its termination metric. The given **decreases** clause does indeed give an expression that decreases with each recursive call. In particular, the recursive call has the proof obligation $hi - (lo+1) < hi - lo$.

How about instead choosing just $-lo$ as the way to compute termination metrics, since, after all, $-(lo+1) < -lo$? The problem with such a termination measure is that if we don't stay within the non-negative subset of the integers, then there *are* infinite decreasing chains, so we cannot conclude that the recursion terminates. At a recursive call like the one in `SeqSum`, Dafny checks that the termination metric of the caller is non-negative. If you try **decreases** $-lo$, you will therefore get an error message that Dafny cannot verify termination.

Exercise 3.0.

Write a **decreases** clause that proves the termination of the following function:

```
function F(x: int): int {
  if x < 10 then x else F(x - 1)
}
```

Exercise 3.1.

Write a **decreases** clause that proves the termination of the following function:

```
function G(x: int): int {
  if 0 <= x then G(x - 2) else x
}
```

Exercise 3.2.

Write a **decreases** clause that proves the termination of the following function:

```
function H(x: int): int {
  if x < -60 then x else H(x - 1)
}
```



```
}
```

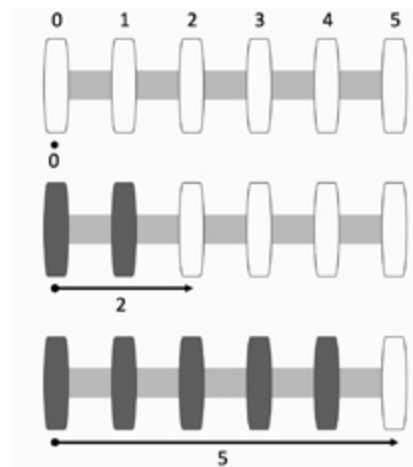
Exercise 3.3.

Write a **decreases** clause that proves the termination of the following function:

```
function I(x: nat, y: nat): int {  
  if x == 0 || y == 0 then  
    12  
  else if x % 2 == y % 2 then  
    I(x - 1, y)  
  else  
    I(x, y - 1)  
}
```

Sidebar 3.0

If you have 5 steps to make, there are 6 places along your journey: at any time, you may have taken 0, 1, 2, 3, 4, or all 5 steps. These diagrams depict your progress after 0 steps, 2 steps, and 5 steps, respectively:



If n denotes the steps you've taken, we have

$$0 \leq n \leq 5$$

That is, n can be as small as 0 and as large as 5. Notice the inclusive upper and lower bounds (that is, $\leq$ as opposed to $<$). That's the kind of interval you get when you

express how much has been done, that is, your progress so far.

More generally, if there are N steps along your journey, there are $N + 1$ different measures of progress.

A journey that starts at l_0 and ends at h_i (as in the interval of sequence elements to be added by function `SeqSum`) has $h_i - l_0$ steps. There are $h_i + 1 - l_0$ different measures of progress along the way. For example, during the evaluation of `SeqSum(s, 0, 5)`, we're summing $5 - 0$ elements and we will evaluate the body of `SeqSum` a total of 6 times.

Exercise 3.4.

Write a **decreases** clause that proves the termination of the following function:

```
function J(x: nat, y: nat): int {  
  if x == 0 then  
    y  
  else if y == 0 then  
    J(x - 1, 3)  
  else  
    J(x, y - 1)  
}
```

Exercise 3.5.

Write a **decreases** clause that proves the termination of the following function:

```
function K(x: nat, y: nat, z: nat): int {  
  if x < 10 || y < 5 then  
    x + y  
  else if z == 0 then  
    K(x - 1, y, 5)  
  else  
    K(x, y - 1, z - 1)  
}
```

Exercise 3.6.

Write a **decreases** clause that proves the termination of the following function:

```
function L(x: int): int {  
  if x < 100 then L(x + 1) + 10 else x  
}
```

Exercise 3.7.

The following function computes the *Hofstadter G sequence*:

```
function G(n: nat): nat {  
  if n == 0 then 0 else n - G(G(n - 1))  
}
```

Find an appropriate **decreases** clause to prove that *G* terminates.

3.2. Well-Founded Relations

Termination metrics are not restricted to natural numbers. As long as the “less than” relation we use is *well-founded* (which I will define in a moment), the technique I discussed above lets you prove termination. I will use the notation $>$ for a well-founded order, where the smaller element goes on the right. That is, for this ordering, I'm switching to a “greater than” notation instead of a “less than” notation. This lets us think of going from the caller's termination metric (on the left) to the callee's termination metric (on the right). You can pronounce $a > b$ as “*a* exceeds *b*”, “*a* goes down to *b*”, or “*a* reduces to *b*”.

A binary relation $>$ is *well-founded* when the following three conditions hold:

- $>$ is *irreflexive*: The relation never relates an element to itself. In formulas, $a > a$ never holds for any a .

- $>$ is *transitive*: Whenever $a > b$ and $b > c$ hold, then so does $a > c$, for any a , b , and c .
- $>$ satisfies the *descending chain condition*: The relation has no *infinite descending chain*, that is, there is no infinite sequence of values $a_0, a_1, a_2, \dots$ such that

$$a_0 > a_1 > a_2 > \dots$$

A relation that satisfies the first two conditions is called a *strict partial order* (or, sometimes, an *irreflexive partial order*). Being a *partial* order allows the possibility of not ordering all elements. For example, there may be elements a and b such that neither $a > b$ nor $b > a$ holds. (If either $a > b$ or $b > a$ does hold for every pair of distinct values a and b , then the strict order is called *total*.) So, a well-founded relation is a strict partial order that additionally satisfies the descending chain condition.^{[2](#)}

Generalizing from the ordering on natural numbers to a well-founded order $>$ on the values of all types in our program, we have the following recipe for proving that there is no infinite recursion: Decorate every activation record with a value (of any type) so that, for every call, the caller's decoration a and callee's decorations b satisfy $a > b$.

We are going to use one, fixed well-founded relation $>$ throughout this book, namely the one that Dafny uses. Dafny pre-defines $>$ for each type, and in some cases also for values between different types (but in most cases, values from different types are not related in this partial order).

Here are the built-in well-founded relations for some of Dafny's types:

type of x and y	$x \succ y$ (" x reduces to y ")
bool	$x \ \&\& \ !y$
int	$y < x \ \&\& \ 0 \leq x$
real	$y \leq x - 1.0 \ \&\& \ 0.0 \leq x$
set<T>	y is a proper subset of x
seq<T>	y is a consecutive proper sub-sequence of x
inductive datatypes	y is structurally included in x

Some remarks are in order. For booleans, the ordering says that the only way to decrease is from **true** to **false**. Dafny's well-founded relation on integers orders nonnegative numbers above negative numbers, but the negative numbers themselves are not ordered. The relations on integers and reals are really the same, if you think of $y < x$ for integers as $y \leq x - 1$. Note that $y < x \ \&\& \ 0.0 \leq x$ would not be a well-founded order on the reals, because this order has infinite descending chains. We'll talk about inductive datatypes in Chapter 4 (and Section 4.3 talks about structural inclusion).

Exercise 3.8.

Write a **decreases** clause that proves the termination of the following function:

```
function M(x: int, b: bool): int {
  if b then x else M(x + 25, true)
}
```

Exercise 3.9.

Write a **decreases** clause that proves the termination of the following function:

```
function N(x: int, y: int, b: bool): int {
  if x <= 0 || y <= 0 then
    x + y
  else if b then
    N(x, y + 3, !b)
```

```
    else  
      N(x - 1, y, true)  
  }
```

Exercise 3.10.

An answer to Exercise 3.1 is **decreases** x . (a) Write the termination proof obligation using $>$. (b) Expand this $>$ for type `int`.

3.3. Lexicographic Tuples

Sometimes when proving termination, it is necessary or convenient to use a *lexicographic tuple*. A tuple is a list of values, like a pair or a triple. The *lexicographic order* on tuples is a component-wise comparison where earlier components are treated as more significant. For example, for pairs of natural numbers, we have that 4,12 exceeds 4,11 and 4,2 and 3,525600 and 2,0. The reason that 4,12 exceeds 4,2 is that the first components are equal and, for the second component, 12 goes down to 2. The reason that 4,12 exceeds 3,525600 is that 4 goes down to 3 in the first component, so the second component does not matter.

We can even compare tuples of different lengths. For example, we can compare a pair with a quadruple or compare a 16-tuple with a triple. In these cases, if one is a proper prefix of the other, then the *shorter* tuple exceeds the other; otherwise, comparing as many components as they have in common gives you the answer. For example, 4,6,0 exceeds 4,6,0,25,3, because 4,6,0 is a proper prefix of 4,6,0,25,3, and 2,5 exceeds 1, because the prefix 2 exceeds the prefix 1.

If we impose some upper bound on the length of tuples, then the well-foundedness of the lexicographic order defined above follows from the well-foundedness of the component-wise orders.

To allow lexicographic tuples in proofs of termination, Dafny allows each **decreases** clause to give a *list* of expressions. Although there is no limit on how many expressions a **decreases** clause can list, any given program has some maximum-length **decreases** clause. Therefore, the lexicographic order we use is indeed well-founded. I will write $>$ for the lexicographic order on tuples as well as for the order on each component.

Exercise 3.11.

Determine if the first tuple exceeds the second.

- a) 2,5 and 1,7
- b) 1,7 and 7,1
- c) 5,0,8 and 4,93
- d) 4,9,3 and 4,93
- e) 4,93 and 4,9,3
- f) 3 and 2,9
- g) **true**,80 and **false**,66
- h) **true**,2 and 19,1
- i) 4,**true**,50 and 4,**false**,800
- j) 7.0,{3,4,9},**false**,10 and 7.0,{3,9},**true**,10

In (j), the expressions {3,4,9} and {3,9} denote sets.

Next, I will show some more examples that use lexicographic tuples.

3.3.0. Remaining school work

Suppose you have n courses left before graduation. When you start a new course c , a call to the method

method RequiredStudyTime(c : **nat**) **returns** ($hours$: **nat**)

tells you how many hours of studying is needed to complete the course. Exactly which material will be covered in the

course depends on the teacher, so there is nothing more precise we can say about the out-parameter hours.

The following method simulates your time until graduation. It says you have h hours left in course n , after which you also have to complete courses 0 to n .

```
method Study(n: nat, h: nat)
  decreases n, h
{
  if h != 0 {
    // first, study for an hour, and then:
    Study(n, h - 1);
  } else if n == 0 {
    // you just finished course 0 - woot woot, graduation time!
  } else {
    // find out how much studying is needed for the next course
    var hours := RequiredStudyTime(n - 1);
    // get started with course n-1:
    Study(n - 1, hours);
  }
}
```

To prove termination of `Study`, we need a lexicographic tuple. Let's see how the **decreases** n, h proves termination. There are two recursive calls. For the first one, we need to prove

$n, h > n, h-1$

and for the other, we need to prove

$n, h > n-1, \text{hours}$

Both of these conditions hold, so it follows that method `Study` terminates.

Exercise 3.12.

Suppose the university cracks down on professors to limit the required study time for a course to 200. We can then write a postcondition for method `RequiredStudyTime`:

```
method RequiredStudyTime(c: nat) returns (hours: nat)
  ensures hours <= 200
```


Knowing such a bound, you can still use the termination metric n, h for `Study`, but it's also possible to write a termination metric without using a lexicographic tuple. Do so.

3.3.1. Ackermann

The next example is the famous Ackermann function. It runs for a long time, but it does terminate. Proving termination is easy if we use a lexicographic pair.

```
function Ack(m: nat, n: nat): nat
  decreases m, n
{
  if m == 0 then
    n + 1
  else if n == 0 then
    Ack(m - 1, 1)
  else
    Ack(m - 1, Ack(m, n - 1))
}
```

To check termination, we check that the lexicographic pair m, n of natural numbers decreases for each recursive call. The recursive call in the second branch and the outer recursive call in the third branch use $m - 1$ as the first argument, thus decreasing the first component of the pair m, n . For the inner recursive call in the third branch, we have that m, n exceeds $m, n - 1$. Thus, we have proved termination.

Sidebar 3.1

To use a lexicographic tuple in a termination proof in Dafny, list the components of the tuple after the **decreases** keyword. Dafny also supports tuple types, but they are built-in inductive datatypes, so they are ordered by structural inclusion. So, whereas **decreases** E, F says to use the

lexicographic pair E, F as the termination metric, **decreases** (E, F) says to use the single value (E, F) .

We have that **true**, 5 > **false** in lexicographic order, because the first component **true** goes down to **false**. We also have $(\{2, 4, 6\}, \text{true}) > \{2, 4, 6\}$, because the set $\{2, 4, 6\}$ is structurally included in the pair $(\{2, 4, 6\}, \text{true})$. However, neither $(\text{true}, 5) > \text{false}$ nor $\{2, 4, 6\}, \text{true} > \{2, 4, 6\}$ holds.

3.3.2. Mutually recursive methods

One common pattern among mutually recursive methods is that one of the methods can be considered as an “outer” computation that calls an “inner” computation. A variation of the study example from Section 3.3.0 shows this pattern. Here, I'm using n as the number of courses *completed*, out of 40, and h as the number of hours remaining for the current class.

```
method StudyPlan(n: nat)
  requires n <= 40
  decreases 40 - n
{
  if n == 40 {
    // done
  } else {
    var hours := RequiredStudyTime(n);
    Learn(n, hours);
  }
}
```

```
method Learn(n: nat, h: nat)
  requires n < 40
  decreases 40 - n, h
{
  if h == 0 {
```

```

    // done with class n; continue with the rest of the study
plan
  StudyPlan(n + 1);
} else {
  // some learning to take place here...
  Learn(n, h - 1);
}
}

```

Method `StudyPlan` is the “outer” method and `Learn` is the “inner” method.

With the given **decreases** clauses, the proof obligations for termination are as follows:

call	proof obligation for establishing termination

StudyPlan calls Learn	$40 - n > 40 - n, h$

Learn calls StudyPlan	$40 - n, h > 40 - (n + 1)$

Learn calls Learn	$40 - n, h > 40 - n, h - 1$

For the first call, the proof obligation holds because $40 - n$ is a proper prefix of $40 - n, h$. For the second call, the proof obligation holds because the first component is decreased. For the third call, the first component stays the same and the second component decreases. So, the two methods terminate.

Exercise 3.13.

Add **decreases** clauses to prove termination of the following variation of the outer/inner `StudyPlan` program from above.

```

method Outer(a: nat) {
  if a != 0 {
    var b := RequiredStudyTime(a - 1);
    Inner(a, b);
  }
}

```

```

method Inner(a: nat, b: nat)
  requires 1 <= a
{
  if b == 0 {
    Outer(a - 1);
  } else {
    Inner(a, b - 1);
  }
}

```

Exercise 3.14.

Add **decreases** clauses to prove termination of the following variation of the outer-inner StudyPlan program from above.

```

method Outer(a: nat) {
  if a != 0 {
    var b := RequiredStudyTime(a - 1);
    Inner(a - 1, b);
  }
}

method Inner(a: nat, b: nat) {
  if b == 0 {
    Outer(a);
  } else {
    Inner(a, b - 1);
  }
}

```

3.3.3. Refactor a subcomputation

Here's an example that illustrates a useful trick with lexicographic tuples. Consider the following function, which computes $2^n - 1$:

```

function ExpLess1(n: nat): nat {
  if n == 0 then 0 else 2 * ExpLess1(n - 1) + 1
}

```

It is easy to see that `ExpLess1` terminates, because it decreases `n`. Now, suppose that we want to refactor the

computation of the **else** branch into a function `ExpLess2(n)` that computes $2^n - 2$:

```
function ExpLess1(n: nat): nat {  
  if n == 0 then 0 else ExpLess2(n) + 1 // what about  
                                     // termination of the call to ExpLess2?  
}  
function ExpLess2(n: nat): nat  
  requires 1 <= n  
{  
  2 * ExpLess1(n - 1)  
}
```

Clearly, this refactoring does not alter what `ExpLess1` does, so we expect that `ExpLess1` still terminates. But how do we *prove* termination of the mutually recursive `ExpLess1` and `ExpLess2`?

If we declare both functions with **decreases** `n`, we can prove termination of the call from `ExpLess2` to `ExpLess1`, but not of the call from `ExpLess1` to `ExpLess2`, since it passes the same value for `n`. If we instead try **decreases** `n` for `ExpLess1` and **decreases** `n-1` for `ExpLess2`, then we can prove termination of the call from `ExpLess1` to `ExpLess2`, but not of the call from `ExpLess2` to `ExpLess1`.

In a situation like this, when one function calls another with the same parameter values, we would like to order one of the functions “below” the other. We can do that by adding a second component to the lexicographic tuple. It doesn't matter what the values of the second component is, as long as the value is greater for `ExpLess1` than it is for `ExpLess2`. Here is the most straightforward way to do it:

```
function ExpLess1(n: nat): nat  
  decreases n, 1  
{  
  if n == 0 then 0 else ExpLess2(n) + 1  
}  
function ExpLess2(n: nat): nat  
  requires 1 <= n  
  decreases n, 0  
{
```

```

    2 * ExpLess1(n - 1)
}

```

When you read this specification out loud, it may feel counterintuitive to “decrease 1”, since 1 is a constant. But remember, the **decreases** clause is just a way to prescribe how to compute the decorations of the function's activation records. That is, the **decreases** clause really just says “here's how I want to compute the termination metric for this function”. Following our rules for checking termination, it is easy to see that these **decreases** clauses do indeed prove termination:

call	proof obligation for establishing termination
ExpLess1 calls ExpLess2	$n, 1 > n, 0$
ExpLess2 calls ExpLess1	$n, 0 > n - 1, 1$

In fact, because of the rule that a proper prefix of a tuple exceeds the tuple itself, we could also have chosen **decreases** n for ExpLess1 and **decreases** $n, 257$ for ExpLess2.

Exercise 3.15.

The following functions F and M compute *Hofstadter Female and Male sequences*.

```

function F(n: nat): nat {
  if n == 0 then 1 else n - M(F(n - 1))
}
function M(n: nat): nat {
  if n == 0 then 0 else n - F(M(n - 1))
}

```

Find **decreases** clauses that prove termination of F and M .

Exercise 3.16.

The StudyPlan example in Section 3.3.2 above benefited from the rule that a proper prefix exceeds a longer

lexicographic tuple. Write **decreases** clauses for `StudyPlan` and `Learn` that prove termination without making use of that rule. That is, your **decreases** clauses for `StudyPlan` and `Learn` will be equally long.

Exercise 3.17.

Find a **decreases** clause that proves the termination of the following function. (This one is tricky.)

```
function F(x: nat, y: nat): int {
  if 1000 <= x then
    x + y
  else if x % 2 == 0 then
    F(x + 2, y + 1)
  else if x < 6 then
    F(2 * y, y)
  else
    F(x - 4, y + 3)
}
```

3.4. Default decreases in Dafny

In Dafny, if a recursive function or method is not given an explicit **decreases** clause, the verifier makes one up. More or less, this default termination metric is the lexicographic tuple of the function or method's parameters, in the order given.

For example, here are the signatures of some of the examples in this chapter, showing which default **decreases** they would get if no **decreases** clause is supplied manually.

```
function Fib(n: nat): nat
  decreases n
function SeqSum(s: seq<int>, lo: int, hi: int): int
  decreases s, lo, hi
method Study(n: nat, h: nat)
  decreases n, h
function Ack(m: nat, n: nat): nat
  decreases m, n
method StudyPlan(n: nat)
  decreases n
```

```
method Outer(a: nat) // from Exercise 3.13
  decreases a
method Inner(a: nat, b: nat) // from Exercise 3.13
  decreases a, b
```

The default termination metric for SeqSum is not right for proving termination (it needs $hi - lo$), so a **decreases** clause has to be supplied manually. The same goes for StudyPlan, which needs the termination metric $40 - n$. But for the others, the default **decreases** clause is correct, so if you omitted the **decreases** clause, the verifier would prove termination automatically without prompting you to supply anything else.

I told you *more or less* what the default **decreases** clause is. The exact rules are slightly more involved. Luckily, those details are not important, because the Dafny IDEs tell you in hover text what gets chosen. So, rather than memorizing the exact rule that Dafny uses to arrive at a default **decreases** clause, you can simply inspect what gets used.

It is also good to remember that the default **decreases** clause is merely a simple guess. Nevertheless, this guess very often works in practice. In the rest of this book, I will omit **decreases** clauses when Dafny guesses correctly.

Exercise 3.18.

Try removing the explicit **decreases** clauses in the examples shown in this chapter. Which of them does Dafny guess correctly? Check the hover text to inspect what Dafny guesses.

3.5. Summary

The key to proving termination is to decorate recursive (and mutually recursive) functions and methods with termination metrics and to show that those metrics decrease as the program runs. We perform all comparisons of termination metrics using a fixed well-founded relation,

because the lack of infinite descending chains in a well-founded relation implies termination.

A function or method is given a termination metric by declaring a **decreases** clause. Such a clause takes a list of expressions, which are interpreted as a lexicographic tuple. In practice, the lexicographic tuples used in termination metrics frequently coincide with the function's or method's list of parameters. So, an effective convention for reducing clutter in the program text is to declare a **decreases** clause explicitly only when it cannot be guessed from the function or method signature. Dafny supports this convention by supplying default **decreases** clauses, and in the same way, I will from now on show **decreases** clauses in this book only when Dafny's defaults are not sufficient.

Notes

The Ackermann function was developed by Wilhelm Ackermann as an example of a total computable function that goes beyond what is known as *primitive* recursive.

It is the use of a well-founded order that makes our termination proofs work. Indeed, this was Floyd's approach to proving termination [50]. A classic paper compares several ideas for proving termination [70].

In the absence of a **decreases** clause, Dafny *guesses* one from the *signature* of a method or function. An alternative would be to try to *infer* a default **decreases** clause from the *implementation* of a method or function. The literature contains a large body of work for such inference (see, e.g., [21, 33]).

Some programs are designed to run forever or to run for an unbounded amount of time. A simple example is a chat program that tirelessly accepts more messages to be delivered, ending only if the user chooses to quit the app. Another example is an operating system that continues to

service its running processes. For situations where you don't want to prove termination of a method, Dafny allows you to declare the method with **decreases ***, which says “permit this method to not terminate”. The **decreases *** clause is allowed on methods (and loops), but not on functions.

Not all verification tools check total correctness. Some check only partial correctness, and some check a mix of partial and total correctness. For example, the verifier OpenJML [105] checks termination of loops, but not of recursive calls.

For functions, non-termination can give rise to a mathematical inconsistency, like the function `Dubious()` at the beginning of this chapter. Proving termination for functions ensures mathematical consistency, but there are ways to ensure mathematical consistency besides termination. For example, ACL2 [71] safely admits tail-recursive functions, whether or not they terminate. In this book, we will prove termination of all functions.

²Technically, a well-founded relation is a strict partial order where every nonempty subset has a minimal element. This is the same as satisfying the descending chain condition, *provided* we assume what is called the *Axiom of Dependent Choice*. I just find the descending chain condition easier to understand and use.

Chapter 4

Inductive Datatypes



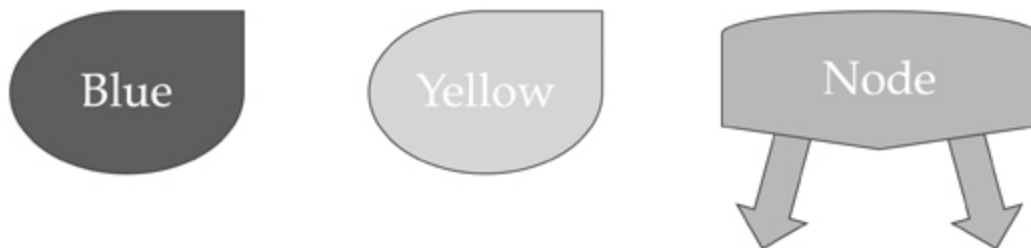
Programs have a need to define their own data structures. A useful and elegant way to do that is to use *datatypes*. These define what shapes, or *variants*, data can have and the values, or *payload*, that each variant carries. For example, a datatype representing coffee-shop products may have the variants `MatchaTea` and `Latte`, and the payload of `Latte` may be an integer describing a number of shots of espresso.

A datatype describes immutable *values* (not mutable *state*, as the classes we'll study in Chapters 16 and 17 do). This makes them well-suited not just in code, but also in specifications and proofs. Because of this mathematical form, datatypes are sometimes called *algebraic datatypes*.

Datatypes are often defined recursively. That is, values can be built up from simple variants or from variants that contain other datatypes. For this reason, they are known as *inductive datatypes*.

4.0. Blue-Yellow Trees

A datatype defines the possible structure of its values, divided into different *variants*. As an example, let us consider a binary tree structure where each leaf node is either blue or yellow. There are three variants of such blue-yellow trees: a blue leaf node is a blue-yellow tree, a yellow leaf node is a blue-yellow tree, and an interior node with two subtrees (that is, with two components that are themselves blue-yellow trees) is a blue-yellow tree. Here is a pictorial view of the three forms of blue-yellow trees:



In the program text, this datatype, call it `BYTree`, is declared as follows:

```
datatype BYTree = BlueLeaf | YellowLeaf | Node(BYTree, BYTree)
```

The right-hand side of the `=` describes *every possible variant* of a `BYTree` that can be constructed. This is an important property of a datatype. It means that if you have an unknown value of a datatype, you still know it is one of

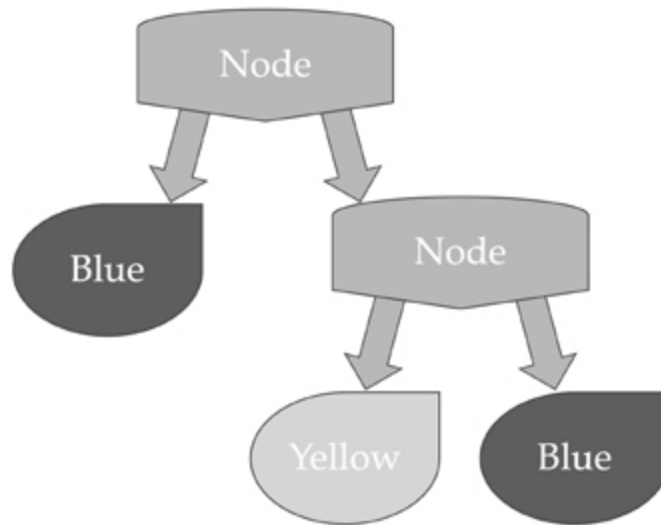
the variants declared in the type. Each variant has a named *constructor*. The constructors are separated by `|`'s and the order in which they are listed is immaterial.

In the example, `BlueLeaf` and `YellowLeaf` are parameter-less constructors, and `Node` is a constructor that takes two arguments. Dafny allows us to leave off the parentheses `()` for parameter-less constructors. In this example, the types of `Node`'s arguments are themselves `BYTree`, but in general, the types of constructor arguments are not limited to the type being defined.

The constructors are used as functions (or, in the parameter-less case, as constants), so the expression

```
Node(BlueLeaf, Node(YellowLeaf, BlueLeaf))
```

denotes the tree



4.1. Matching on Datatypes

When we define a function on a datatype, it is typical to determine the value of the function based on the variant of the given datatype value. This is most easily done using a `match` expression. To illustrate, consider a function that counts the number of blue nodes in a given blue-yellow tree:

```

function BlueCount(t: BYTree): nat {
  match t
  case BlueLeaf => 1
  case YellowLeaf => 0
  case Node(left, right) => BlueCount(left) + BlueCount(right)
}

```

A **match** expression takes a source expression (t in this case) followed by a number of **case** branches. Each **case** gives a *pattern*, and the **match** expression evaluates to the expression that follows the => of the **case** that the source expression matches. A pattern is a datatype constructor whose parameters (if any) are given by bound variables or by nested patterns. The bound variables (left and right in the Node case of the example) get bound to the values that had parameterized the Node when it was constructed. The scope of the bound variables is the body of the **case** in whose pattern they are declared.

A **match** expression must include a **case** for each possible value that the source expression can have when the **match** expression is reached. The cases are matched in order.

Exercise 4.0.

What happens if you delete the line **case** Yellow => 0 in function BlueCount?

If the bound variable in a pattern is not used in the corresponding **case** body, it can be given as `_`. For example, here is a function that computes the length of a tree's leftmost spine:

```

function LeftDepth(t: BYTree): nat {
  match t
  case BlueLeaf => 0
  case YellowLeaf => 0
  case Node(left, _) => 1 + LeftDepth(left)
}

```

Since this function does not need the second parameter of the Node, the Node pattern uses an underscore instead of naming the parameter.

Exercise 4.1.

Write a function `ReverseColors` that takes a blue-yellow tree and returns a tree just like it, except with blue nodes turned into yellow nodes and yellow nodes turned into blue nodes.

Exercise 4.2.

A program verifier is typically used to prove that a program behaves as expected for all inputs. But you can also use it to *test* that a function behaves as expected for some simple inputs. For example, the following method calls `ReverseColors` from Exercise 4.1 and uses **assert** statements to check that the function behaves as expected for two inputs:

```
method TestReverseColors() {  
  var a := Node(BlueLeaf, Node(BlueLeaf, YellowLeaf));  
  var b := Node(YellowLeaf, Node(YellowLeaf, BlueLeaf));  
  assert ReverseColors(a) == b;  
  assert ReverseColors(b) == a;  
}
```

We'll see more test harnesses like this later in the book. Write three more simple test cases for `ReverseColors`. Also, write a “negative test”—a test that you expect to fail.

Exercise 4.3.

Write a function `Oceanize` that takes a blue-yellow tree and returns a tree just like it, except with all leaf nodes turned into blue leaf nodes.

4.2. Discriminators and Destructors

Sometimes, we want to know the variant of a datatype value. That is, we want to know which constructor was used to create the value. We can write a function to determine this. Here is a function `IsNode` that returns **true** if

the given datatype was constructed using the `Node` constructor:

```
predicate IsNode(t: BYTree) {  
  match t  
  case BlueLeaf => false  
  case YellowLeaf => false  
  case Node(_, _) => true  
}
```

(Remember that a **predicate** is simply a **function** with result type **bool**.)

Since `IsNode` is a common operation, Dafny pre-defines such a *discriminator* for each constructor. The name of the discriminator is the name of the constructor followed by a question mark, so the names of the discriminators for type `BYTree` are `BlueLeaf?`, `YellowLeaf?`, and `Node?`. The discriminators are *members* of the type, which means they are accessed by following an expression with a dot and the name of the member. So, whereas the predicate `IsNode` we defined above is invoked as `IsNode(t)` for an expression `t` of type `BYTree`, a discriminator is invoked as `t.Node?`. Note, for a parameter-less constructor like `BlueLeaf`, the expression `t.BlueLeaf?` is equivalent to `t == BlueLeaf`.

Another useful operation on a datatype is to extract one of the parameters passed to a constructor. For example, the following function retrieves the first parameter of a `Node`:

```
function GetLeft(t: BYTree): BYTree  
  requires t.Node?  
{  
  match t  
  case Node(left, _) => left  
}
```

Note that `GetLeft`'s precondition says the function can only be applied to a blue-yellow tree of the `Node` variant. This means that the `match` expression only needs to consider one case.

Since `GetLeft` is a common operation, Dafny provides a convenient way to declare such a *destructor* for each parameter of a constructor, simply by naming the parameter. For example, the following datatype declaration defines a destructor for each argument to `Node`, calling them `left` and `right`, respectively:

```
datatype BYTree = BlueLeaf
                  | YellowLeaf
                  | Node(left: BYTree, right: BYTree)
```

It is not necessary to name a destructor for every constructor argument, but you can if you like.

As with discriminators, destructors are members of the datatype. Thus, whereas function `GetLeft` is invoked by `GetLeft(t)` for an expression `t` of type `BYTree`, a destructor is invoked by `t.left`. Just like we declared a precondition for function `GetLeft` above, a destructor automatically gets that precondition. In particular, the precondition of `t.left` is `t.Node?`, so the destructor `left` can be applied only to `BYTree` values of the `Node` variant.

Note that we can use discriminators and destructors in an alternative implementation of function `BlueCount` from above:

```
function BlueCount(t: BYTree): nat {
  if t.BlueLeaf? then 1
  else if t.YellowLeaf? then 0
  else BlueCount(t.left) + BlueCount(t.right)
}
```

Exercise 4.4.

Define function `LeftDepth` from above in terms of discriminators and destructors and a single **if-then-else** expression.

Exercise 4.5.

Consider a predicate `HasLeftTree` on two blue-yellow trees, `t` and `u`, that determines whether or not `t` is an internal node with `u` as its left subtree. Implement this function in

two different ways, one using a `match` expression and one using a discriminator and destructor. How do you make sure the destructor meets its precondition?

4.3. Structural Inclusion

Datatypes are often defined recursively. We saw this above, where `BYTree`'s `Node` constructor itself takes arguments of type `BYTree`. Consequently, functions on datatypes are often recursive as well. We saw this above, too, where functions `BlueCount` and `LeftDepth` were defined recursively. How can we be sure this recursion terminates?

A salient property of inductive datatypes is that their values can be obtained by a finite number of constructor invocations. For example, the last tree depicted in Section 4.0 can be written as the expression

```
Node(BlueLeaf, Node(YellowLeaf, BlueLeaf))
```

which uses five constructor invocations. We say that the datatype parameters passed to a datatype constructor are *structurally included* in the result. By the finiteness of inductive datatypes, this structural inclusion is a well-founded relation. For example, for any values `t` and `u` of type `BYTree`, we have:

```
Node(t, u) > t
```

and

```
Node(t, u) > u
```

Structural inclusion is the built-in well-founded order on inductive datatypes in Dafny. For example, the **decreases** clause that proves termination of both `BlueCount` and `LeftDepth` above is **decreases** `t`. Since `t` is the single parameter of these functions, Dafny defaults to this **decreases** clause if none is supplied explicitly (see Section 3.4), which explains why the examples above did not produce any complaints about termination.

4.4. Enumerations

Suppose we want to define trees whose leaf nodes can be of more colors than just blue and yellow. We could then add further constructors, like `GreenLeaf` and `RedLeaf`. Alternatively, we can declare a tree datatype with just two constructors, `Leaf` and `Node`, and parameterize `Leaf` with a color:

```
datatype ColoredTree = Leaf(Color)
                      | Node(ColoredTree, ColoredTree)
```

If the color comes from a fixed set of possible colors, then we can use a datatype to define the possible color variants. For example:

```
datatype Color = Blue | Yellow | Green | Red
```

In this special case where every constructor is parameterless, we say that the datatype is an *enumeration*.

As expected, we can define functions on enumerations, like

```
predicate IsSwedishFlagColor(c: Color) {
  c.Blue? || c.Yellow?
}
predicate IsLithuanianFlagColor(c: Color) {
  c != Blue
}
```

Exercise 4.6.

Define a predicate `IsSwedishColoredTree` on colored trees that determines whether or not all leaves have colors that also appear in the Swedish flag.

4.5. Type Parameters

By now, we have already seen two kinds of trees, blue-yellow trees and colored trees. It is not hard to imagine that we will want to have many other kinds of trees as well,

all of which share the basic leaf/node structure. For this purpose, we want to parameterize the definition of a tree with the type of its payload, that is, the data stored in the tree.

Here is a definition of general binary trees. It takes a type T as a parameter:

```
datatype Tree<T> = Leaf(data: T)
                  | Node(left: Tree<T>, right: Tree<T>)
```

This declaration defines a whole family each trees, one for each possible type T . For example, by parameterizing `Tree` with `Color`, written `Tree<Color>`, we denote a type that is similar to `ColoredTree` above. However, we typically think and speak of `Tree` (without a parameter instantiation) as one type. A type that takes type parameters is often called a *generic type*.

We can, of course, define a function on a specific tree type, like

```
predicate AllBlue(t: Tree<Color>) {
  match t
  case Leaf(c) => c == Blue
  case Node(left, right) => AllBlue(left) && AllBlue(right)
}
```

This function can be applied only to values of the type `Tree<Color>`. We can also parameterize a function by a type. This lets us define the function on any type in the family `Tree`. Here is a function `Size`, parameterized by a type T and a value t of type `Tree<T>`, that returns the number of leaf nodes in t :

```
function Size<T>(t: Tree<T>): nat {
  match t
  case Leaf(_) => 1
  case Node(left, right) => Size<T>(left) + Size<T>(right)
}
```

When calling a function, the type arguments can usually be inferred. If so, we can omit them (along with the angle

brackets) from the program text. For example, it would be more common for us to define `Size` as follows:

```
function Size<T>(t: Tree<T>): nat {  
  match t  
  case Leaf(_) => 1  
  case Node(left, right) => Size(left) + Size(right)  
}
```

Exercise 4.7.

Define a function `Mirror` that returns the mirror image of a given tree.

4.6. Abstract Syntax Trees for Expressions

The final example in this chapter defines an *abstract syntax tree* (AST) of simple arithmetic expressions like $(5 + 3) * 2$ and $10 * (x + 7 * y)$. Such data structures are used in every compiler. The example illustrates mutual recursion, and it also defines the common type `List` that we will use in many examples to come.

For the example, expressions have three variants: literal constants, variables, and operations. The corresponding datatype, `Expr`, is

```
datatype Expr = Const(nat)  
              | Var(string)  
              | Node(op: Op, args: List<Expr>)
```

The parameter to the constructor `Const` is the numeric literal being represented. The parameter to `Var` is the name of the variable. Constructor `Node` takes an operation, which in this simple example I define to be one of the following:

```
datatype Op = Add | Mul
```

and a list of (any number of) `Expr` arguments.

The generic type `List` is defined as follows:

```
datatype List<T> = Nil | Cons(head: T, tail: List<T>)
```

This uses the common names `Nil` to denote an empty list and `Cons` to denote a nonempty list. A nonempty list has two components, the head, which carries the payload, and `tail`, which denotes the rest of the list.

As an example, the arithmetic expression $10 * (x + 7 * y)$ is represented by

```
Node(Mul,
  Cons(Const(10),
    Cons(Node(Add,
      Cons(Var("x"),
        Cons(Node(Mul,
          Cons(Const(7),
            Cons(Var("y"),
              Nil))),
          Nil))),
      Nil))),
  Nil)))
```

Let's define a simple interpreter for our expressions. It computes the value of a given expression. Since the value of a variable depends on some *environment*, we let the evaluation function take a *map* from variable names to values.

```
function Eval(e: Expr, env: map<string, nat>): nat {
  match e
  case Const(c) => c
  case Var(s) => if s in env.Keys then env[s] else 0
  case Node(op, args) => EvalList(args, op, env)
}

function EvalList(args: List<Expr>, op: Op,
  env: map<string, nat>): nat
{
  match args
  case Nil =>
    (match op case Add => 0 case Mul => 1)
  case Cons(e, tail) =>
    var v0, v1 := Eval(e, env), EvalList(tail, op, env);
    match op
    case Add => v0 + v1
    case Mul => v0 * v1
}
```

Four remarks are in order.

First, we will see many examples with collections such as maps later. For this example, it suffices to know that a map of type `map<A, B>` associates with each of a finite number of A values a B value. The former are called the *keys* of the map and the latter are called the *values* of the map. The set of keys of the map is retrieved using the `.Keys` member. So the expression `a in m.Keys` tells you whether or not `a` is a key of `m`. For an `a` known to be a key of `m`, the expression `m[a]` yields the B value that `m` associates with the key `a`.

I chose to make our method `Eval` work on any given environment, even one that does not contain a mapping for every variable mentioned in the given expression. If a variable name in the expression is not mapped by the environment, `Eval` uses `0` as a default value. An alternative would be to require as a precondition of `Eval` that the given environment has a mapping for every variable mentioned in the given expression, see Exercises 4.9 and 4.10.

Second, the `Cons` case of `EvalList` uses local variable bindings to introduce names for the results of the recursive calls to `Eval` and `EvalList`. These could have been written as

```
var v0 := Eval(e, env);  
var v1 := EvalList(tail, op, env);
```

but instead, I chose to write them as a simultaneous assignment. Choosing between two separate assignments and one simultaneous assignment is a matter of style, though the latter is of course possible only if the right-hand sides can be evaluated before the bindings take place (in other words, if `v0` is not used in the right-hand side for `v1` and vice versa). In a `match` construct, any variable introduced in one `case` goes out of scope at the end of that `case`.

A `var` binder in an expression is usually called a *let expression*. This name comes from the common functional-languages syntax

```
let x = E in F // not Dafny syntax
```

which evaluates expression E , binds the result to variable x , and then results in the evaluation of F , which may mention x . As we just saw, the syntax for such a `let`-expression in Dafny is

```
var x := E; F // Dafny syntax, which can be pronounced as
              // "let x be E in F"
```

which is made to look like the statement counterpart that introduces local variables.

Third, note that the first nested `match` expression requires enclosing parentheses, or else `case Cons(e, tail)` would be parsed as a third case of the `match` op. In contrast, note that the second nested `match` does not need more parentheses, since all remaining `case`'s belong to it. Syntactically, Dafny allows the use of curly braces to group together all `case`'s of a `match`; so as an alternative to writing `(match op ...)`, one can write

```
match op {
  case Add => 0
  case Mul => 1
}
```

Note the placement of the curly braces, compared to the placement of the enclosing round parentheses.

Fourth, the functions `Eval` and `EvalList` are mutually recursive. Termination follows, because the datatype argument, e for `Eval` and $args$ and `EvalList`, are structurally smaller for every (mutually) recursive call. That is, the essential ingredients of the proof of termination are

```
Node(op, args) > args
Cons(e, tail) > e
Cons(e, tail) > tail
```

Since these parameters happen to be listed first in the function signatures, Dafny's default **decreases** clauses suffice for the verifier to check termination.

| Exercise 4.8.

Change the signature of function `EvalList` so that parameter `op`: `Op` precedes parameter `args`: `List<Expr>`. This will cause Dafny to emit an error about not being able to prove termination. Why? Solve this problem by writing an explicit **decreases** clause for `EvalList`.

Exercise 4.9.

Write a predicate `GoodEnv(e: Expr, env: map<string, nat>)` that holds whenever the keys of `env` include all the variable names mentioned in `e`. This will also require writing a similar, auxiliary predicate for not just one expression `e`, but for a list of expressions.

Exercise 4.10.

Change the `Var` case of function `Eval` to be simply `env[s]`. Using the predicates developed in Exercise 4.9, add the necessary preconditions to make `Eval` and `EvalList` verify.

4.7. Summary

A datatype gives a way to define a set of values. The values of a datatype are partitioned into a fixed set of variants. The datatype has one constructor for each variant. The constructor is like a function in that it can take parameters. It is a special function, because it is *injective*, which means that it is possible to recover the parameters passed to the constructor from the value produced by the constructor. This deconstruction is commonly done with a `match` statement or `match` expression, but can also be done using a combination of discriminators and destructors.

I've referred to the datatypes as “inductive”. This essentially means that each of their values is constructed bottom-up using a finite number of steps. A value that is produced by a constructor is said to structurally include any datatype values passed in as parameters to the

constructor. Because of the finite structure of datatype values, this structural inclusion is well-founded.

In the next chapter, we'll explore mathematical induction, which is a way to write proofs in terms of smaller proofs. This is not unlike inductive datatypes. Indeed, to prove properties of inductive datatypes, you almost always use mathematical induction, but there are also many other applications of mathematical induction.

Notes

Inductive datatypes are the bread and butter of all languages and verifiers based on functional programming. For example, the definition of type `Tree` is written

```
type tree (t:Type) : Type =  
  | Leaf : data:t -> tree t  
  | Node : left:tree t -> right:tree t -> tree t
```

in F* [53]. Enumerations (that is, datatypes where the constructors don't take any parameters) are supported by most imperative languages. The use of payloads is also possible in languages like Scala, Rust, Java, and C#, but because these languages expose details of the allocation of such values in memory, they are harder to use in specification and verification.

The datatypes introduced in this chapter are inductive, which implies their structure is finite. For example, if you keep following the left branch of a `BYTree`, you will eventually get to a leaf. Although much less common, there are also *coinductive datatypes*, whose values can have an infinite structure. At first, it may seem that such values are infeasible for a programming language whose programs will execute in a computer's finite memory. But by evaluating these structures *lazily*, one can represent finite portions of the infinite structures. This can lead to beautiful renditions of programs, a hallmark of the Haskell

programming language [\[110\]](#). The duality of inductive and coinductive datatypes is emphasized in the prototype programming language Charity [\[28\]](#). Coinductive datatypes can also be used in specifications to describe the behavior of some programs. For example, you can use a coinductive datatype to represent an endless stream of requests being processed by a web server. Dafny, Liquid-Haskell [\[86\]](#), and Agda [\[22\]](#) support coinductive datatypes, but proving properties of such programs is an advanced topic beyond the scope of this book.

Chapter 5

Lemmas and Proofs



As we reason about programs, we often need to write some parts of the proofs manually. When such a proof is long or when we want to reuse the same proof in multiple places, we give the property we're proving a name. This is done by a *lemma*, which you can think of as a subroutine in a larger proof.

This chapter focuses on writing and proving lemmas. It is central to everything we do when reasoning about programs. We'll see that there is a strong link between proofs and Floyd logic, we'll learn about induction, and we'll be writing proof calculations. Crucially, you'll also start to see how to interact with the verifier when it complains that it cannot prove your program.

The thinking and debugging you do when proving a lemma is the same as when proving programs correct. Therefore, practicing the skills of writing proofs gives you the chops to reason about more difficult programs. When you're comfortable writing proofs, you'll be able to think about the interesting aspects of your programs instead of spending your time wondering how to diagnose failed proof attempts.

5.0. Declaring a Lemma

To illustrate the need for lemmas, let me start with a simple example. Consider the following function, which always returns something larger than its argument:

```
function More(x: int): int {  
    if x <= 0 then 1 else More(x - 2) + 3  
}
```

I said More “always returns something larger than its argument”. Does it? How can we be sure of that? If you thought about this in your head just now, you would be considering two cases, just like the body of the function has two cases. In one case, when x is no greater than 0, it is easy to see that $\text{More}(x)$ returns 1, which is indeed larger than such an x . In the other case, when x is strictly positive, perhaps you thought about the repeated unwinding of recursive calls until the first case applies. The function adds 3 to the result of each recursive call, so if we count the recursive calls correctly, maybe we can convince ourselves, or at least find it plausible, that $\text{More}(x)$ always returns something greater than x .

We'll prove this property about `More` later (in Section 5.2). For now, let's just think about how to write down the property itself. Here's how you do that, where I've given the property the name `Increasing`:

```
lemma Increasing(x: int)
```

ensures $x < \text{More}(x)$

A *lemma*, or some would call it a *theorem*, is a mathematical assertion that has a proof. In Dafny, a lemma is very much like a method: it has a name, it can be parameterized, it has a pre- and postcondition, and it can be called. The logical assertion made by the lemma is declared by the postcondition. Just like the caller of a method learns the postcondition upon termination of the call, so the caller of a lemma learns the logical assertion made by the lemma. Any restrictions on when the lemma can be applied are declared by the precondition of the lemma, again just as for methods.

Well, there is a difference between methods and lemmas: a lemma is never compiled into code for use at run time. Instead, the lemma is used only by the verifier, like a ghost construct. In fact, in Dafny, a lemma is simply a ghost method. This means you already know how to declare, use, and—yes—*prove* lemmas. But all of these things are worthy of explanation again, thinking of the lemmas as named mathematical assertions.

5.1. Using a Lemma

Let's consider an example that illustrates the use of this lemma. The following method makes two calls to `More`:

```
method ExampleLemmaUse(a: int) {  
  var b := More(a);  
  var c := More(b);  
  assert 2 <= c - a;  
}
```

The example method computes `More(More(a))` into local variable `c` and then asserts that the difference between `c` and `a` is at least 2. If `More` always returns something greater than its argument, then `More(a)` returns something greater than `a`, that is, at least `a+1`. The second call to `More` is

invoked on the result of the first call, so we expect its result to be at least $a+2$. But the verifier complains that the assertion may not hold. The reason is that the verifier does not know enough about `More`. In particular, it needs the `Increasing` property that we stated as a lemma in Section 5.0.

To prove the assertion in `ExampleLemmaUse`, we invoke the `Increasing` lemma. We need two calls to the lemma, since we need the `More` property for both `a` and `More(a)`.

```
method ExampleLemmaUse(a: int)
{ ①
  var b := More(a); ①
  Increasing(a); ②
  Increasing(b); ③
  var c := More(b); ④
  assert 2 <= c - a; ⑤
}
```

Now the assertion verifies. 😊

It is worthwhile, just this once, to go through the example in excruciating detail so we can see how program correctness and lemmas relate. Let's consider what is known at each step of the method.

- At ①, all we know is that `a` is an integer. It could be any integer.
- At ①, we additionally know that `b` holds the value of `More(a)`. If we traced through the definition of `More`, we could learn more, but since `More` is recursive, we won't get very far without an inductive proof, which we haven't yet done.
- Next up is the call to lemma `Increasing`. This means there is a proof obligation to establish the precondition of this call. Luckily, since `Increasing` does not have a declared precondition, this is trivial. Upon return of the call, at ②, we therefore gain the knowledge of the call's postcondition, namely $a < \text{More}(a)$.

- Similarly, at ③, we additionally learn $b < \text{More}(b)$. Since we know that b equals $\text{More}(a)$ at this point, we can conclude
 $\text{More}(a) < \text{More}(\text{More}(a))$
- In the next line, local variable c is set to $\text{More}(b)$. Since the value of b is $\text{More}(a)$, the value of c is $\text{More}(\text{More}(a))$. Thus, we know the following equality and inequalities at ④:
 $a < \text{More}(a) < \text{More}(\text{More}(a)) == c$
 Since we're dealing with integers, these conditions imply $a + 2 \leq c$.
- We must now prove that the assertion holds, given what we know to be true at ④. This assertion itself does not give us any more information, so at ⑤ we know the same as what we know at ④.

Note that lemma calls are just like method calls in that we need to (of course!) invoke them on the right values and at the right times. For example, the following code does not verify the assertion, since it invokes the lemma on a and $\text{More}(\text{More}(a))$, but never on $\text{More}(a)$:

```
var b := More(a);
Increasing(a); // this gives us: a < More(a)
var c := More(b);
Increasing(c); // here, this gives us:
                // More(More(a)) < More(More(More(a)))
assert 2 <= c - a; // this assertion cannot be verified here
```

On the other hand, since the lemma doesn't modify the program state—it only gives us more knowledge about the program state—we can write our two lemma calls in the other order:

```
var b := More(a);
Increasing(b); // this gives us: More(a) < More(More(a))
Increasing(a); // and this gives us: a < More(a)
var c := More(b);
assert 2 <= c - a; // verifies fine
```


Finally, just like calls to other methods, the information gained from a lemma call applies only along the control paths that actually call the lemma:

```
Increasing(a); // this call is along all control paths
var b := More(a);
var c := More(b);
if a < 1000 {
    Increasing(More(a)); // we learn More(a) < More(More(a))
                        // only here, i.e., when a < 1000
    assert 2 <= c - a; // this verifies
}
assert 2 <= c - a; // this does not verify
```

The last assertion above does not verify, because we're missing the information `More(a) < More(More(a))` when the (implicit) else branch of the `if` is taken.

Exercise 5.0.

Change the last assertion in the previous example to

```
assert 2 <= c - a || 200 <= a;
```

What happens? Why?

Exercise 5.1.

Method `ExampleLemmaUse` introduced two local variables, `b` and `c`. Instead of introducing `c`, we could have updated the value of `b`, as in

```
b := More(b);
```

Reuse `b` and eliminate `c` in this way in the five variations of the body of `Example-LemmaUse` in this section. Do you need to adjust the assertions and lemma calls? Justify.

5.2. Proving a Lemma

So far, we only declared the lemma and used it, but we never proved it. To prove it, we need to supply the lemma with a body. This is in analogy to how we supply a method with a body to give the method an implementation. A

lemma without a body is therefore an unproved lemma, sometimes referred to as a *conjecture* or an *axiom*.

Sidebar 5.0

While the *verifier* is happy to accept a body-less method (or lemma), the *compiler* will complain if it finds a method (or lemma) without a body. In fact, even though the compiler erases all ghost declarations, it will first check that the erasure is permitted. In this process, it will complain about body-less lemmas. When you work in the integrated development environment (IDE), the verifier runs in the background, but the compiler does not. Therefore, you won't see complaints about body-less methods, lemmas, and functions until you (manually) run the compiler.

It is time to add a body to our lemma, so bring out them curly braces:

```
lemma Increasing(x: int)  
  ensures x < More(x)  
{  
}
```

If you try this out, you may be surprised that you *still* don't get any complaint from the verifier. This time, the reason is that the verifier proves the lemma automatically, without needing any further input from you (but you *do* have to supply the empty body, or the verifier will not even attempt a proof). Behind the scenes, the verifier automatically applies a smidgen of induction, and this smidgen is enough to prove this simple lemma. This is really too easy! Okay, we'll appreciate this later, but it doesn't help us learn about

proofs and induction. Therefore, for the rest of this chapter, we shall turn off the automatic induction for each lemma. This is done by marking the lemma with the attribute `{:induction false}`. (It is also possible to change the settings in the Dafny IDEs to turn off automatic induction everywhere.)

So, here's our starting point:

```
lemma {:induction false} Increasing(x: int)
  ensures x < More(x)
{
} // the lemma does not verify like this
```

This gives an error from the verifier, because it is unable to confirm that all control paths through the method body lead to the postcondition. Let's recall our informal reasoning from Section 5.0. We said the proof is easy to see for the case when $x \leq 0$. Making this explicit in the program text is a good idea, and we can do that with an `if` statement:

```
lemma {:induction false} Increasing(x: int)
  ensures x < More(x)
{
  if x <= 0 { // this case verifies
  } else { // this case does not verify yet
  }
}
```

The verifier still responds with a complaint, but this time the complaint only highlights the `else` branch—it has proven the postcondition for all control paths that go through the then branch. (Strictly speaking, it's too early to celebrate having a proof of the then branch. The verifier usually considers controls in the same order as they are given in the program text, but this is not guaranteed.) The then branch is taken when $x \leq 0$ holds; in those cases, the definition of `More(x)` gives 1; so we can conclude $x \leq 0 < 1 == \text{More}(x)$.

It is no accident that the `if` condition in the lemma about function `More` is the same as the `if` condition in the definition of function `More`. This is because the proof usually depends on which part of the function's definition is being used. As a good rule of thumb, proofs follow the syntactic structure of what is being proved.

For the `else` branch of the lemma, we need to help the verifier along more. By definition, `More(x)` equals `More(x-2) + 3` when `x <= 0` does not hold. Our proof obligation is therefore `x < More(x-2) + 3`, which is the same as `x-3 < More(x-2)`. The shape of this formula is the same as that in our lemma. To gain this information, all we need to do is call the lemma on `x-2`:

```
lemma {:induction false} Increasing(x: int)
  ensures x < More(x)
{
  if x <= 0 {
  } else {
    Increasing(x-2); // this call gives us: x-2 < More(x-2)
  }
}
```

This completes the proof, but let's not forget a detail that the verifier took care of automatically for us. Because the lemma is called recursively, we need to prove termination. By default, Dafny uses **decreases** `x` as the termination metric (see Section 3.4), and indeed, our recursive call makes this termination metric decrease.

Termination checks are important, because they prevent proof attempts like this one:

```
lemma {:induction false} Increasing(x: int)
  ensures x < More(x)
{
  Increasing(x); // this would give us x < More(x), but
                // the termination check fails
}
```

This proof attempt is bogus for the same reason that the attempted implementation of method `Impossible` in Section

3.0 is—both try to achieve their postcondition with a never-ending recursive call.

Exercise 5.2.

Using the same level of detail as in the bulleted list in Section 5.1, spell out what is known at each program point in the last good version of lemma `Increasing` above.

The recursive call to the lemma obtains the property we're trying to prove for a smaller instance of the problem. This is called *induction*. The *Principle of Induction* says that when proving a property $P(x)$, we are free to assume that $P(x')$ holds for every x' that is smaller than x . “Smaller” means smaller in a well-founded order, as discussed in Section 3.2. If you just think of a lemma as a method, it all just comes down to proving termination of recursive calls. Nevertheless, following standard mathematical jargon, we say that recursive lemma calls obtain the *induction hypothesis*.

As our rule of thumb says, a proof tends to follow the same structure as the formula that is being proved. They don't need to have the same syntactic form, though. For example, we can reverse the order of then and else branches, we can omit empty else branches, and we can make use of local variables. Here is another proof of lemma `Increasing`:

```
lemma {:induction false} Increasing(x: int)
  ensures x < More(x)
{
  if 0 < x {
    var y := x - 2;
    Increasing(y);
  }
}
```

5.3. Back to Basics

The proof we wrote for lemma `Increasing` is pleasantly short. It convinces the verifier, but what if we still don't get it ourselves? Or perhaps you are convinced by the proof in retrospect, but you're wondering how you would have come up with it in the first place. Let's aim for more details, starting here with the simple `Increasing` lemma.

5.3.0. Floyd-logic proof

I presented Floyd logic in Chapter 2 as the bread and butter for reasoning about programs. Since proofs of lemmas are just implementations of methods, Floyd logic applies equally well here. Let's use that technique to build up the proof of `Increasing`.

The overall proof of `Increasing` starts with the lemma's precondition (here, the implicit requires `true`) and goes to the lemma's postcondition (the proof goal ensures $x < \text{More}(x)$). Using Hoare-triple notation, we thus write the body of `Increasing` as follows:

```
lemma {:induction false} Increasing(x: int)
  ensures  $x < \text{More}(x)$ 
{
  {{ true }}
  ?
  {{  $x < \text{More}(x)$  }}
}
```

Focusing in on where I wrote a question mark, it makes sense to break the proof into two cases, corresponding to the two cases in the definition of `More`. Using the scaffolding of conditional statements (see Section 2.5), we write

```
{{ true }}
if  $x \leq 0$  {
  {{  $x \leq 0$  }}
  ?
  {{  $x < \text{More}(x)$  }}
} else {
  {{  $0 < x$  }}
```

```

    ?
    {{ x < More(x) }}
}
{{ x < More(x) }}

```

The introduction of the **if** statement was motivated by the definition of **More**, so let's add a fact of the form **More(x) == ...** in each branch. We'll use the consecutive-formula notation from Section 2.3.1:

```

{{ true }}
if x <= 0 {
    {{ x <= 0 }}
    {{ x <= 0 && More(x) == 1 }} // def. More for x <= 0
    ?
    {{ x < More(x) }}
} else {
    {{ 0 < x }}
    {{ 0 < x && More(x) == More(x - 2) + 3 }} // def. More for 0 < x
    ?
    {{ x < More(x) }}
}
{{ x < More(x) }}

```

The formula before the first question mark implies the formula after that question mark. This means we are done with the proof of the first branch of the conditional.

For the second branch, we want to know more about **More(x - 2)**. By calling lemma **Increasing** on **x - 2**, we obtain the induction hypothesis **x - 2 < More(x - 2)**. In our notation, we continue to expand the **else** branch of our proof as follows:

```

{{ 0 < x }}
{{ 0 < x && More(x) == More(x - 2) + 3 }} // def. More for 0 < x
Increasing(x - 2); // induction hypothesis
{{ 0 < x && More(x) == More(x - 2) + 3 && x - 2 < More(x - 2) }}
?
{{ x < More(x) }}

```

It doesn't look like we'll need $0 < x$ any more, so let's drop it, and let's also rewrite $x - 2 < \text{More}(x - 2)$ by adding 3 to each side:

```
{{ 0 < x }}
```

```
{{ 0 < x && More(x) == More(x - 2) + 3 }} // def. More for 0 < x
```

```
Increasing(x - 2); // induction hypothesis
```

```
{{ 0 < x && More(x) == More(x - 2) + 3 && x - 2 < More(x - 2) }}
```

```
{{ More(x) == More(x - 2) + 3 && x + 1 < More(x - 2) + 3 }}
```

```
?
```

```
{{ x < More(x) }}
```

The two facts about $\text{More}(x - 2) + 3$ on the line before the question mark imply $x + 1 < \text{More}(x)$, so we write:

```
{{ 0 < x }}
```

```
{{ 0 < x && More(x) == More(x - 2) + 3 }} // def. More for 0 < x
```

```
Increasing(x - 2); // induction hypothesis
```

```
{{ 0 < x && More(x) == More(x - 2) + 3 && x - 2 < More(x - 2) }}
```

```
{{ More(x) == More(x - 2) + 3 && x + 1 < More(x - 2) + 3 }}
```

```
{{ x + 1 < More(x) }}
```

```
?
```

```
{{ x < More(x) }}
```

It's now easy to see that the line before the question mark implies the one after the question mark (because if $\text{More}(x)$ exceeds $x + 1$, then it also exceeds x). So, we have completed the proof.

In summary, our whole proof of `Increasing` looks like this:

```
lemma {:induction false} Increasing(x: int)
  ensures x < More(x)
{
  {{ true }}
  if x <= 0 {
    {{ x <= 0 }}
    {{ x <= 0 && More(x) == 1 }} // def. More for x <= 0
    {{ x < More(x) }}
  } else {
    {{ 0 < x }}
```



```

    {{ 0 < x && More(x) == More(x - 2) + 3 }} // def. More for
0 < x
    Increasing(x - 2); // induction hypothesis
    {{ 0 < x && More(x) == More(x - 2) + 3 &&
        x - 2 < More(x - 2) }}
    {{ More(x) == More(x - 2) + 3 &&
        x + 1 < More(x - 2) + 3 }}
    {{ x + 1 < More(x) }}
    {{ x < More(x) }}
}
{{ x < More(x) }}
}

```

In this Floyd-logic proof, you can check each step (that is, each Hoare triple and each pair of consecutive formulas). When you check the recursive call to `Increasing`, you'll also need to check termination, which I glossed over a moment ago when constructing this proof.

5.3.1. Proof assertions

The Floyd-logic proof above uses a lot of the `{{ }}` notation, which looks fine on paper, but isn't accepted as syntax in a Dafny program. What if we want the help of the program verifier to check these conditions? We can then replace each of the formulas with an **assert** statement.

If we go crazy with this idea, we'll write the proof of our lemma as follows (which *is* accepted by Dafny):

```

lemma {:induction false} Increasing(x: int)
  ensures x < More(x)
{
  assert true;
  if x <= 0 {
    assert x <= 0;
    assert x <= 0 && More(x) == 1; // def. More for x <= 0
    assert x < More(x);
  } else {
    assert 0 < x;
    assert 0 < x && More(x) == More(x - 2) + 3; // def. More
for 0 < x
    Increasing(x - 2); // induction hypothesis

```

```

    assert 0 < x && More(x) == More(x - 2) + 3 &&
           x - 2 < More(x - 2);
    assert More(x) == More(x - 2) + 3 &&
           x + 1 < More(x - 2) + 3;
    assert x + 1 < More(x);
    assert x < More(x);
}
assert x < More(x);
}

```

If you made a mistake in any of these assertions, the program verifier will tell you instantly. Great! But this level of verbosity is not very readable for a human. We want proofs and programs to be as to-the-point as possible. So, let's think of ways to reduce the clutter.

Let's omit the first and last assertions, since they already appear as the pre- and postcondition of the lemma itself. Let's also omit the scaffolding of the **if** statement, since that scaffolding just copies the **if** guard (or its negation) to the beginning of each branch and copies the postcondition to the end of each branch. We are now left with

```

lemma {:induction false} Increasing(x: int)
  ensures x < More(x)
{
  if x <= 0 {
    assert More(x) == 1; // def. More for x <= 0
  } else {
    assert More(x) == More(x - 2) + 3; // def. More for 0 < x
    Increasing(x - 2); // induction hypothesis
    assert More(x) == More(x - 2) + 3 &&
           x - 2 < More(x - 2);
    assert More(x) == More(x - 2) + 3 &&
           x + 1 < More(x - 2) + 3;
    assert x + 1 < More(x);
  }
}

```

If it seems clear from the program text that the variables in a formula are not changed, then we don't need to repeat the formula. So, let's omit any formula that is repeated. This gives us:

```

lemma {:induction false} Increasing(x: int)
  ensures x < More(x)
{
  if x <= 0 {
    assert More(x) == 1; // def. More for x <= 0
  } else {
    assert More(x) == More(x - 2) + 3; // def. More for 0 < x
    Increasing(x - 2); // induction hypothesis
    assert x - 2 < More(x - 2); // what we get from the
recursive call
    assert x + 1 < More(x - 2) + 3; // add 3 to each side
    assert x + 1 < More(x); // previous line and def. More
above
  }
}

```

Again, the program verifier immediately alerts you of any mistakes you make in writing down these assertions.

Exercise 5.3.

Change some assertions above to introduce mistakes to see what happens. Also, what happens if you omit the recursive call to `Increasing`?

5.4. Proof Calculations

There are more structured ways to write proofs. I have already used a format for writing a sequence of expressions and explaining why each successive expression is equal to the previous. For example, in the weakest-precondition calculations in Sections 2.7.2, 2.10, and 2.11, I used a notation like that. I used a similar style of assertions interleaved with explanations in the Dubious example of Section 3.0, and the consecutive-formula notation of Section 2.3.1 is also a form of step-by-step transformations to argue that one formula implies another. This structured form of proofs is called *proof calculations*. They are directly supported in Dafny, where they are commonly known as “the `calc` statement”.

A proof calculation gives a list of expressions related by a chain of operators. Here is an example:

```
calc {
  5 * (x + 3);
== // distribute multiplication over addition
  5 * x + 5 * 3;
== // use the arithmetic fact that 5 * 3 == 15
  5 * x + 15;
}
```

This calculation says that the value of expression $5 * (x + 3)$ equals the value of expression $5*x + 5*3$ and, in turn, that $5*x + 5*3$ equals $5 * x + 15$. Note that each expression is terminated by a semi-colon, so that the operator is not interpreted by Dafny as being part of the expression. (If you wrote the expressions on paper, you'd leave off these semi-colons, because your line breaks and indentation would make it clear where expressions end. Dafny requires the semi-colons, since whitespace is not significant in Dafny.) The verifier checks each step of the calculation and then concludes, by transitivity of equality, that $5 * (x + 3)$ equals $5 * x + 15$. When, as in this example, the operator is equality, the proof calculation can be seen as a series of equality-preserving transformations: expression $5 * (x + 3)$ is transformed into $5*x + 5*3$, and $5*x + 5*3$ is in turn transformed into $5 * x + 15$.

Here is another example:

```
calc {
  (x + y) * (x - y);
== // distribute * over + and - (i.e., cross multiply)
  x*x - x*y + y*x - y*y;
== // * is commutative: y*x == x*y
  x*x - x*y + x*y - y*y;
== // the terms - x*y and + x*y cancel
  x*x - y*y;
}
```

Calculations can also use other operators, provided the chain is in the same direction. For example, if n has type **nat**, then we have

```

calc {
  3*x + n + n;
  == // n + n == 2*n
  3*x + 2*n;
  <= // 2*n <= 3*n, since 0 <= n
  3*x + 3*n;
  == // distribute * over +
  3 * (x + n);
}

```

By verifying each step in this calculation, we then conclude

$3*x + n + n \leq 3 * (x + n)$

by the transitivity of the chaining operators. Note that the middle step holds because of our assumption $0 \leq n$. If n could be an arbitrary integer, then this step in the proof is in error (and the verifier would flag it as such).

Exercise 5.4.

For each of the following equations, write a proof calculation that proves it. Then, use Dafny to declare a lemma of the property, and record your proof using a **calc** statement.

- a) $5*x - 3 * (y + x) == 2*x - 3*y$
- b) $2 * (x + 4*y + 7) - 10 == 2*x + 8*y + 4$
- c) $7*x + 5 < (x + 3) * (x + 4)$

Exercise 5.5.

For boolean expressions A, B, and C, explain the difference between

```

calc {
  A;
  ==>
  B;
  ==>
  C;
}

```

and

```

assert A ==> B ==> C;

```

5.4.0. Checked proof hints

Some steps in a calculation may require further justification. In other words, the verifier may require further hints for some steps. For example, the correctness of a step may require the calls to various lemmas. Such hints can be given in curly braces after the operator that relates two successive lines in the calculation. Even for steps that the verifier can handle automatically, it can be nice to supply a human-readable comment, a practice I follow in this book.

Here is a **calc** version of the more detailed proof of Increasing given in Section 5.2:

```
lemma {:induction false} Increasing(x: int)
  ensures x < More(x)
{
  if x <= 0 {
    // let's get ridiculously detailed, to illustrate what you
    can do
    calc {
      x;
      <= { assert x <= 0; }
      0;
      < // arithmetic
      1;
      == // def. More
      More(x);
    }
  } else {
    // here's a nice calculation
    calc {
      More(x);
      == // def. More, since 0 < x
      More(x - 2) + 3;
      > { Increasing(x - 2); }
      x - 2 + 3;
      > // arithmetic
      x;
    }
  }
}
```

In both of these calculations, the expression on each line has type integer. Here is a way to write the second calculation using boolean expressions:

```
calc {  
  true;  
==> { Increasing(x - 2); }  
  x - 2 < More(x - 2);  
== // add 3 to each side  
  x + 1 < More(x - 2) + 3;  
== // def. More, since 0 < x  
  x + 1 < More(x);  
==> // arithmetic  
  x < More(x);  
}
```

It is common to start a calculation with the “more complicated” side, since this gives more guidance for what to apply in subsequent steps. Following that practice, the previous `calc` statement can be formulated in the reverse order:

```
calc {  
  x < More(x);  
<== // arithmetic  
  x + 1 < More(x);  
== // def. More, since 0 < x  
  x + 1 < More(x - 2) + 3;  
== // subtract 3 from each side  
  x - 2 < More(x - 2);  
<== { Increasing(x - 2); }  
  true;  
}
```

If you're reading this proof from top to bottom, the sudden introduction of the +1 in the first step may come as a surprise. Like, why would you think to add 1 to x as the first step of the proof? If so, you may prefer to reorder the steps, perhaps like this:

```
calc {  
  x < More(x);  
== // def. More, since 0 < x  
  x < More(x - 2) + 3;
```

```

== // subtract 3 from each side, to get More(x-2) alone on the
right
  x - 3 < More(x - 2);
<== // arithmetic
  x - 2 < More(x - 2);
<== { Increasing(x - 2); }
  true;
}

```

Just how the `calc` statement, or indeed any proof, is formulated is mostly up to personal taste.

Exercise 5.6.

State and prove a lemma that, for any natural number n , $\text{Ack}(1, n) == n + 2$, where Ack is the Ackermann function defined in Section 3.3.1.

5.5. Example: Reduce

Let's do another example that involves arithmetic, lemmas, proof calculations, and induction. The example uses the following function, which, compared to function `More` earlier in this section, is less obvious.

```

function Reduce(m: nat, x: int): int {
  if m == 0 then x else Reduce(m / 2, x + 1) - m
}

```

Looking at the body of this function, we see that it sometimes returns the argument x and other times it's subtracting a positive value from what is returned by a recursive call. While the x argument gets 1 bigger with each recursive call, the amount subtracted off gets halved with each recursive call. So, might it be the case that $\text{Reduce}(m, x)$ never returns anything larger than x ? Let's try to prove this.

5.5.0. An upper bound

We declare a lemma (again, we will turn off Dafny's automatic induction, so that we get a chance to do something ourselves) and, following the definition of `Reduce`, we consider two cases in the body of the lemma:

```
lemma {:induction false} ReduceUpperBound(m: nat, x: int)
  ensures Reduce(m, x) <= x
{
  if m == 0 {
    // trivial
  } else {
    // ...
  }
}
```

The case where `m` is 0 follows immediately from the definition of `Reduce`, since `Reduce(0, x) == x`. The verifier performs this trivial step automatically, but if we thought it would make the proof more human readable, we could add an **assert** statement with this condition.

The other case is more interesting. Let's start a proof calculation from the expression `Reduce(m, x)` and apply the definition of `Reduce` to it:

```
calc {
  Reduce(m, x);
== // def. Reduce
  Reduce(m / 2, x + 1) - m;
```

To obtain more information about this call to `Reduce`, we will apply the induction hypothesis. For our own benefit (since we're taking pretty small proof steps at this time while learning how to write proofs), we can follow the recursive call to the lemma with an assertion that spells out the lemma's postcondition instantiated with the parameters we gave it:

```
== { ReduceUpperBound(m / 2, x + 1);
    assert Reduce(m / 2, x + 1) <= x + 1; }
```

The verifier confirms that we are allowed to invoke the induction hypothesis on the given arguments. In other words, the verifier checks that we are allowed to make the

recursive call. Recall, this check comes down to verifying the precondition and termination of the call (the latter follows from the fact $m/2 < m$). The verifier also confirms that the assertion we wrote is correct.

Okay, so now what? There is no way for us to do an equality-preserving transformation from the information we gained in this hint. So, apparently we wrote the `==` before the hint a bit too hastily. The information gained by the induction hypothesis does give us a way to “grow” the expression `Reduce(m/2, x+1)` into something possibly larger, namely `x+1`. So, we will use `<=` as the relational operator in this proof step, which gives us the following next step:

```

    Reduce(m / 2, x + 1) - m;
  <= { ReduceUpperBound(m / 2, x + 1);
      assert Reduce(m / 2, x + 1) <= x + 1; }
    x + 1 - m;

```

This looks good, since we have now eliminated any mention of `Reduce` from the expression we are calculating with. Using the fact that `m` is strictly positive in this proof branch, we can grow `x + 1 - m` into `x`. Here is the full lemma and its proof:

```

lemma {:induction false} ReduceUpperBound(m: nat, x: int)
  ensures Reduce(m, x) <= x
{
  if m == 0 {
    // trivial
  } else {
    calc {
      Reduce(m, x);
    == // def. Reduce
      Reduce(m / 2, x + 1) - m;
    <= { ReduceUpperBound(m / 2, x + 1);
        assert Reduce(m / 2, x + 1) <= x + 1; }
      x + 1 - m;
    <= { assert 0 < m; }
      x;
    }
  }
}

```

This proof is at an appropriate level of detail for our learning at this point. It also gives a taste for how we develop proofs and how we “debug” why an attempted verification fails. These are good skills to learn and they require practice. Therefore, in the rest of this chapter, I will continue to write proofs that include more details than the verifier needs.

Exercise 5.7.

Write five variations of the proof of `ReduceUpperBound`. Here are some things you can try: do the calculation steps in the opposite order (starting with x and ending with `Reduce(m, x)`), remove proof steps that only apply a function definition, remove `assert` statements, replace the calculation with just the contents of its hints, change the `if` condition to $m \neq 0$ and omit the else branch. (Here and elsewhere in the book, you may try on your own to alter or eliminate some proof steps to discover what the verifier can do automatically and to develop a taste for what style and level of detail you prefer.)

Sidebar 5.1

The Dafny IDE can give you feedback about each line of the proof while you're entering it in the editor. This quickly lets you detect any problems in your thinking. The verifier runs automatically in the background, but it only gets started on programs that are syntactically valid and type check.

Therefore, it is well advised to include the closing curly brace of the `calc` statement right away. This keeps the program syntactically valid, so the verifier can run. In the progressive developments of proof calculations in this book, I don't show the

final closing brace until the end, but you should make sure to always include it from the start of the development.

5.5.1. A lower bound

We proved an upper bound of $\text{Reduce}(m, x)$. Let us now also prove a lower bound, namely $x - 2 * m$. We start by declaring the lemma and doing the usual case split on whether or not m is 0:

```
lemma {:induction false} ReduceLowerBound(m: nat, x: int)
  ensures x - 2 * m <= Reduce(m, x)
{
  if m == 0 {
  } else {
    // ...
  }
}
```

The verifier immediately reassures us that the then branch is fine. For the else branch, we start a proof calculation. This time, we'll start with the right-hand side of the proof goal, since it gives us more guidance in how to proceed with the proof. Since we're trying to prove that the right-hand side is at least the left-hand side, we can include steps in our calculation that may decrease the value of the expression we're manipulating. Here is our first step:

```
calc {
  Reduce(m, x);
  == // def. Reduce
  Reduce(m / 2, x + 1) - m;
```

Next, we invoke the induction hypothesis, much like in the proof of `ReduceUpperBound`, but this time the inequality goes the other way:

```
  Reduce(m / 2, x + 1) - m;
  >= { ReduceLowerBound(m / 2, x + 1);
      assert x + 1 - 2 * (m / 2) <= Reduce(m / 2, x + 1); }
  x + 1 - 2 * (m / 2) - m;
```

Do you see the rest of the proof yet? The verifier does, as you can tell by the absence of complaints. The rest of the proof is not clear to me, so, for me 😊, let's do some more steps in detail. One thing that springs to mind is that doubling something that you first halve gets you back to where you started. We express that as follows:

```
x + 1 - 2 * (m / 2) - m;
== // arithmetic
x + 1 - m - m; // error: this proof step might not hold
```

But the verifier complains. Perhaps it needs more information from us. Let's debug this verification step by supplying a hint to the verifier. This will let us drill down in the matter to discover what's wrong or missing.

```
x + 1 - 2 * (m / 2) - m;
== { assert 2 * (m / 2) == m; } // wrong
x + 1 - m - m;
```

This assertion expresses the property that sprung to mind. But the verifier complains again. Why? Does it not have the extensive knowledge of arithmetic that we do? Sometimes, that will indeed turn out to be the case. But here, the verifier has every right to flag the assertion, because the condition we wrote down is about integers (not reals) and it holds only for even integers. If m is odd, then $2 \cdot (m/2)$ is one less than m . (If you want to see if the verifier knows this, you can write these assertions:

```
assert m % 2 == 0 ==> 2 * (m / 2) == m;
assert m % 2 == 1 ==> 2 * (m / 2) == m - 1;
```

You'll see that it does know this about integer arithmetic.) Where does this leave us for our proof? We change the assertion to reflect the fact that m may be larger than $2 \cdot (m/2)$. Also, since the term $2 \cdot (m/2)$ sits in a negative context (that is, there is a minus in front of it), replacing $2 \cdot (m/2)$ by m in our context may shrink the expression, so we use the relation operator $\geq$ in the calculation step:

```
x + 1 - 2 * (m / 2) - m;
```

```
>= { assert 2 * (m / 2) <= m; }
x + 1 - m - m;
```

By subtracting off 1 from this line, we make it even smaller and we arrive at the formula in the left-hand side of our proof goal. Here is the complete lemma and proof:

```
lemma {:induction false} ReduceLowerBound(m: nat, x: int)
  ensures x - 2 * m <= Reduce(m, x)
{
  if m == 0 {
  } else {
    calc {
      Reduce(m, x);
      == // def. Reduce
      Reduce(m / 2, x + 1) - m;
      >= { ReduceLowerBound(m / 2, x + 1);
          assert x + 1 - 2 * (m / 2) <= Reduce(m / 2, x + 1); }
      x + 1 - 2 * (m / 2) - m;
      >= { assert 2 * (m / 2) <= m; }
      x + 1 - m - m;
      > // reduce it further by 1
      x - 2 * m;
    }
  }
}
```

Note that we are free to use $>$ in the last step instead of $\geq$. Thus, the proof calculation actually proves $\text{Reduce}(m, x) > x - 2 * m$.

Exercise 5.8.

Maybe we can prove a tighter lower bound. Try changing the postcondition of the lemma to $x - 2 * m < \text{Reduce}(m, x)$. What happens?

5.6. Example: Commutativity of Multiplication

As an exercise, let us define a function that performs multiplication of natural numbers by repeated addition:

```

function Mult(x: nat, y: nat): nat {
  if y == 0 then 0 else x + Mult(x, y - 1)
}

```

If this function is really multiplication as we know it, then one of the properties we would expect of it is that it is *commutative*, that is, that its arguments can be swapped without altering the result. Let us verify that this is really so.

```

lemma {:induction false} MultCommutative(x: nat, y: nat)
  ensures Mult(x, y) == Mult(y, x)

```

There are several reasonable ways to proceed for the proof. Let us proceed by considering some cases that may be simple or interesting, leaving a longer and more general case for last. The simplest case to consider is that x and y happen to be equal, for then there is nothing else to be done:

```

{
  if x == y {

```

Another simple case that comes to mind is that one of the arguments, let's say x , is 0. Then, we have $\text{Mult}(x, y) == x + \text{Mult}(x, y-1)$ and we can complete the proof by induction on the other argument:

```

  } else if x == 0 {
    MultCommutative(x, y - 1);

```

Next, we may consider the equally simple case $y == 0$. The proof of it would be entirely symmetric to what we just did in the case of $x == 0$. But rather than doing just that special case, we can take care of all cases where y is less than x at the same time:

```

  } else if y < x {
    MultCommutative(y, x);

```

Notice that we are indeed entitled to invoking the induction hypothesis in this case—that is, making the recursive call $\text{MultCommutative}(y, x)$ —because Dafny is using the default termination-metric declaration

decreases x, y

(see Section 3.4) and the lexicographic pair y, x is strictly smaller than x, y , provided $y < x$.

By now, we have exhausted the simple cases we can think of, so it is time to roll up our sleeves and do the more general case. So, in the `else` branch of our cascaded `if` statement, we start a calculation from the left-hand side of our proof goal, `Mult(x, y)`. From there, the only sensible step is to apply the definition of `Mult`:

```
} else {  
  calc {  
    Mult(x, y);  
    == // def. Mult  
      x + Mult(x, y - 1);
```

Because this `Mult` term has arguments that are smaller—in the **decreases** ordering—than what we started with, we can apply the induction hypothesis:

```
== { MultCommutative(x, y - 1); }  
   x + Mult(y - 1, x);
```

We don't have many things we could do from here, but applying the definition of `Mult` once more is one of them:

```
== // def. Mult  
   x + y - 1 + Mult(y - 1, x - 1);
```

This looks like it was a good step, because we have obtained a situation that is fairly symmetric in x and y . The exception is the `Mult` term itself, but even it can be considered rather symmetric in x and y , since we can apply the induction hypothesis to obtain `Mult(x-1, y-1)`. Or can we? If we just tried to call `MultCommutative(y-1, x-1)` here, then the verifier would complain that we have not properly showed termination of the recursive call—the lexicographic pair $y-1, x-1$ is not necessarily smaller than x, y !

One way out would be to change the termination metric for the lemma we are trying to prove. For example, if we were to use

decreases $x + y$

then we justify making the call `MultCommutative(y-1, x-1)` here (see Exercise 5.9). Upon further thought, we realize we can actually proceed with the next step in the proof calculation in a different way, without changing the termination metric. The post-condition of the call we are trying to make, `MultCommutative(y-1, x-1)`, would tell us that the terms `Mult(y-1, x-1)` and `Mult(x-1, y-1)` are equal. Because everything is so symmetric, we can obtain this same equality by instead invoking `MultCommutative(x-1, y-1)`! Let's do that now.

```
== { MultCommutative(x - 1, y - 1); }  
    x + y - 1 + Mult(x - 1, y - 1);
```

From here on, the remaining steps are more or less the steps that got us here, but in reverse order and with the roles of x and y reversed:

```
== // def. Mult  
    y + Mult(x - 1, y);  
== { MultCommutative(x - 1, y); }  
    y + Mult(y, x - 1);  
== // def. Mult  
    Mult(y, x);  
}  
}  
}
```

Take a look at this proof. It has various interesting steps that demonstrate what makes something a proof. In particular, notice that we are allowed to chop up the proof task any way we'd like, as long as we can establish the proof goal in each case we consider. In our proof, the cases we considered were

- $x == y$
- $0 == x \neq y$
- $y < x$
- $0 < x < y$

Also, we are allowed to make a recursive call to the lemma with whatever parameters we want, as long as each call meets the precondition and decreases whatever termination metric we have chosen to use with the lemma. In our proof, the recursive calls were

- `MultCommutative(x, y - 1)` in a context where $0 < y$
- `MultCommutative(y, x)` in a context where $y < x$
- `MultCommutative(x, y - 1)`, again where $0 < y$
- `MultCommutative(x - 1, y - 1)` in a context where $0 < x$ and $0 < y$
- `MultCommutative(x - 1, y)` in a context where $0 < x$

Notice how the context of each call ensures that both arguments are non-negative, as required by the parameters' type `nat`, and that each call decreases the lexicographic pair x, y .

Exercise 5.9.

Declare lemma `MultCommutative(x, y)` with **decreases** $x + y$ and write a proof for it.

5.7. Example: Mirroring a Tree

Properties about inductive datatypes are well suited for inductive proofs. Let's try our hand at some of these.

5.7.0. Mirror is an involution

Here is a datatype definition for trees that we saw before in Section 4.5:

```
datatype Tree<T> = Leaf(data: T)
                  | Node(left: Tree<T>, right: Tree<T>)
```

and here is a function that produces the mirror image of a given tree (see Exercise 4.7):

```
function Mirror<T>(t: Tree<T>): Tree<T> {
```

```

match t
case Leaf(_) => t
case Node(left, right) => Node(Mirror(right), Mirror(left))
}

```

Note that to get the mirror image of a tree, it's not enough just to swap the left and right subtrees of a node. We must also compute the mirror image of each subtree.

If `Mirror` is applied twice, we get back the tree we started with. A function with that property is called an *involution*. Let's state and prove a lemma that `Mirror` is indeed an involution. The lemma is declared to be parametric over the type of the tree's payload, τ :

```

lemma {induction false} MirrorMirror<T>(t: Tree<T>)
  ensures Mirror(Mirror(t)) == t
{
  match t
  case Leaf(_) =>
    // trivial
  case Node(left, right) =>

```

To prove the lemma, we will consider the two kinds of possible trees separately, and so we write a `match` with those two cases. Since the body of a lemma is a list of statements, not an expression, we're actually using a *match statement* here. It looks and operates like the `match` expression, except that the body of each alternative is a list of statements, not an expression. Thus, it is legal (and here, desirable) to leave the `Leaf` case empty. This is similar to our use of `if` statements in the lemmas we just saw, where there is no reason to write anything when a proof is trivial.

For the `Node` case, we get an error saying the lemma's postcondition might not hold. Apparently, we need to help the verifier construct the proof. To that end, we start a proof calculation:

```

calc {
  Mirror(Mirror(Node(left, right)));
== // def. Mirror (inner)
  Mirror(Node(Mirror(right), Mirror(left)));

```

```

== // def. Mirror (outer)
   Node(Mirror(Mirror(left)), Mirror(Mirror(right)));

```

We start this calculation with the left-hand side of our proof goal, since that side is more complicated than the right-hand side and thus gives us more structure to help us think of what to do in the proof steps to come. Well, the left-hand side of the proof goal is `Mirror(Mirror(t))`, but the branch of the `match` statement that we're working in ensures that `t == Node(left, right)`.

In the first two steps, we simply apply the definition of `Mirror`: first to the inner `Mirror` call, which we can see is applied to a `Node` so we know which case of the definition of `Mirror` to use, and second to the outer `Mirror`, which by that time also has a visible `Node` as its argument. Now what?

We see that the two arguments to `Node` look like the left-hand side of the lemma we're trying to prove, so we'll call the lemma recursively. As we have seen before, this is done not in a comment, but inside curly braces, since this is something we need to tell the verifier. We might as well do both recursive calls in the same hint, so we get:

```

== { MirrorMirror(left); MirrorMirror(right); } // I.H.
   Node(left, right);
}
}

```

which completes the proof (since, again, we're working in the branch of the `match` statement where `t` is `Node(left, right)`). In a bit more detail, the last hint calls `Mirror-Mirror` on the subtrees `left` and `right` (and the “I.H.” in the comment stands for “induction hypothesis”). Since these are structurally included in `t`, the proofs of termination for the recursive calls go through. The postconditions of the two calls then give us the knowledge

```

Mirror(Mirror(left)) == left && Mirror(Mirror(right)) == right

```

These two equalities are what justify the rewrite in the last line.

5.7.1. Mirror preserves leaf count

Let's prove one more property of `Mirror`. This example uses a function that returns the number of leaves in a given tree:

```
function Size<T>(t: Tree<T>): nat {  
  match t  
  case Leaf(_) => 1  
  case Node(left, right) => Size(left) + Size(right)  
}
```

We prove that `Mirror` preserves the number of leaves in a tree. Since we're more used to proofs now, I will just show the whole thing at once:

```
lemma {:induction false} MirrorSize<T>(t: Tree<T>)  
  ensures Size(Mirror(t)) == Size(t)  
{  
  match t  
  case Leaf(_) =>  
  case Node(left, right) =>  
    calc {  
      Size(Mirror(Node(left, right)));  
    == // def. Mirror  
      Size(Node(Mirror(right), Mirror(left)));  
    == // def. Size  
      Size(Mirror(right)) + Size(Mirror(left));  
    == { MirrorSize(right); MirrorSize(left); } // I.H.  
      Size(right) + Size(left);  
    == // def. Size  
      Size(Node(left, right));  
    }  
}
```

The proof has the usual shape, applying the definition of various functions in the expression we're massaging and then, when we see subexpressions that look like the lemma we're trying to prove, invoking the induction hypothesis (here, twice). In this proof, we also apply a function definition after using the induction hypothesis. This is also common in inductive proofs. A general scheme is: break down the formula into parts that make the induction hypothesis applicable, then apply the induction hypothesis,

and lastly build the formula back up again to complete the proof. Of course, not all proofs have this structure or are as straightforward as the ones we've seen, but this general scheme gives a good place to start.

Exercise 5.10.

Replace the proof calculation in `MirrorSize` by just the calls to the induction hypothesis. Which version of the proof do you like best—the readable (but verbose) version that explains each proof step or the concise (but cryptic) version that just shows where the induction hypothesis is applied?

5.7.2. Variations on the proof calculation

Let me make two more remarks about the proof of `MirrorSize`.

One remark is that the last step applies the definition of `Size`, nonchalantly reversing the order of `left` and `right`. We could have made this explicit in the proof by preceding the last step by:

```
Size(right) + Size(left);  
== // + is commutative  
Size(left) + Size(right);
```

But we don't have to, because Dafny has built-in knowledge about arithmetic addition and is happy to supply this proof glue automatically. For other operators and operations, we may have to give such steps explicitly.

A second remark is that it would be easy to accidentally have written

```
Size(right) + Size(left);  
== // def. Size  
Size(Node(right, left));
```

in our last proof step above. As a proof step, it would also be okay. The conclusion of the proof calculation (that is, the

relation between the first and last line of the calculation) would then be:

```
Size(Mirror(Node(left, right))) == Size(Node(right, left))
```

but this is not our proof goal.

If you try this (and you should!), you'll find that Dafny is nevertheless happy with proof of the lemma. How come? As always, the verifier supplies various pieces of proof glue by itself. You don't have to understand exactly what it does or how it does it, but if you're curious, here's what it does for the example at hand:

One of the things the verifier does automatically is apply the definition of functions. More precisely, it does this once for every mention of a function in the program text. `Size(t)` occurs in the goal of the lemma, so the verifier will use the equality

```
Size(Node(left, right)) == Size(left) + Size(right)
```

`Size(Node(right, left))` occurs in the conclusion of the `calc` statement, so the verifier will use the equality

```
Size(Node(right, left)) == Size(right) + Size(left)
```

The verifier knows enough about arithmetic to see that the right-hand sides of these two equalities are the same, and therefore it also deduces

```
Size(Node(left, right)) == Size(Node(right, left))
```

So, this is why the verifier still completes the proof of the lemma, even if our `calc` statement only proves the property with `left` and `right` reversed in the last line.

In other cases, we might accidentally supply a valid but irrelevant hint in a proof and still not get a complaint from the verifier. In such cases, we may find ourselves confused (if we even notice), but Dafny is still doing the reasoning correctly by always trying to go above and beyond what we supply explicitly.

5.7.3. Summary

In this section, I defined two functions on trees. By stating and proving lemmas about these functions, we can show that the functions have some properties that we'd expect. This process gives us higher confidence that we have defined the functions correctly.

Exercise 5.11.

Function `ReverseColors` from Exercise 4.1 reverses the colors in a blue-yellow tree. State and prove (with automatic induction turned off) a lemma that shows `Reverse-Colors` to be an involution.

Exercise 5.12.

Define a function `LeafCount` that returns the number of blue or yellow leaves in any blue-yellow tree from Section 4.0. Then, prove (without automatic induction) that function `ReverseColors` from Exercise 4.1 preserves the `LeafCount` of any given tree.

Exercise 5.13.

Function `Oceanize` in Exercise 4.3 turns all tree nodes blue. State and prove (with automatic induction turned off) a lemma that shows `Oceanize` to be *idempotent*, that is, that applying the function twice is the same as applying it once.

Exercise 5.14.

Prove a lemma that shows

`BlueCount(t) <= BlueCount(Oceanize(t))`

for any blue-yellow tree `t` from Section 4.0, where function `BlueCount` is defined in Section 4.1 and function `Oceanize` is defined in Exercise 4.3. For your proof, turn off automatic induction and use a `calc` statement in the `Node` case.

5.8. Example: Working on Abstract Syntax Trees

In Section 4.6, I defined ASTs for simple expressions and two mutually recursive functions for evaluating expressions. Let's define two functions on such ASTs and prove properties about them.

As a reminder, here are the datatypes of the example:

```
datatype List<T> = Nil | Cons(head: T, tail: List<T>)
datatype Op = Add | Mul
datatype Expr = Const(nat)
               | Var(string)
               | Node(op: Op, args: List<Expr>)
```

5.8.0. Substitution is correct

A common operation in program analysis is to replace a variable in an AST by another expression. Let us consider one such substitution, replacing a variable by a constant. More exactly, for a given variable name n and natural number c , we replace in a given expression every subexpression $\text{Var}(n)$ with the expression $\text{Const}(c)$. For this purpose, we define two mutually recursive functions:

```
function Substitute(e: Expr, n: string, c: nat): Expr
{
  match e
  case Const(_) => e
  case Var(s) => if s == n then Const(c) else e
  case Node(op, args) => Node(op, SubstituteList(args, n, c))
}

function SubstituteList(es: List<Expr>, n: string, c: nat):
List<Expr>
{
  match es
  case Nil => Nil
  case Cons(e, tail) =>
    Cons(Substitute(e, n, c), SubstituteList(tail, n, c))
}
```

Function `Substitute` has no effect on constants and it replaces a `Var` expression if the variable it denotes has the name `n`. For an operator expression, it uses function `SubstituteList` to recursively perform the substitution in all argument expressions.

Let us prove that substitution has the expected effect on expression evaluation. That expected effect is that substituting the constant `c` for variable `n` has the same effect as changing the environment to one that maps `n` to `c`:

```
lemma EvalSubstitute(e: Expr, n: string, c: nat, env:
map<string, nat>)
ensures Eval(Substitute(e, n, c), env) == Eval(e, env[n :=
c])
{
  match e
  case Const(_) =>
  case Var(_) =>
  case Node(op, args) =>
    EvalSubstituteList(args, op, n, c, env);
}
```

The statement of this lemma uses the expression `env[n := c]`, which denotes the map that is like `env` except that it maps `n` to `c` (whereas `env` may be mapping `n` to something else or may not have any mapping for `n` at all).

The first two cases of the proof are simple and Dafny verifies these automatically. In the `Node` case, we need an auxiliary, mutually recursive lemma, following the basic structure of the mutually recursive functions `Eval` and `EvalList`:

```
lemma {:induction false} EvalSubstituteList(
  args: List<Expr>, op: Op, n: string, c: nat, env:
map<string, nat>)
ensures EvalList(SubstituteList(args, n, c), op, env)
  == EvalList(args, op, env[n := c])
{
  match args
  case Nil =>
  case Cons(e, tail) =>
    EvalSubstitute(e, n, c, env);
```

```

    EvalSubstituteList(tail, op, n, c, env);
}

```

The proof is not difficult. Given a list `Cons(e, tail)`, it (by mutual recursion) invokes the lemma `EvalSubstitute` on `e` and (by recursion) invokes the lemma `EvalSubstituteList` on `tail`.

Exercise 5.15.

Make the proof of `EvalSubstituteList` more explicit. In particular, replace the `Cons` case with a calculation that starts from the left-hand side of the lemma's proof goal and ends with its right-hand side. The hints of the calculation will include the two lemma calls shown above.

Exercise 5.16.

For any key `n` in a given environment `env`, show that substituting constant `env[n]` for variable `n` does not change the evaluation of the expression. That is, prove the following lemma:

```

lemma EvalEnv(e: Expr, n: string, env: map<string, nat>)
  requires n in env.Keys
  ensures Eval(e, env) == Eval(Substitute(e, n, env[n]), env)

```

Hint: Do the proof by induction (and using an auxiliary, mutually recursive lemma). Also, for reasons I explain in Section 10.2.4, you will need to include the following assertion, which says that a map is unchanged by updating it with a value it already has:

```

assert env == env[n := env[n]];

```

Exercise 5.17.

For any key `n` not in a given environment `env`, show that substituting constant `0` for variable `n` does not change the evaluation of the expression. That is, prove the following lemma:

```

lemma EvalEnvDefault(e: Expr, n: string, env: map<string,
nat>)

```

```
requires n !in env.Keys  
ensures Eval(e, env) == Eval(Substitute(e, n, 0), env)
```

Exercise 5.18.

Prove that substitution is *idempotent*, that is, that applying the operation twice is the same as applying it once. Stated in symbols, prove the following lemma:

```
lemma SubstituteIdempotent(e: Expr, n: string, c: nat)  
  ensures Substitute(Substitute(e, n, c), n, c)  
    == Substitute(e, n, c)
```

5.8.1. Optimization is correct

As the final example in this section, let us do a lemma that will require more work. There will be plenty of room for making small typos that will cause the lemma not to verify. As much as we'd like to think our programming and proving efforts are all straightforward and beautifully simple, we will always encounter problems that are more work than we expect. So, grab a snack to boost your patience as we get crackin'.

Defining the optimizing transformation

The function we focus on performs some peephole optimizations on expressions. In particular, it removes unit elements, that is, it removes any operation that would add by 0 or multiply by 1. These unit elements are returned by the following function:

```
function Unit(op: Op): nat {  
  match op case Add => 0 case Mul => 1  
}
```

(If we had declared this function earlier, we could have used it in the definition of `EvalList` in Section 4.6.) The optimization function is defined as follows:

```
function Optimize(e: Expr): Expr {  
  if e.Node? then  
    var args := OptimizeAndFilter(e.args, Unit(e.op));
```

```

    Shorten(e.op, args)
  else
    e // no change
}

```

We only optimize Node expressions, so I chose to use an **if** statement rather than a **match**. The plan for optimizing a node is to optimize each of its arguments and drop any argument that equals the operation's unit element. This is achieved by calling `OptimizeAndFilter`, whose result is placed in a local variable, `args`. We could now just return `Node(e.op, args)`, but we'll consider one more optimization: if we're left with no arguments, we'll just return the unit element, and if we're left with exactly one argument, we'll just return it. This further optimization is done in function `Shorten`.

Function `Shorten` is most easily implemented using a **match** with three cases:

```

function Shorten(op: Op, args: List<Expr>): Expr {
  match args
  case Nil => Const(Unit(op))
  case Cons(e, Nil) => e
  case _ => Node(op, args)
}

```

The first case matches the list `Nil`; that is, the list of length 0. The second case pattern restricts its matches to a `Cons` whose second argument is `Nil`; in other words, a list of length 1. The third case pattern uses a “catch all” underscore; since cases are considered in order, this case will match lists of length 2 or greater.

The final function of our optimization is `OptimizeAndFilter`:

```

function OptimizeAndFilter(es: List<Expr>, unit: nat):
List<Expr>
{
  match es
  case Nil => Nil
  case Cons(e, tail) =>
    var e', tail' := Optimize(e), OptimizeAndFilter(tail,
unit);
    if e' == Const(unit) then tail' else Cons(e', tail')
}

```

```
}
```

For a nonempty list of arguments, it optimizes the head of the list and recursively optimizes the tail of the list, placing these respective results in local variables `e'` and `tail'`. If `e'` denotes the unit element, only the optimized tail is returned; otherwise, `e'` is returned along with the optimized tail.

Exercise 5.19.

Find the recursive and mutually recursive calls in the definitions of `Optimize` and `OptimizeAndFilter` and show why those calls terminate.

Correctness of Optimize

What happens next is pretty typical: we will state and prove one lemma for each of the three functions. Let's start with the lemma that states the correctness of the top-level function, `Optimize`:

```
lemma OptimizeCorrect(e: Expr, env: map<string, nat>)
  ensures Eval(Optimize(e), env) == Eval(e, env)
```

No surprise here. We're saying that `e` and `Optimize(e)` evaluate to the same value, for any environment. For the proof, we use an `if` statement to mimic the structure of function `Optimize`. Then, we embark on a calculation, starting from the more complicated side of our proof goal. The first step of the calculation will apply the definition of `Optimize`, so it will be convenient to have a name for the long subexpression that denotes the call to `OptimizeAndFilter`. For this purpose, we introduce a local variable, in fact just like function `Optimize` does. Here is what we then have:

```
if e.Node? {
  var args := OptimizeAndFilter(e.args, Unit(e.op));
  calc {
    Eval(Optimize(e), env);
    == // def. Optimize
      Eval(Shorten(e.op, args), env);
```

This is where we apply whatever correctness lemma we are about to formulate for `Shorten`. As I had mentioned above, `Shorten(e.op, args)` is supposed to be semantically equivalent to `Node(e.op, args)`, so let's declare such a lemma:

```
lemma ShortenCorrect(op: Op, args: List<Expr>, env: map<string, nat>)  
  ensures Eval(Shorten(op, args), env) == Eval(Node(op, args), env)
```

Sidebar 5.2

As you're developing this proof in the IDE, you will probably want to provide the matching close braces in `OptimizeCorrect`, so that Dafny will not give you syntax errors but type-checking errors and verification errors. The lemma `ShortenCorrect` can be declared anywhere in the file, but it will probably feel natural to place it after `OptimizeCorrect`. Once you have done that, you can start using the lemma and the verifier will check that you're using the lemma correctly.

Our next step in the proof calculation in `OptimizeCorrect` is:

```
== { ShortenCorrect(e.op, args, env); }  
   Eval(Node(e.op, args), env);
```

Surely, the next step will be concerned with the transformation performed by `OptimizeAndFilter` (remember that we defined `args` as the result of `OptimizeAndFilter`). We know where we would like to end up, namely with the right-hand side of our proof goal. If we're lucky, we're just one step away, in which case we can design the correctness

lemma for `OptimizeAndFilter` by looking at what we need. Thus, we wrap up the proof of `OptimizeCorrect` as follows:

```

    == { OptimizeAndFilterCorrect(e.args, e.op, env); }
      Eval(Node(e.op, e.args), env);
    }
  }
}

```

which suggests the following definition for `OptimizeAndFilterCorrect`:

```

lemma OptimizeAndFilterCorrect(args: List<Expr>, op: Op,
                                env: map<string, nat>)
  ensures Eval(Node(op, OptimizeAndFilter(args, Unit(op))),
env)
    == Eval(Node(op, args), env)

```

Before we start the proof of `ShortenCorrect`, this is a good time to make sure that everything we have supplied (parsing, type checks, and) verifies. If we have made mistakes in our reasoning, in our entry of the functions and lemmas, or in our confidence in the verifier's ability to fill in proof glue, this would be a good time to take notice and to take corrective action.

Exercise 5.20.

With the default **decreases** clauses of the three lemmas we have defined, write out the proof obligations for termination in the body of `OptimizeCorrect`. Do these proof obligations hold?

Correctness of Shorten

Function `Shorten` seems very simple—so simple we'd expect the muscle of the verifier to take care of it automatically. Perhaps surprisingly, this is not so. It can sometimes be hard to predict when a proof can be done automatically, so let's practice some verification debugging.

Brimming with optimism, our first attempt at proving `ShortenCorrect` is to supply an empty body:


```
{  
}
```

Alas, this gives a complaint that the postcondition of the lemma may not hold. We might be trying to prove an incorrect lemma or it might just be that the verifier is unable to construct the proof automatically. To shed more light on the situation, we'll start writing the proof manually. As usual, the structure of the proof will match the structure of the definition of Shorten:

```
match args  
case Nil =>  
case Cons(a, Nil) => // error: postcondition might not  
                     // hold on this return path  
case _ =>
```

We can tell we've done *something* right, because the postcondition complaint we're now getting points to the second case. We deduce from this that the verifier completed the proof for the first case. If the verifier cannot prove something—in our situation at hand, the postcondition—it highlights only one trace leading to the error. So, we know that the first case goes through and the second case does not, and we don't yet know anything about the third case.

Let's finish up the proof of ShortenCorrect. In the Cons(a, Nil) case, we start a proof calculation and start expanding function definitions as usual:

```
calc {  
  Eval(Node(op, Cons(a, Nil)), env);  
== // def. Eval  
  EvalList(Cons(a, Nil), op, env);  
== // def. EvalList  
  var v0, v1 := Eval(a, env), EvalList(Nil, op, env);  
  match op  
  case Add => v0 + v1  
  case Mul => v0 * v1;  
== // def. EvalList  
  var v0, v1 := Eval(a, env), Unit(op);  
  match op
```

```

case Add => v0 + v1
case Mul => v0 * v1;

```

This looks a bit different than before, but only because the formulas got longer and involve a `let`-expression and a `match`-expression. Remember that, syntactically, each expression in a calculation is terminated by a semi-colon, not to be confused by the semi-colon that is part of the syntax of `let`-expressions.

We are almost there. Here we go:

```

== // substitute for v0, v1
  match op
    case Add => Eval(a, env) + Unit(op)
    case Mul => Eval(a, env) * Unit(op);
== // def. Unit in each case
  Eval(a, env);
}

```

This completes the proof of lemma `ShortenCorrect`. This was perhaps the first lemma we have seen where the verifier could not do the proof automatically (in this case because it does not automatically expand `EvalList` twice, as is required by the proof) and yet there is no use of induction.

Correctness of `OptimizeAndFilter`

For our final lemma, `OptimizeAndFilterCorrect`, let's see if whatever intuition we may have developed about induction so far lets us think about what the proof will look like, rather than more-or-less mechanically plunging into a `calc`. The proof goal of our lemma involves function `OptimizeAndFilter`, so let us inspect its definition. We see a `match` on the first argument, which is a list. Then, there is a mutually recursive call to `Optimize` and a recursive call to `OptimizeAndFilter`, followed by an `if-then-else` that seems straightforward. From this, we can guess that our proof will involve a call to the correctness lemmas for `Optimize` and `OptimizeAndFilter`, parameterized in a way that is analogous

to `OptimizeAndFilter`'s calls to these functions. This leads us to the following proof attempt:

```
{
  match args
  case Nil =>
  case Cons(e, tail) =>
    OptimizeCorrect(e, env);
    OptimizeAndFilterCorrect(tail, op, env);
}
```

Lo and behold! The verifier accepts this as the proof. Had we been less lucky in our guess, we would probably have resorted to some more detailed proof calculation, like those that we have seen before.

Summary of optimization example

This optimization example involved a datatype `Expr` that is recursive via the datatype `List`, the mutually recursive functions `Eval` and `EvalList`, the mutually recursive functions `Optimize` and `OptimizeAndFilter`, along with the helper functions `Unit` and `Shorten`. As the number of functions involved grows, so does the proof. The organization of the lemmas and their proofs tend to follow the structure of the functions involved. For example, we have seen that lemma `OptimizeCorrect` calls `ShortenCorrect` and `OptimizeAndFilterCorrect`, just like function `Optimize` calls `Shorten` and `OptimizeAndFilter`.

Exercise 5.21.

Write another optimization function, `PartiallyEvaluate`, that combines all the constant arguments of `Node` into one. Prove that this transformation does not change the semantic value of the expression.

5.9. Summary

In this chapter, we have seen several lemmas and their proofs.

A lemma in Dafny is just a ghost method. This means it has a pre- and postcondition specification. The precondition is the antecedent of the lemma and Dafny enforces that the lemma can only be invoked in situations where the precondition holds. The postcondition is the conclusion of the lemma. It is a proof obligation of the body of the lemma, and callers of the lemma get to assume the postcondition to hold upon return from the lemma call.

Lemmas can call lemmas recursively. This incurs a proof obligation that the recursion eventually terminates. A recursive call to a lemma corresponds to what logicians know as an appeal to the induction hypothesis. There is no need for us to tell Dafny what the induction hypothesis is, because it is just whatever is stated in the pre- and postconditions of the lemmas we call recursively.

We have seen that Dafny automatically supplies a lot of proof glue for us. Although we did not allow ourselves to rely on it in this chapter, Dafny also tries to find ways to automatically invoke lemmas recursively, which in effect often gives us the induction hypothesis for free. Dafny does not call *other* lemmas for us, however, so calls to helper lemmas and calls to mutually recursive lemmas are all up to us.

If a lemma concerns a function and that function is defined by cases, then chances are that the proof of the lemma will split control flow along those cases as well.

Whenever we needed to introduce more proof details, in some cases for the purpose of debugging an attempted verification, our main tool is the `calc` statement. It gives us step-by-step proof calculations where each step is justified by a hint. In most cases we have seen so far, these proof calculations are equality-preserving transformations, but

we have also seen a few cases where the proof steps use operators other than equality.

From the chapters so far—basics, formalities, termination, datatypes, and lemmas—we have covered the basis of both programs and proofs in Dafny. We have also seen plenty of interaction with the verifier. The basic idea is that if the verifier complains, we need to supply more of the proof ourselves. This process will either lead to the verifier accepting the proofs or to us realizing that what we're trying to do is not right.

Notes

The primary proof format I use in this book, calculations with hints *between* the steps, is attributed to Wim H. J. Feijen. It is used in many texts (see [\[41\]](#), [\[18\]](#), [\[11\]](#), [\[56\]](#), [\[9\]](#), [\[52\]](#), to mention but a few). Another common calculational format places the hints in the right margin. Though highly usable in our context, this is not the only form of structured proofs (e.g., see [\[72\]](#), [\[125\]](#)) and also not the only form of proofs supported in Dafny.

Proof calculations using `calc` statements [\[85\]](#) are also supported in F* [\[53\]](#), Liquid-Haskell [\[123\]](#), and Lean [\[75\]](#). The Isar proof language for Isabelle [\[102\]](#) features a rich notation for authoring human-readable proofs, including calculational proofs [\[15\]](#). The standard library of Agda [\[22\]](#) also provides a natural way to write calculational proofs.

When you get used to writing and reading `calc` statements in Dafny, you may find it verbose to have to repeat the same operator between lines. Dafny actually lets you omit the operator between two lines; if you omit it, it defaults to `==`. Using this abbreviation, the first proof calculation in Section 5.4 can be written as

```
calc {  
  5 * (x + 3);
```

```

    5 * x + 5 * 3;
    5 * x + 15;
}

```

If two lines need to be connected by a different operator, you can fill it in as needed. For example, the calculation involving $3*(x+n)$ in Section 5.4 can be written as follows:

```

calc {
    3*x + n + n;
    3*x + 2*n;
<=
    3*x + 3*n;
    3*(x + n);
}

```

You can even change the default operator to something other than `=`. This is indicated immediately after the `calc` keyword. For example, the same calculation involving $3*(x+n)$ can be written as

```

calc <= { // set default operator to be <=
    3*x + n + n;
==
    3*x + 2*n;
    3*x + 3*n;
==
    3*(x + n);
}

```

Being able to omit the operator is a welcome feature for advanced users, but it also makes the calculations look mysterious to beginners. In this book, I will continue to write the operator.

In the proof of a lemma, the Dafny verifier makes a simpleminded attempt at invoking the induction hypothesis as needed. This modest support of automatic induction is surprisingly effective [77]. Other verifiers have longstanding support of advanced heuristics for automatic induction, notably ACL2 [71] and its Boyer-Moore theorem-prover predecessors [23]. For more details, I refer you to a historical account of automated induction [91].

Part 1

Functional Programs

In the functional-programming style, programs are structured as mathematical functions that return values. Data is immutable and tends to be organized into data structures defined by inductive datatypes.

This Part of the book considers functional programs and how to reason about such programs. We have already seen the basic elements of functional programs in the previous chapters. Now, we put the elements to work to build and prove programs.

I will introduce the concepts of intrinsic versus extrinsic specifications, data-structure invariants, abstraction functions, and modules.

Chapter 6

Lists



Lists are the most common data structure in functional programs. It is therefore appropriate to devote this first chapter in this Part to programs on lists.

In Chapter 5, where we were learning to write proofs, we turned off Dafny's automatic induction. From now on, we're going to allow Dafny to apply its automatic induction. This will not eliminate the need for manually written proofs, but it will reduce our work.

6.0. List Definition

Throughout this chapter, I will use the following definition of a list datatype:

```
datatype List<T> = Nil | Cons(head: T, tail: List<T>)
```

The value `Nil` represents the empty list, whereas `Cons(x, xs)` represents the list that starts with the element `x` and then continues with the elements of list `xs`. The list type is parameterized by a type `T`, which is the type of every element in the list.

Here are four examples of lists. To the left is how you might write each list on a piece of paper. In the middle is the corresponding expression of type `List`, and to the right is a type for the list value.

informal notation	List expression	possible type
2, 5, 3	<code>Cons(2, Cons(5, Cons(3, Nil)))</code>	<code>List<nat></code>
<code>true, true</code>	<code>Cons(true, Cons(true, Nil))</code>	<code>List<bool></code>
"gimmie", "lists"	<code>Cons("gimmie", Cons("lists", Nil))</code>	<code>List<string></code>

The element `Nil` can have type `List<T>` for any type `T`. Note that all elements in a list have the same type. It is not legal write

```
Cons(true, Cons(3, Nil)) // error: elements must have common type
```

6.1. Length

Here is a function that returns the length of a list:

```
function Length<T>(xs: List<T>): nat {  
  match xs  
  case Nil => 0
```

```
    case Cons(_, tail) => 1 + Length(tail)
}
```

It says that an empty list has length 0 and other lists are 1 longer than their tail.

Here are two little reminders from previous chapters.

As a first reminder, the result type of `Length` is `nat`, which denotes the natural numbers, that is, the non-negative integers.

As a second reminder, the definition of `Length` uses a `match` expression to distinguish the two variants of lists. An alternative definition would use an `if-then-else` expression:

```
if xs == Nil then 0 else 1 + Length(xs.tail)
```

The choice between various control structures, like `match` versus `if`, is simply a matter of taste, but `match` is the typical choice when there are many constructors. Note that `tail` in the first definition is a bound variable introduced as part of the `match` case, whereas `tail` in the second definition names the destructor that gives us the second component of a `Cons`, as defined in the datatype declaration for `List`.

Exercise 6.0.

Define `Length` as above with a `match` and define a function `Length'` with the `if` above (but with the recursive call to `Length'`, not `Length`). Declare and prove a lemma that says that `Length` and `Length'` always return the same value.

The constructor `Cons` makes a list by putting an element in front of another list. Sometimes, we want just the opposite—to make a list by putting an element after another list, an operation commonly named `Snoc`. Because inductive lists are accessed from front to back, `Snoc` is computationally more expensive than `Cons`. Here is a definition:

```
function Snoc<T>(xs: List<T>, y: T): List<T> {
  match xs
  case Nil => Cons(y, Nil)
```

```
    case Cons(x, tail) => Cons(x, Snoc(tail, y))
}
```

To gain more confidence that we have defined a function as intended, it is always a good idea to check that it has some properties we'd expect. We can do that by stating and proving lemmas. For example, we expect `Snoc(xs, x)` to be a list 1 longer than `xs`:

```
lemma LengthSnoc<T>(xs: List<T>, x: T)
  ensures Length(Snoc(xs, x)) == Length(xs) + 1
{
}
```

Exercise 6.1.

Turn off automatic induction and write a proof for `LengthSnoc`.

6.2. Intrinsic versus Extrinsic Specifications

We are about to encounter an important distinction between two styles of writing specifications.

Let's define a function `Append` that returns the concatenation of two lists:

```
function Append<T>(xs: List<T>, ys: List<T>): List<T> {
  match xs
  case Nil => ys
  case Cons(x, tail) => Cons(x, Append(tail, ys))
}
```

There is a relation between `Append` and `Length`. Let's capture that relation in a lemma, which is proved automatically:

```
lemma LengthAppend<T>(xs: List<T>, ys: List<T>)
  ensures Length(Append(xs, ys)) == Length(xs) + Length(ys)
{
}
```

As we have seen before, introducing a lemma like this is commonly how we state and prove properties of functions.

The property stated by the lemma is said to be *extrinsic* to the function definition of `Append`, because the lemma is not part of the function declaration itself. This is not the only way to state and prove properties of functions. Another way is to define the function with a postcondition, like we do for methods. A property given in a function postcondition is called *intrinsic*, because it is part of the function declaration and is verified as part of checking the well-formedness of the function.

Here is an alternative definition of `Append` that intrinsically states the property that lemma `LengthAppend` states extrinsically:

```
function Append<T>(xs: List<T>, ys: List<T>): List<T>
  ensures Length(Append(xs, ys)) == Length(xs) + Length(ys)
{
  match xs
  case Nil => ys
  case Cons(x, tail) => Cons(x, Append(tail, ys))
}
```

To refer to the value returned by the function, the postcondition simply mentions the function invoked on its parameters, namely `Append(xs, ys)`.

Methods in Dafny are always opaque, so the only properties that are known about them are those stated in the method's specification. In other words, properties about methods are always intrinsic, never extrinsic. Functions, on the other hand, are transparent, so we have the choice of stating properties about them intrinsically or extrinsically.

One advantage of intrinsic specifications is that they get used with every application of the function. In contrast, an extrinsic lemma needs to be applied explicitly in order to make use of its property during verification. Having specifications get used automatically might seem like it would always be an advantage, but it is not. When verification conditions get large, having too much information around may overwhelm the verifier, resulting

in long verification times or even in failed verifications (because of reaching some internal resource bounds in the verifier). Therefore, common practice for functions is to state and prove properties extrinsically.

In some cases, writing a property intrinsically is not even an option. This occurs when the property mentions the function several times. For example, it would not be possible to state the property $\text{Mirror}(\text{Mirror}(t)) == t$ in the postcondition of function `Mirror` (Section 5.7). The reason is that the outer call to `Mirror` is a recursive call, so one would have to prove that its argument, namely $\text{Mirror}(t)$, is smaller than t . This is not possible for all t : Consider some particular t and t' where $t' == \text{Mirror}(t)$, which means that $t == \text{Mirror}(t')$. We certainly cannot argue both that t is smaller than t' and that t' is smaller than t . Other examples of properties ill-suited for intrinsic specifications are commutativity and transitivity.

The one situation where intrinsic specifications make sense and can be recommended occurs when the property is likely to be of concern for all clients of the function. It will be convenient in this chapter to think of the length of `Append` as falling in that category, so I will assume the definition of `Append` with the postcondition

$\text{Length}(\text{Append}(xs, ys)) == \text{Length}(xs) + \text{Length}(ys)$

Exercise 6.2.

Prove that `Snoc` is just a special case of `Append`, that is, that

$$\text{Snoc}(xs, y) == \text{Append}(xs, \text{Cons}(y, \text{Nil}))$$

6.2.0. Other properties of Append

It is often useful to know various algebraic properties about functions we define. For example, consider a binary function $\oplus$, which here I will write as an infix operator. A value L is called a *left unit element* of $\oplus$ if, for all x , $L \oplus x =$


```
requires Length(a) == Length(c)
requires Append(a, b) == Append(c, d)
ensures a == c && b == d
```

Prove this lemma, with and without automatic induction.

Exercise 6.6.

For an operator on the integers with a left unit L and a right unit R , prove that L and R are equal. We can set this up in Dafny as follows:

```
function F(x: int, y: int): int
```

```
const L: int
const R: int
```

```
lemma LeftUnit(x: int)
  ensures F(L, x) == x
```

```
lemma RightUnit(x: int)
  ensures F(x, R) == x
```

Here, the body-less function $F(x, y)$ denotes an arbitrary operation (which I had written as $x \oplus y$ above). I declared the names L and R as arbitrary constants; an alternative would be to declare them as body-less nullary functions. The two lemmas, which state properties about F , L , and R , are given as axioms, so go ahead and use them but don't try to prove them. State and prove a lemma that $L == R$.

Exercise 6.7.

A value L is called a *left zero element* of an operator $\oplus$ if, for all x , $L \oplus x = L$, and a value R is called a *right zero element* of $\oplus$ if, for all x , $x \oplus R = R$. If an operator has a left zero element L and a right zero element R , then these two elements are equal. State and prove this property for an arbitrary operator on the integers. Hint: See Exercise 6.6 for how to set this up.

6.3. Take and Drop

Function Append makes two lists into one. The functions Take and Drop do the reverse, they break a list into two parts.

```
function Take<T>(xs: List<T>, n: nat): List<T>
  requires n <= Length(xs)
{
  if n == 0 then Nil else Cons(xs.head, Take(xs.tail, n - 1))
}
```

```
function Drop<T>(xs: List<T>, n: nat): List<T>
  requires n <= Length(xs)
{
  if n == 0 then xs else Drop(xs.tail, n - 1)
}
```

Defined with the precondition $n \leq \text{Length}(xs)$ like this, these functions are rather picky about how many elements can be taken or dropped. The precondition allows the function bodies to access `xs.head` and `xs.tail` (which requires `xs.Cons?`) in the branch where $n \neq 0$.

Exercise 6.8.

Write more liberal versions of Take and Drop that do not have any precondition. Asked to take more elements than the length of the list, the liberal version returns the entire list. Asked to drop more elements than the length of the list, the liberal version return the empty list. Prove that if $n \leq \text{Length}(xs)$, then the liberal versions return the same value as the respective strict versions above.

Exercise 6.9.

With automatic induction turned off, prove the lemma in Exercise 6.8.

It is easy to prove the correspondence between Append, Take, and Drop:

lemma AppendTakeDrop<T>(xs: List<T>, n: **nat**)


```

requires n <= Length(xs)
ensures Append(Take(xs, n), Drop(xs, n)) == xs
{
}

```

```

lemma TakeDropAppend<T>(xs: List<T>, ys: List<T>)
  ensures Take(Append(xs, ys), Length(xs)) == xs
  ensures Drop(Append(xs, ys), Length(xs)) == ys
{
}

```

The proof of lemma TakeDropAppend makes use of the intrinsic property given by the postcondition of function Append.

Exercise 6.10.

Disable automatic induction and prove lemma AppendTakeDrop.

Exercise 6.11.

Disable automatic induction and prove lemma TakeDropAppend.

6.4. At

The elements of a list are ordered. We can ask for an element at a given index into the list.

```

function At<T>(xs: List<T>, i: nat): T
  requires i < Length(xs)
{
  if i == 0 then xs.head else At(xs.tail, i - 1)
}

```

The precondition of At requires *i* to be a proper index into the list *xs*. Note that this implies *xs*.Cons?, which justifies our use of the destructors *xs*.head and *xs*.tail in the body.

The element at position *i* in a list *xs* is the element preceded by *i* elements. So, this element is the head of the list you obtain by dropping *i* elements from *xs*. Let's try to formulate this property as a lemma:

```

lemma AtDropHead<T>(xs: List<T>, i: nat)
  requires i < Length(xs)
  ensures At(xs, i) == Drop(xs, i).head // error: .head
                                           // requires Cons?

```

It's clear we need a precondition for the lemma, because to talk about `At(xs, i)`, we must have `i < Length(xs)`. But trying to state the lemma as I did here produces a complaint that we might be access the head member of a list that does not have a head. In order for `Drop(xs, i).head` to be well-defined, we need to know that `Drop(xs, i)` returns a list of the `Cons` variant (that is, a non-`Nil` list). There are several ways out of this situation. The simplest, which is appropriate here, is to also include `Drop(xs, i).Cons?` as a proof goal of the lemma. This gives us

```

lemma AtDropHead<T>(xs: List<T>, i: nat)
  requires i < Length(xs)
  ensures Drop(xs, i).Cons? && At(xs, i) == Drop(xs, i).head
{
}

```

which is handled by Dafny's automatic induction.

We also connect `At` and `Append` in a lemma:

```

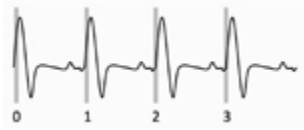
lemma AtAppend<T>(xs: List<T>, ys: List<T>, i: nat)
  requires i < Length(Append(xs, ys))
  ensures At(Append(xs, ys), i)
    == if i < Length(xs) then
      At(xs, i)
    else
      At(ys, i - Length(xs))
{
}

```

Sidebar 6.0

When we're talking about how many steps we have taken on a “journey”, it's clear what the numbers mean. For example, when you take your pulse, you start a 6-

second interval and count “zero, one, two, . . .”

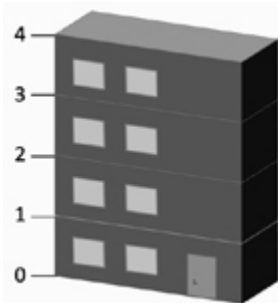


and then you multiply the final number you said by 10 to get your beats per minute. But what if we want to name the steps (or, in this case, heartbeats) themselves? For that, the world has more than one convention.

When you have lived for 19 years and someone asks your age, you say “19”. During the year following your birth, you're living your “year 0”.



If you live in an apartment 2 floors above the ground floor (street level), then in some countries (e.g., most of Europe), you'd press 2 in the elevator to get to your floor, whereas in other countries (e.g., the US), you'd press 3.



Here and elsewhere, the names given to elements of a list, sequence, or array start with 0, like heartbeats, ages, and floors in certain countries. Function `At(xs, i)`

therefore has the precondition $0 \leq i < \text{Length}(xs)$.

Exercise 6.12.

Turn off automatic induction and write a proof for `AtAppend`. Hint: Use a proof of the form

```
match xs
case Nil =>
case Cons(x, tail) =>
  if i == 0 {
    // calc...
  } else {
    // calc...
  }
```

(Are you happy about automatic induction?)

6.5. Find

We define a function `Find` that returns the position of an element in a list. If the element occurs more than once, the index to the first element is returned. If the element does not occur in the list, the length of the list is returned. Stated differently and more concisely, `Find(xs, x)` returns the length of the longest prefix of `xs` that does not contain `x`.

```
function Find<T(==)>(xs: List<T>, y: T): nat
  ensures Find(xs, y) <= Length(xs)
{
  match xs
  case Nil => 0
  case Cons(x, tail) =>
    if x == y then 0 else 1 + Find(tail, y)
}
```

Whereas our previous operations have been entirely parametric in the element type, function `Find` only makes sense if the element type is one whose values can be compared, that is, a type on which equality is defined. In Dafny, every type has an equality operation in ghost

contexts, so we can write `Find` as a ghost function. But not every type offers an equality comparison in compiled contexts. For example, you cannot compare two functions for equality in compiled code, for that may take infinite time to evaluate. If we want `Find` to be a compiled function, we must therefore restrict it to type parameters that support a compiled version of equality. This is done by decorating the type parameter with the suffix `(==)`, as I did above (and I'll say more about it in Section 9.4).

Exercise 6.13.

Remove the suffix `(==)` from `Find`'s type parameter. What error message do you get?

`Find(xs, x)` always returns a value between 0 and `Length(xs)`, inclusive, which is a property that most clients of `Find` are going to need. Therefore, we declare this property intrinsically, the lower bound by using result type `nat` and the upper bound by the declared `ensures` clause.

Many properties can be proved about the various list functions we have seen so far. Here are but four examples, where `xs` and `ys` have type `List<T>` for some type `T`, `x` has type `T`, and `i` has type `nat`:

```
lemma AtFind<T>(xs: List<T>, y: T)
  ensures Find(xs, y) == Length(xs) || At(xs, Find(xs, y)) == y
```

```
lemma BeforeFind<T>(xs: List<T>, y: T, i: nat)
  ensures i < Find(xs, y) ==> At(xs, i) != y
```

```
lemma FindAppend<T>(xs: List<T>, ys: List<T>, y: T)
  ensures Find(xs, y) == Length(xs)
    || Find(Append(xs, ys), y) == Find(xs, y)
```

```
lemma FindDrop<T>(xs: List<T>, y: T, i: nat)
  ensures i <= Find(xs, y) ==> Find(xs, y) == Find(Drop(xs, i),
y) + i
```

Exercise 6.14.

The proof goals of lemmas `AtFind` and `BeforeFind` call function `At`, which has a precondition. Explain for each of those calls how the precondition is met.

Exercise 6.15.

Prove lemma `AtFind`, with and without automatic induction.

Exercise 6.16.

Prove lemma `BeforeFind`, with and without automatic induction.

Exercise 6.17.

Prove lemma `FindAppend`, with and without automatic induction.

Exercise 6.18.

Prove lemma `FindDrop`, with and without automatic induction.

6.6. List Reversal

As a final list operation, we consider reversing the elements of a list. Here is one function that reverses a list:

```
function SlowReverse<T>(xs: List<T>): List<T> {  
  match xs  
  case Nil => Nil  
  case Cons(x, tail) => Snoc(SlowReverse(tail), x)  
}
```

This version may be easy to understand, but it would execute slowly. To be more precise, `SlowReverse(xs)` requires a number of steps proportional to the square of the length of `xs`. We will write a more efficient version and prove that it outputs the same value as this slow version. But first, let us at least prove that `SlowReverse(xs)` has the same length as `xs`.

```
lemma LengthSlowReverse<T>(xs: List<T>)  
  ensures Length(SlowReverse(xs)) == Length(xs)
```

Let's see if we can predict what is needed for its proof. Looking at the definition of `SlowReverse`, we expect the `Nil` case to immediately satisfy the lemma. In the `Cons` case, the induction hypothesis will give us that `SlowReverse(tail)` has the same length as `tail`, so if we only have the fact that `Snoc` increases the list length by 1, then we would be done. Let's try that:

```
{  
  match xs  
  case Nil =>  
  case Cons(x, tail) =>  
    LengthSnoc(SlowReverse(tail), x);  
}
```

Indeed, the verifier is able to fill in the rest of the details.

The trick to writing a more efficient version of list reversal is to add what is called an *accumulator* parameter that keeps track of the part that has been reversed so far. More precisely, the idea for our auxiliary function `ReverseAux(xs, acc)` is that it return `Append(xs', acc)`, where `xs'` is the list `xs` reversed. Here is the definition:

```
function ReverseAux<T>(xs: List<T>, acc: List<T>): List<T>  
{  
  match xs  
  case Nil => acc  
  case Cons(x, tail) => ReverseAux(tail, Cons(x, acc))  
}
```

To reverse `Cons(x, tail)` and append `acc` to it, we reverse `tail` and append `Cons(x, acc)` to it.

If we were writing a method, we would probably have recorded the intended effect of `ReverseAux` in an **ensures** clause, as an intrinsic specification. We could do that for `ReverseAux`. For one, we expect only one other caller of the auxiliary function, so our guiding principle of using an intrinsic specification applies: every caller is interested in this property. However, to verify the property, we need to

provide some manual guidance. I will show how to do that in Section 6.7. For now, let's state and prove the property extrinsically.

```
lemma ReverseAuxSlowCorrect<T>(xs: List<T>, acc: List<T>)
  ensures ReverseAux(xs, acc) == Append(SlowReverse(xs), acc)
{
  match xs
  case Nil =>
  case Cons(x, tail) =>
    calc {
      Append(SlowReverse(xs), acc);
      == // def. SlowReverse
        Append(Snoc(SlowReverse(tail), x), acc);
```

In the proof, we start with the right-hand side, since it is more complicated and may therefore give us better guidance. After applying the definition of `SlowReverse`, we may hope to apply the definition of `Snoc`. Since we are not in a situation where we know which of the two cases of `Snoc` applies, we would need to bring in the entire definition of `Snoc` into the calculation. Sometimes, this is what we have to do. In the current situation, however, we have the alternative of rewriting `Snoc` in terms of `Append`.

```
== { SnocAppend(SlowReverse(tail), x); }
   Append(Append(SlowReverse(tail), Cons(x, Nil)), acc);
```

Here, we can apply the associativity of `Append`.

```
== { AppendAssociative(SlowReverse(tail), Cons(x, Nil),
acc); }
   Append(SlowReverse(tail), Append(Cons(x, Nil), acc));
```

Appending `acc` to the singleton list `Cons(x, Nil)` will give a list that starts with `x` and ends with `acc`. We don't have a named lemma for this property, but it follows from two unrollings of the definition of `Append`. To document this proof step, we use an assertion in the hint.

```
== { assert Append(Cons(x, Nil), acc) == Cons(x, acc); }
   Append(SlowReverse(tail), Cons(x, acc));
```


The formula now has a shape like that in the lemma we are trying to prove, so we invoke the induction hypothesis.

```
== { ReverseAuxSlowCorrect(tail, Cons(x, acc)); }  
   ReverseAux(tail, Cons(x, acc));
```

The verifier is now happy with the proof. Actually, the verifier was happy even before our explicit invocation of the induction hypothesis, because that was taken care of by automatic induction. But to finish up a readable proof, we take one more step:

```
== // def. ReverseAux  
   ReverseAux(xs, acc);  
}  
}
```

We have now proved that `ReverseAux` does what we intended. What remains is to use it in a function `Reverse` and to prove that the end result is the same as applying `SlowReverse`.

```
function Reverse<T>(xs: List<T>): List<T> {  
  ReverseAux(xs, Nil)  
}
```

The proof of correctness makes use of the fact that `Nil` is a right unit element of `Append`, but is otherwise straightforward.

```
lemma ReverseCorrect<T>(xs: List<T>)  
  ensures Reverse(xs) == SlowReverse(xs)  
{  
  calc {  
    Reverse(xs);  
    == // def. Reverse  
       ReverseAux(xs, Nil);  
    == { ReverseAuxSlowCorrect(xs, Nil); }  
       Append(SlowReverse(xs), Nil);  
    == { AppendNil(SlowReverse(xs)); }  
       SlowReverse(xs);  
  }  
}
```

Since `Reverse` is more efficient than `SlowReverse`, we prefer it for compiled code. To make note of this in the source code, you can declare `SlowReverse` to be a ghost function.

The fact that `Reverse` and `SlowReverse` are the same, except for their execution time, means that they have the same properties. For example, we can now state the correctness lemma about `ReverseAux` in terms of `Reverse` instead of `SlowReverse`.

```
lemma ReverseAuxCorrect<T>(xs: List<T>, acc: List<T>)
  ensures ReverseAux(xs, acc) == Append(Reverse(xs), acc)
{
  ReverseCorrect(xs);
  ReverseAuxSlowCorrect(xs, acc);
}
```

Similarly, the fact that the reversal of a list does not change its length follows from the connection between `Reverse` and `SlowReverse` and the lemma we already proved about `SlowReverse` preserving length.

```
lemma LengthReverse<T>(xs: List<T>)
  ensures Length(Reverse(xs)) == Length(xs)
{
  ReverseCorrect(xs);
  LengthSlowReverse(xs);
}
```

Exercise 6.19.

Prove $\text{Length}(\text{ReverseAux}(xs, acc)) == \text{Length}(xs) + \text{Length}(acc)$, for any lists `xs` and `acc`.

Exercise 6.20.

Write an alternative proof for `LengthReverse` that uses the lemma in Exercise 6.19.

An interesting property to consider is the reversal of the concatenation of two lists. We start with a property that relates `ReverseAux` and `Append`.

```
lemma ReverseAuxAppend<T>(xs: List<T>, ys: List<T>, acc:
List<T>)
```

```

ensures ReverseAux(Append(xs, ys), acc)
    == Append(Reverse(ys), ReverseAux(xs, acc))
{
  match xs
  case Nil =>
    ReverseAuxCorrect(ys, acc);
  case Cons(x, tail) =>
}

```

Unusually, this proof requires a manual step in the Nil case and goes through automatically in the Cons case.

Exercise 6.21.

Mark lemma ReverseAuxAppend with `{:induction false}` and write a calculational proof for the Cons case.

Exercise 6.22.

Prove the following lemma:

```

lemma ReverseAppend<T>(xs: List<T>, ys: List<T>)
  ensures Reverse(Append(xs, ys))
    == Append(Reverse(ys), Reverse(xs))

```

Exercise 6.23.

Prove that Reverse is an involution. That is, prove that

```
Reverse(Reverse(xs)) == xs
```

Hint: Use lemmas from earlier in this section and from Exercise 6.22.

6.7. Lemmas in Expressions

We have used three proof-authoring constructs: inline assertions, proof calculations, and lemma calls. These are statements, and we used them in method bodies and lemma bodies to help discharge proof obligations. There are also proof obligations to establish the well-formedness of functions. So far, all such proof obligations for functions have gone through automatically, but there are plenty of functions where this is not the case.

Although general statements cannot be used in expressions in Dafny, proof-authoring statements can be. If s is an inline assertion, proof calculation, or lemma call, and E is an expression, then $s \ E$ is also an expression. For example, `assert 0 < x; 100 / x` is an expression.

The value of an expression $s \ E$ is simply E . The statement s does not affect the value of E . In fact, the s part is always ghost, so it is erased during compilation and is not present at run time. The proof-authoring statement s is used only to help establish any proof obligations that arise in E (and downstream of E in any enclosing expression). For example, the value of `assert 0 < x; 100 / x` is `100 / x`, but the proof obligation that the divisor x is non-zero follows from the assertion `assert 0 < x;`. Of course, the s part may itself entail proof obligations (like $0 < x$ when the statement is `assert 0 < x`) and these need to be established as well.

The next two examples make use of proof-authoring constructs in expressions.

6.7.0. Intrinsic specification of ReverseAux

In Section 6.6, we proved a lemma that `ReverseAux` correctly implements our intent. I argued that this property is a good candidate to be stated intrinsically, that is, as an **ensures** clause of the function. To do that, we declare `ReverseAux` as follows:

```
function ReverseAux<T>(xs: List<T>, acc: List<T>): List<T>
  ensures ReverseAux(xs, acc) == Append(SlowReverse(xs), acc)
{
  match xs
  case Nil => acc
  case Cons(x, tail) =>
    ReverseAux(tail, Cons(x, acc)) // error: cannot prove
    postcondition
}
```

Alas, the verifier is unable to prove this postcondition automatically from the given function body. More precisely, the verifier is unable to show that `ReverseAux(xs, acc)`, which in the **ensures** clause denotes the result value of the function invocation, equals

```
Append(SlowReverse(xs), acc)
```

To help the verifier along, let us insert a proof calculation before the recursive call, showing that this expression equals

```
Append(SlowReverse(xs), acc)
```

```
function ReverseAux<T>(xs: List<T>, acc: List<T>): List<T>
  ensures ReverseAux(xs, acc) == Append(SlowReverse(xs), acc)
{
  match xs
  case Nil => acc
  case Cons(x, tail) =>
    calc {
      Append(SlowReverse(xs), acc);
      == // def. SlowReverse
      Append(Snoc(SlowReverse(tail), x), acc);
      == { SnocAppend(SlowReverse(tail), x); }
      Append(Append(SlowReverse(tail), Cons(x, Nil)), acc);
      == { AppendAssociative(SlowReverse(tail), Cons(x, Nil),
acc); }
      Append(SlowReverse(tail), Append(Cons(x, Nil), acc));
      == { assert Append(Cons(x, Nil), acc) == Cons(x, acc); }
      Append(SlowReverse(tail), Cons(x, acc));
      == // postcondition of ReverseAux
      ReverseAux(tail, Cons(x, acc));
    }
  ReverseAux(tail, Cons(x, acc))
}
```

The proof calculation itself is identical to the one we wrote in the extrinsic lemma `ReverseAuxSlowCorrect` in Section 6.6, except that we do not include the last line of the previous calculation that appealed to the definition of `ReverseAux`.

Note that the proof calculation goes *before* the final expression of `ReverseAux`, even though we're using it to help prove the postcondition of the enclosing function. This is

because the syntax is always $s \ E$, where s is a proof-authoring statement and E is an expression. If you prefer to see it in the other order, you can use a `let` expression:

```
var r := ReverseAux(tail, Cons(x, acc));  
calc {  
  // proof calculation goes here  
}  
r
```

Regardless of which of these formulations you prefer, having the proof calculation in the body of function `ReverseAux` allows us to give the `ReverseAux` correctness property intrinsically. On the downside, it clutters up the body of the function. (Again, this does not affect the run-time behavior of the function, but it makes the function more difficult to read for a human). So, yet another alternative is to refactor the proof calculation into a separate lemma and to call this lemma from the function body of `ReverseAux`:

```
function ReverseAux<T>(xs: List<T>, acc: List<T>): List<T>  
  ensures ReverseAux(xs, acc) == Append(SlowReverse(xs), acc)  
{  
  match xs  
  case Nil => acc  
  case Cons(x, tail) =>  
    ReverseAuxHelper(xs, acc);  
    ReverseAux(tail, Cons(x, acc))  
}
```

```
lemma ReverseAuxHelper<T>(xs: List<T>, acc: List<T>)  
  requires xs.Cons?  
  ensures ReverseAux(xs.tail, Cons(xs.head, acc))  
    == Append(SlowReverse(xs), acc)  
  decreases xs, acc, 0  
{  
  var x, tail := xs.head, xs.tail;  
  calc {  
    Append(SlowReverse(xs), acc);  
    == // def. SlowReverse  
    Append(Snoc(SlowReverse(tail), x), acc);  
    == { SnocAppend(SlowReverse(tail), x); }  
  }
```

```

    Append(Append(SlowReverse(tail), Cons(x, Nil)), acc);
  == { AppendAssociative(SlowReverse(tail), Cons(x, Nil),
acc); }
    Append(SlowReverse(tail), Append(Cons(x, Nil), acc));
  == { assert Append(Cons(x, Nil), acc) == Cons(x, acc); }
    Append(SlowReverse(tail), Cons(x, acc));
  == // postcondition of ReverseAux
    ReverseAux(tail, Cons(x, acc));
}
}

```

Note that this transformation makes the function `ReverseAux` mutually recursive with `ReverseAuxHelper`—the function calls the lemma and the lemma uses the function (in both its postcondition and its body). This means that we need to think about termination. Since `ReverseAux` does not have an explicit **decreases** clause, Dafny by default uses the lexicographic tuple `xs, acc` (that is, the parameters of the function). It would do the same for the lemma, but then nothing is decreased when the function calls the lemma. Instead, we manually write a **decreases** clause for the lemma. It suffices to take any lexicographic triple that starts with `xs, acc`, since Dafny orders the longer tuple strictly below the shorter one (see Section 3.3.3).

Exercise 6.24.

The `acc` component of the **decreases** clauses of function `ReverseAux` and lemma `ReverseAuxHelper` is never used. Write explicit **decreases** clauses for `ReverseAux` and `ReverseAuxHelper` that prove termination and do not mention `acc`.

6.7.1. Lemma about `At` and `Reverse`

Here is one more example where the well-formedness of expressions requires a proof. We consider a lemma that states that the first-but-`i` element of a list `xs` is the same as the last-but-`i` element of `Reverse(xs)`:

Lemma `AtReverse<T>(xs: List<T>, i: nat)`

```

requires i < Length(xs)
ensures At(xs, i) == At(Reverse(xs), Length(xs) - 1 - i)
    // possible precondition violation on previous line

```

As written, the statement of this lemma is not well-defined. The problem is that `At` requires a proper index. The precondition of the lemma tells us that `i` is a proper index into `xs`, but without knowing more about the length of `Reverse(xs)`, it is not clear that `Length(xs) - 1 - i` is a proper index into `Reverse(xs)`. We could add this as another postcondition:

```

lemma AtReverse<T>(xs: List<T>, i: nat)
  requires i < Length(xs)
  ensures Length(xs) - 1 - i < Length(Reverse(xs))
  ensures At(xs, i) == At(Reverse(xs), Length(xs) - 1 - i)

```

akin to the extra postcondition conjunct we added to lemma `AtDropHead` in Section 6.4. Here, however, this extra conjunct only clutters up the statement of the lemma. Instead, to convince Dafny that the call to `At` does satisfy its precondition, we invoke the lemma `LengthReverse` and we do so in the postcondition expression itself!

```

lemma AtReverse<T>(xs: List<T>, i: nat)
  requires i < Length(xs)
  ensures (LengthReverse(xs);
    At(xs, i) == At(Reverse(xs), Length(xs) - 1 - i))

```

(For backward compatibility with specification clauses that end with a semi-colon, Dafny allows an optional semi-colon at the end of specification clauses. An unfortunate consequence of this support is that we need to wrap parentheses around a specification clause that calls a lemma, like this postcondition.)

In the body of this lemma, we would like to use a proof calculation that starts with `At(Reverse(xs), Length(xs) - 1 - i)`. Again, this bites us with a complaint that we may be passing an improper index to `At`. Our new call to `LengthReverse` in the postcondition gives information about `Length(Reverse(xs))`, which is used in the rest of the

postcondition, but this does not make the information available in the body. Instead, we need to invoke the lemma in the body as well.

Soon, it will also be convenient to have names for the head and tail of `xs`, so we introduce local variables to hold these. As I remarked with the introduction of function `At` in Section 6.4, note that the precondition `i < Length(xs)` implies `xs.Cons?`. Okay, so we start the body of `AtReverse` like this:

```
{
  var x, tail := xs.head, xs.tail;
  LengthReverse(xs);
  calc {
    At(Reverse(xs), Length(xs) - 1 - i);
  == // def. Reverse
    At(ReverseAux(xs, Nil), Length(xs) - 1 - i);
  == // def. ReverseAux
    At(ReverseAux(tail, Cons(x, Nil)), Length(xs) - 1 - i);
  == { ReverseAuxSlowCorrect(tail, Cons(x, Nil)); }
    At(Append(SlowReverse(tail), Cons(x, Nil)), Length(xs) - 1
- i);
  == { ReverseCorrect(tail); }
    At(Append(Reverse(tail), Cons(x, Nil)), Length(xs) - 1 -
i);
```

These steps in the calculation are fairly straightforward. With all this talk about the well-formedness of expressions, you may be wondering why Dafny does not complain about the precondition of `At` in the second and subsequent lines of the calculation. This is because when Dafny checks the well-formedness of a line in a calculation, it assumes the well-formedness of the previous line. This usually works so smoothly that you probably did not even think about that it had to be checked.

The current line in our proof performs an `At` of an `Append`. The `AtAppend` lemma tells us that such an expression can be rewritten into an `At` of either the first or second list argument to `Append`. However, at this stage in the proof, we

cannot tell which part it will be, so we must keep the **if-then-else** expression:

```

== { AtAppend(Reverse(tail), Cons(x, Nil), Length(xs) - 1 -
i); }
  if Length(xs) - 1 - i < Length(Reverse(tail)) then
    At(Reverse(tail), Length(xs) - 1 - i)
  else
    At(Cons(x, Nil), Length(xs) - 1 - i -
Length(Reverse(tail)));

```

Let's not be discouraged by the length of this expression. Instead, let us simplify parts of it to make it easier to understand:

```

== { LengthReverse(tail); }
  if Length(xs) - 1 - i < Length(tail) then
    At(Reverse(tail), Length(xs) - 1 - i)
  else
    At(Cons(x, Nil), Length(xs) - 1 - i - Length(tail));
== // arithmetic, using Length(xs) == Length(tail) + 1 and 0
<= i
  if 0 < i then
    At(Reverse(tail), Length(tail) - 1 - (i - 1))
  else
    At(Cons(x, Nil), 0);

```

Alright, now things would get messier if we just continued carrying the **if** forward in the calculation. Instead, let us close the calculation and break into two cases, one of which is proved automatically by the verifier:

```

}
if 0 < i {

```

Guarded by this condition, we would like to continue our calculation from the line `At(Reverse(tail), Length(tail) - 1 - (i - 1))`. A new calculation will not remember the well-formedness of steps of some other calculation, so we need to invoke the `LengthReverse` lemma once more, this time on `tail`.

```

  LengthReverse(tail);
  calc {
    At(Reverse(tail), Length(tail) - 1 - (i - 1));

```

Well, the good news is that we are almost done. This expression has the form of the lemma we are trying to prove, so we invoke the induction hypothesis, and that just about wraps 'er up.

```
== { AtReverse(tail, i - 1); }
   At(tail, i - 1);
== // def. At
   At(xs, i);
}
}
```

In stating and proving this lemma, we had to make several appeals to lemma `LengthReverse`. We could have avoided the need for these lemma calls if we had instead written the `LengthReverse` property as a postcondition of function `Reverse`. Writing such intrinsic specifications is tempting, and sometimes they are a good idea. They do simplify our proof task by automatically introducing more facts for the verifier. However, as I have mentioned, in more complex situations, these additional facts may just overwhelm the verifier. So, use intrinsic specifications sparingly.

Exercise 6.25.

In Section 6.4, we came across an initial problem with stating lemma `AtDropHead`, because the expression in the proof goal was not well-defined by itself. State a lemma

```
lemma DropLessThanEverything<T>(xs: List<T>, i: nat)
  requires i < Length(xs)
  ensures Drop(xs, i).Cons?
```

and use this lemma in the proof goal of the original attempt at writing lemma `AtDropHead`. Then, prove both `DropLessThanEverything` and your new `AtDropHead`.

6.8. Eliding Type Arguments

In this section, we have seen numerous examples of functions and lemmas about lists. In all of these examples, the list element type was abstract. That is, our functions and lemmas were parameterized by a type, which in this chapter I happened to always name τ . Since the specific type of this payload is irrelevant, you may have the feeling that type signatures like

```
function ReverseAux<T>(xs: List<T>, acc: List<T>): List<T>
```

are overly verbose or cluttered. When you don't have any need for listing the type τ by itself, Dafny has an elision rule that saves you from mentioning τ at all. I will use two forms of this elision rule in other parts of the book. The elision rule appears to be unique to Dafny, so I will explain those forms here.

Recall that, in Dafny, a parameterized type like `List` must always be instantiated with some type argument; for example, `List<int>`, or `List<T>` where τ is type parameter. In the type signature of a function, method, or lemma, any parameterized type that is mentioned without any type arguments (that is, without the angle brackets where such type arguments would be given) gets filled in automatically with the list of type parameters of the enclosing function, method, or lemma. This form of the elision rule means that the type signature of function `ReverseAux` above can be given as

```
function ReverseAux<T>(xs: List, acc: List): List
```

Also, the type signature of function `Length` (Section 6.1) can be given as

```
function Length<T>(xs: List): nat
```

and the type signature of function `At` (Section 6.4) can be given as

```
function At<T>(xs: List, i: nat): T
```

The other form of the elision rule is that, if the function, method, or lemma's type parameter is not mentioned

anywhere else, then it, too, can be elided. So, the type signatures of `ReverseAux` and `Length` can be abbreviated further as

```
function ReverseAux(xs: List, acc: List): List
```

and

```
function Length(xs: List): nat
```

However, this second elision rule is of no further help for function `At`, because `At` needs to mention the type parameter as its result type.

Exercise 6.26.

Make use of the elision rules to shorten the signatures of `Append`, `Take`, `AtAppend`, `Find`, `AtDropHead`, and `Reverse`.

6.9. Summary

In this chapter, I introduced the datatype `List`, which is used throughout functional programming. Since operations on `List` are defined as recursive functions, lemmas that state properties of the operations are a good match for inductive proofs.

A property of a function that is declared as a postcondition of the function is called an *intrinsic* specification. It is an integral part of the function and has to be proved as part of showing the well-formedness of the function.

A property of a function that is declared separately from the function, as a lemma, is called an *extrinsic* specification. It is proved by looking at the definition of the function and can easily state properties that involve multiple invocations of the function, such as associativity.

In the chapter, I also showed the use of an accumulator to obtain a more efficient implementation of a function; the use of the type characteristic `(==)`, which says a type

supports compiled equality operations (more about that in Section 9.4); various techniques for making sure expressions are well-defined, including the use of proof-authoring statements in expressions; and Dafny's rule for eliding type parameters in the signatures of functions, methods, and lemmas.

Notes

In Dafny, the well-definedness of a function or method's postcondition must follow from the precondition, without regard to the body of the function or method. This is done differently in SPARK [43] and OpenJML [105], where for a function or method with a body, the well-definedness of the postcondition is checked on exit from the body.

Sidebar 6.1

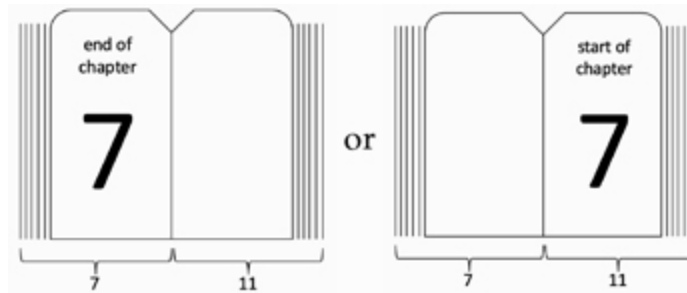
Suppose you have a book with 18 chapters and you open it so that 7 chapters fall to the left and the other 11 chapters fall to the right, as depicted here:



If you have read to the place where you have opened the book, then you have read 7 chapters.

Let's use numbers to name the chapters. What shall we name the chapters visible in this spread? Certainly, it will be convenient to use "7" to name either the chapter on the left or the chapter on the right. But which one?

If we print “7” in this spread, then the two choices come down to writing



As for me, I like seeing the name of the chapter the moment I start reading it, not the moment I have finished it. Therefore, in this book, when you're about to start chapter 7, you know there are 7 chapters before it.

Chapter 7

Unary Numbers



In the days of cave dwellers, the need for doing large calculations was not big. Therefore, some lines drawn on a wall or some pine cones collected in a pile could address the demand for recording numbers. For even slightly more sophisticated applications, we find problems with such unary numbering systems. With our improved state today, we would see a problem not only in the difficulty to distinguish recording the number zero from not having recorded anything in particular, but also in the inefficient ways of doing arithmetic on such numbers. Nevertheless, in

this chapter I will take us back to the primitive unary numbers to define various functions and prove some lemmas.

7.0. Basic Definitions

The unary numbers are defined inductively:

```
datatype Unary = Zero | Suc(pred: Unary)
```

As you can see, a unary number is either zero or the successor of some other unary number. When we define a new datatype, we don't always have the luxury to link it with another equivalent representation. Instead, we often have to prove that what we have defined has the properties we want. But in the case of the unary numbers, we can directly give conversion functions to and from the natural numbers:

```
function UnaryToNat(x: Unary): nat {  
  match x  
  case Zero => 0  
  case Suc(x') => 1 + UnaryToNat(x')  
}
```

```
function NatToUnary(n: nat): Unary {  
  if n == 0 then Zero else Suc(NatToUnary(n-1))  
}
```

From these definitions, it is easy to prove that the unary numbers and natural numbers are in one-to-one correspondence with each other. That is, we prove that the conversion function in each direction is injective.

```
lemma NatUnaryCorrespondence(n: nat, x: Unary)  
  ensures UnaryToNat(NatToUnary(n)) == n  
  ensures NatToUnary(UnaryToNat(x)) == x  
{  
}
```

Exercise 7.0.

Nonsensically change the bodies of `UnaryToNat` and `NatToUnary` in a way that breaks injectivity, that is, that renders lemma `NatUnaryCorrespondence` false.

In this chapter, I will use these functions to prove that the operations we define on unary numbers do indeed correspond to the analogous operations on the natural numbers, for example, that `Add` corresponds to `+`. These properties will be given in lemmas with names of the form ... `Correct`. The whole point of this chapter is to get to work with `Unary` rather than Dafny's built-in `nat`, so let's not use these conversion functions or the `NatUnaryCorrespondence` lemma for anything else.

7.1. Comparisons

Our first operation on unary numbers is a comparison.

```
predicate Less(x: Unary, y: Unary) {  
  y != Zero && (x.Suc? ==> Less(x.pred, y.pred))  
}
```

This definition says that `x` is (strictly) less than `y` exactly when `y` is non-zero and, furthermore, if `x` is also non-zero, then the predecessor of `x` is less than the predecessor of `y`. As mentioned in Section 4.2, of course you could also define `Less` using a `match` expression with two cases. To my taste, I find the definition I gave here to be more straightforward.

Exercise 7.1.

Using a `match` expression, define a predicate `Less'`. Prove that it is equivalent to `Less` above.

Exercise 7.2.

Define `Less` without a `match`, without using the discriminator `Suc?`, and without using the implication operator, `==>`.

To prove to ourselves that we have defined the correct ordering, we write the following lemma, which is proved automatically:

```
lemma LessCorrect(x: Unary, y: Unary)
  ensures Less(x, y) <==> UnaryToNat(x) < UnaryToNat(y)
{
}
```

Note that the lemma postcondition uses the equivalence operator on booleans (“if and only if”, whose binding power is lower than that of any other binary operator). Alternatively, we could have used the equality operator of booleans:

```
  ensures Less(x, y) == (UnaryToNat(x) < UnaryToNat(y))
```

However, unlike `==`, operator `<==>` has lower operator precedence than `<`, so we save a pair of parentheses in the first formulation.

Next, we show that `Less` has the property of being *transitive*. Here is one way to state and prove the property:

```
lemma LessTransitive(x: Unary, y: Unary, z: Unary)
  requires Less(x, y) && Less(y, z)
  ensures Less(x, z)
{
}
```

Often when a lemma has an antecedent, like `Less(x, y) && Less(y, z)` in this case, it is better to write the antecedent in a **requires** clause. This way, if a client fails to establish the precondition when calling the lemma, the call gets flagged with an error. However, for the transitivity property, we may want to allow a client to use the property in a different direction, for example to establish `!Less(y, z)` from `Less(x, y)` and `!Less(x, z)`. Therefore, it is better to state this lemma without a precondition and instead with an implication in the postcondition.

```
lemma LessTransitive(x: Unary, y: Unary, z: Unary)
  ensures Less(x, y) && Less(y, z) ==> Less(x, z)
```

Sidebar 7.0

For many operators in Dafny, you can tell its relative operator precedence from its width: a wider operator has lower precedence. The 4-characters-wide operator `<==>` has the lowest precedence. The 3-characters-wide operators `==>` and `<==` bind more tightly, and the 2-characters-wide operators `&&` and `||` bind more tightly yet. Comparison operators bind even more tightly, though their widths are both 1 and 2 characters: `==`, `!=`, `<=`, `<`, `>`, and `>=`. Single-character operators bind tighter than these, first the additive operators `+` and `-`, then the multiplicative operators `*`, `/`, and `%`, and finally, most strongly, the bitwise operators `&`, `|`, and `^`.

Perhaps surprisingly, this version of the lemma is not proved automatically. It is not necessary to understand why automation fails. Instead, let us just write the proof, which will illustrate good learning points. We are proving `Less(x, z)` under the assumption of

`Less(x, y) && Less(y, z)`

To introduce this assumption in our proof, we use a conditional statement:

```
{  
  if Less(x, y) && Less(y, z) {
```

and write the proof in the then branch. As usual in programming, we get to assume the stated condition to hold upon entering the then branch. In the else branch, there is nothing for us to prove, since the lemma holds trivially if `Less(x, y) && Less(y, z)` does not hold.

To prove `Less(x, z)` inside the `then` branch, we note that the definition of `Less(x, z)` consists of two conjuncts. So, to prove `Less(x, z)`, we'll have to prove both of those conjuncts. The first conjunct is `z != Zero`, which follows directly from the same conjunct in the definition of `Less(y, z)`. The other conjunct is an implication, so we continue the proof with yet another `if` statement:

```
    if x.Suc? {
```

By the definition of `Less`, the outer `if` guard implies that `y` and `z` are non-zero, and the inner `if` guard gives us the additional assumption that `x` is non-zero. This means we can call the lemma recursively on the predecessors of `x`, `y`, and `z`. In other words, we invoke the induction hypothesis on these predecessors:

```
        LessTransitive(x.pred, y.pred, z.pred);
    }
}
```

Evidently, Dafny can fill in the remaining proof glue, so this completes the proof.

Exercise 7.3.

State and prove a lemma that `Less` is trichotomous, that is, that *exactly one* of the following holds: `Less(x, y)`, `x == y`, `Less(y, x)`. (Remember here and elsewhere in this chapter, do not let your proof depend on the conversion of `Unary` to `nat`.)

Exercise 7.4.

Define a predicate `Below(x, y)` as `Less(x, y) || x == y`. State and prove that `Below` is a *total order*, that is, that it is reflexive, transitive, antisymmetric (`Below(x, y) && Below(y, x)` implies `x == y`), and total (`Below(x, y) || Below(y, x)`).

7.2. Addition and Subtraction

The addition of two unary numbers is defined by a recursive function `Add`:

```
function Add(x: Unary, y: Unary): Unary {  
  match y  
  case Zero => x  
  case Suc(y') => Suc(Add(x, y'))  
}
```

This function corresponds to addition on natural numbers, as the following lemma shows.

```
lemma AddCorrect(x: Unary, y: Unary)  
  ensures   UnaryToNat(Add(x, y)) == UnaryToNat(x) +  
  UnaryToNat(y)  
{  
}
```

The definition of `Add(x, y)` proceeds by considering different cases for `y`. This immediately gives us the property `Suc(Add(x, y)) == Add(x, Suc(y))`. In other words, `Suc` distributes over the second argument of `Add`. `Suc` also distributes over the first argument of `Add`, but this property requires an inductive proof:

```
lemma SucAdd(x: Unary, y: Unary)  
  ensures Suc(Add(x, y)) == Add(Suc(x), y)  
{  
}
```

We could instead have chosen to define `Add(x, y)` by considering different cases for `x`. We would then get the distribution of `Suc` over the first argument of `Add` for free, and we would need to give an inductive proof for the property that `Suc` distributes over the second argument to `Add`.

The definition of `Add` immediately gives us that `Zero` is a right unit element of `Add` (that is, `Add(x, Zero) == x` for any `x`). Here is a proof that `Zero` is also a left unit element:

```
lemma AddZero(x: Unary)  
  ensures Add(Zero, x) == x  
{
```

}

Exercise 7.5.

Turn off automatic induction and write proofs for (a) AddCorrect, (b) SucAdd, and (c) AddZero.

Exercise 7.6.

Prove that Add is associative:

$\text{Add}(\text{Add}(x, y), z) == \text{Add}(x, \text{Add}(y, z))$

Exercise 7.7.

Prove that Add is commutative: $\text{Add}(x, y) == \text{Add}(y, x)$.

Exercise 7.8.

Prove the following lemma, which states a combination of associativity and commutativity:

lemma AddCommAssoc(x: Unary, y: Unary, z: Unary)
 ensures Add(Add(x, y), z) == Add(Add(x, z), y)

We also define a function for subtracting unary numbers. Since we do not have any negative unary numbers, we define Sub(x, y) only when x is at least as large as y.

```
function Sub(x: Unary, y: Unary): Unary
  requires !Less(x, y)
{
  match y
  case Zero => x
  case Suc(y') => Sub(x.pred, y')
}
```

```
lemma SubCorrect(x: Unary, y: Unary)
  requires !Less(x, y)
  ensures UnaryToNat(Sub(x, y)) == UnaryToNat(x) -
UnaryToNat(y)
{
}
```

Exercise 7.9.

Write a suitable precondition for the following lemma. Then prove the lemma.

```

lemma AddSub(x: Unary, y: Unary)
  // requires ?
  ensures Add(Sub(x, y), y) == x

```

7.3. Multiplication

For unary numbers, multiplication is defined by repeated addition.

```

function Mul(x: Unary, y: Unary): Unary {
  match x
  case Zero => Zero
  case Suc(x') => Add(Mul(x', y), y)
}

```

Analogous to what we did for the other operation, we prove that `Mul` corresponds to `*` on the natural numbers. This time, the proof requires a little assistance from us.

```

lemma MulCorrect(x: Unary, y: Unary)
  ensures   UnaryToNat(Mul(x, y))   ==   UnaryToNat(x)   *
UnaryToNat(y)
{
  match x
  case Zero =>
  case Suc(x') =>
    calc {
      UnaryToNat(Mul(x, y));
      == // def. Mul
      UnaryToNat(Add(Mul(x', y), y));
      == { AddCorrect(Mul(x', y), y); }
      UnaryToNat(Mul(x', y)) + UnaryToNat(y);
      // Dafny can take it from here on
    }
}

```

7.4. Division and Modulus

Next up, as we define division and modulus (remainder), we encounter several teaching points. For unary numbers, division and modulus are done by repeated subtraction. Since the two operations perform similar computations, we

will package them both in one function. The function we define will therefore return a *pair*. For any types A and B, the type representing pairs with one A value and one B value is denoted (A, B) in Dafny.

```
function DivMod(x: Unary, y: Unary): (Unary, Unary)
```

Our convention will be that the first component of the returned pair is the division of x by y and the second component is their modulus. The notation for constructing a pair of d and m is simply (d, m) . The destructors for the two respective components are called 0 and 1 , so if r is the pair returned by `DivMod(x, y)`, then $r.0$ denotes the division of x by y and $r.1$ denotes their modulus.

We define our function only for non-zero divisors:

```
requires y != Zero
```

A straightforward definition of `DivMod` would seem to be the following:

```
{  
  if Less(x, y) then  
    (Zero, x)  
  else  
    var r := DivMod(Sub(x, y), y); // cannot prove termination  
    (Suc(r.0), r.1)  
}
```

However, Dafny is not automatically able to prove that the recursive call to `DivMod` terminates. Let's look at two different ways to help the verifier along to prove termination.

7.4.0. Termination via natural numbers

To prove termination, we need to use a termination metric that we can show to decrease with every recursive call. We have that `DivMod(x, y)` calls `DivMod(Sub(x, y), y)`, so our only choice is to show that something about the first argument decreases. Interpreted as natural numbers, we have that `UnaryToNat(x) - UnaryToNat(y)` is less than `UnaryToNat(x)`, since

y is known to be non-zero. To say that we want to use the natural number corresponding to x as our termination metric, we declare our own **decreases** clause:

decreases UnaryToNat(x)

To prove termination of the recursive call, we now need to prove

`UnaryToNat(x) > UnaryToNat(Sub(x, y))`

Since `UnaryToNat` results in a **nat**, this condition is

`UnaryToNat(Sub(x, y)) < UnaryToNat(x)`

Alas, this is not done automatically, either.

To help the verifier some more, we can invoke the `SubCorrect` lemma, which says the left-hand side of this inequality is equal to

`UnaryToNat(x) - UnaryToNat(y)`

We insert this lemma call just before the `let` expression:

```
SubCorrect(x, y);  
var r := DivMod(Sub(x, y), y);
```

The lemma call fits nicely on this line by itself in the source text. If we wanted to, we could move it even closer to the place where it is needed:

```
var r := (SubCorrect(x, y); DivMod(Sub(x, y), y));
```

Note the need for the extra parentheses in this case, so that the parser does not just take `SubCorrect(x, y)` as the right-hand side of the `let` binding.

In fact, we can move the lemma call even closer yet. For example:

```
var r := DivMod(SubCorrect(x, y); Sub(x, y), y);
```

The important thing is to give the verifier the lemma's information before it checks the precondition and termination of the call to `DivMod`, which happens after the evaluation of the call's parameters. It is generally desirable to place a lemma close to where it is needed, but there is

no reason to go overboard, so I like the placement on a line by itself before the `let`.

7.4.1. Termination via structural inclusion

Since we happened to have the correspondence with natural numbers in this case, it was easy enough to use `UnaryToNat` and `SubCorrect` in the termination proof. But this is not the only way we can do it. Another way is to argue more directly that the datatype value returned by `Sub(x, y)` is structurally smaller than `x`. Stated differently, if you think of unary numbers as being scribbled as lines on the wall of a cave, then `Sub(x, y)` is a shorter sequence of lines than `x`. Dafny knows that the datatype parameters to a constructor are structurally smaller than what the constructor returns. To get it to understand that this property also holds transitively, we need to prove a lemma:

```
lemma SubStructurallySmaller(x: Unary, y: Unary)
  requires !Less(x, y) && y != Zero
  ensures Sub(x, y) < x
{
}
```

In Dafny, the `<` operator is overloaded to work on various argument types. In the post-condition of this lemma, it works on datatype values and thus stands for structural inclusion. This lemma is proved automatically by induction.

All we have to do now is call the lemma before the `let` expression in `DivMod`. Here is the full definition of `DivMod`:

```
function DivMod(x: Unary, y: Unary): (Unary, Unary)
  requires y != Zero
{
  if Less(x, y) then
    (Zero, x)
  else
    SubStructurallySmaller(x, y);
    var r := DivMod(Sub(x, y), y);
    (Suc(r.0), r.1)
}
```

In Section 6.2, I discussed intrinsic versus extrinsic ways of stating and proving properties of functions. Termination of functions (and of methods, too) must always be done intrinsically in Dafny. That is, you are not allowed to declare a function, ignore termination, and try to prove termination later in a separate lemma—termination must be proved in the function declaration itself (possibly with the help of additional lemmas, as illustrated by `SubCorrect` and `SubStructurallySmaller` above).

7.4.2. Let expressions with patterns

While we are tweaking the definition of `DivMod`, this gives me the opportunity to introduce another convenient language feature. The left-hand side of a `let` expression can be a pattern, much like the cases of a `match`. For example, if $F(e)$ is a function that is known to return a non-zero unary number, then we can use the pattern `Suc(w)` as the left-hand side of the `let`. This will bind `w` to the parameter of the `Suc` value returned by $F(e)$. In other words, it will compute the value of $F(e)$, then “shave off” the outermost `Suc` of that value, and then bind what remains to `w`. For this to work, it must be known that $F(e)$ does indeed return a `Suc`, not `Zero`. So, after the following two `let` bindings

```
var r := F(e);  
var Suc(w) := F(e);
```

we have `r == Suc(w)` and `r.pred == w`.

A `pair` is a built-in datatype with special syntax. In particular, its sole constructor is written with just parentheses. Using a pattern in the `let` expression, we can therefore write the last two lines of `DivMod` as follows:

```
var (d, m) := DivMod(Sub(x, y), y);  
  (Suc(d), m)
```

Finally, note the subtle difference between this `let`, which binds `d` and `m` to the different components of the one `pair`

returned by `DivMod`, and an incorrect expression

```
var d, m := DivMod(Sub(x, y), y); // error
```

where there are *two* left-hand sides (`d` and `m`) and only one right-hand side (the call to `DivMod`).

7.4.3. Correctness of `DivMod`

I'll wrap up this chapter with a proof that `DivMod` is really correct. But instead of proving correctness via the natural numbers, let us prove that division and modulus satisfy the properties we expect. These properties are as follows, for any natural number `a` and positive integer `b`:

```
(a / b) * b + (a % b) == a  
(a % b) < b
```

(All of the parentheses here are unnecessary. I added them to emphasize where the division and modulus operations are.) Stated as a lemma about the unary numbers, we have:

```
lemma DivModCorrect(x: Unary, y: Unary)  
  requires y != Zero  
  ensures var (d, m) := DivMod(x, y);  
    Add(Mul(d, y), m) == x &&  
    Less(m, y)
```

Note how it was convenient to use a `let` expression in the postcondition of this lemma, so that we can give names to what is returned by `DivMod`.

We start the proof by introducing local variables (with the same names as the bound variables inside the postcondition) to stand for the unary numbers we are going to prove the properties of.

```
{  
  var (d, m) := DivMod(x, y);
```

Next, we will follow the two cases in the definition of `DivMod`. Although it is not necessary for the proof, we can use an **assert** statement to remind ourselves of what `DivMod` returns in this case.

```

if Less(x, y) {
  assert d == Zero && m == x; // since (d, m) == DivMod(x, y)

```

Here, we immediately have Less(m, y). The rest of this case is a simple calculation:

```

  calc {
    Add(Mul(d, y), m) == x;
  == // d, m
    Add(Mul(Zero, y), x) == x;
  == // def. Mul
    Add(Zero, x) == x;
  == { AddZero(x); }
  true;
}

```

For the other case, we start by introducing names for what is returned by the recursive call to DivMod, and we use an **assert** to remind ourselves of what DivMod(x, y) returns in this case.

```

} else {
  var (d', m') := DivMod(Sub(x, y), y);
  assert d == Suc(d') && m == m'; // since (d, m) ==
DivMod(x, y)

```

There are several ways to proceed. We usually start a proof calculation with the more complicated side of the proof goal, because this gives us more structure that may help us in figuring out the proof steps. Let us do it differently in this case and start with a formula at the opposite end of the spectrum of structurally rich formulas, the literal **true**.

```

calc {
  true;

```

This seems rather masochistic, but after one more step we will see where this is going: the next step is to introduce the properties gained by the induction hypothesis.

```

==> { SubStructurallySmaller(x, y);
      DivModCorrect(Sub(x, y), y); }
      Add(Mul(d', y), m) == Sub(x, y) && Less(m, y);

```

Note that to prove termination of the recursive call to DivModCorrect, we first invoke the SubStructurallySmaller

lemma in the calculation hint. You can also tell from the calculation connective in the left margin that we are going for a proof of the form `true ==> proof goal`. The rest of the proof is straightforward, using the lemmas proved in Exercises 7.9 and 7.8:

```

==> // add y to both sides
    Add(Add(Mul(d', y), m), y) == Add(Sub(x, y), y) &&
    Less(m, y);
== { AddSub(x, y); }
    Add(Add(Mul(d', y), m), y) == x && Less(m, y);
== { AddCommAssoc(Mul(d', y), m, y); }
    Add(Add(Mul(d', y), y), m) == x && Less(m, y);
== // def. Mul, d
    Add(Mul(d, y), m) == x && Less(m, y);
}
}
}

```

7.5. Summary

This chapter has mostly been a playground to practice proving properties of functions on a simple datatype. But we also came across several important points:

For many recursive functions, there's a choice in how to decompose the parameters for the recursive call and how to build the result from the recursive call. This choice will make some properties of the function trivial to prove, while similar properties may require a proof by induction. For example, if you define `Add` by decomposing its first parameter, the definition trivially shows that `Zero` is a left unit of `Add`, whereas you'll need an inductive proof (using a lemma) to show that `Zero` is a right unit. If you instead define `Add` by decomposing its second parameter, you'll need induction for the left-unit property, whereas the right-unit property follows trivially from the definition. It's a good idea to be aware of the trade-offs with this asymmetry. We have also seen this asymmetry with `Append`

in Section 6.2.0, and we'll see related issues in Sections 12.2 and 12.3.

In this chapter, we also learned more about termination. While many termination proofs are automatic, even to the degree that specifying the termination metric is automatic, we sometimes have to write termination proofs manually. In Dafny, the termination metric for a function is usually (the lexicographic tuple consisting of) the function's list of parameters, but by manually supplying a **decreases** clause, you can make the termination metric be any list of values computed from the parameters. It is common in verification tools (including Dafny) to require the termination of a function to be proved intrinsically; that is, termination must be established as part of the function definition itself.

Functions defined on datatypes often use structural inclusion to prove termination. While direct structural inclusion is automatically considered by the verifier, you may need to prove a lemma to obtain transitive structural inclusion.

To prove a property like $P \implies Q$, you often write a proof of the form

```
if P {  
  // prove Q here  
}
```

Notes

Our `UnaryToNat` and `NatToUnary` functions demonstrate that there is a one-to-one correspondence between the values of type `Unary` and the natural numbers. The idea to represent the natural numbers by an inductive datatype like `Unary`, with constructors `Zero` and `Suc`, follows directly from an *axiomatization* of the natural numbers that was developed by Giuseppe Peano. Therefore, the set of unary numbers—

more precisely, the Zero/Suc representation of the natural numbers—is often known as Peano numbers.

Because the Peano numbers are described by such a simple inductive datatype, they are popular as a subject of formal proofs.

Chapter 8

Sorting



An often occurring theme in computer science is that of sorting. This chapter is concerned with the correctness of two sorting algorithms that operate on lists. The datatype `List` and various functions on lists, such as `Length`, are those from Chapter 6.

In this chapter, I will start to separate the main algorithmic functions, which we eventually want to compile, from the supporting functions used only in specifications. These specification-only functions will be declared as **ghost**.

8.0. Specification

We start by considering the specification of sorting. Obviously, we want the output of our sorting routines to be ordered.

8.0.0. Sorted

For a list of integers to be sorted means that the elements are in ascending numerical order. When we work with lists, this property is most easily expressed by saying that any two consecutive elements of the list are ordered.

```
ghost predicate Ordered(xs: List<int>) {  
  match xs  
  case Nil => true  
  case Cons(x, Nil) => true  
  case Cons(x, Cons(y, _)) => x <= y && Ordered(xs.tail)  
}
```

There are three cases. The first two apply to the empty list and to singleton lists, respectively, which are always sorted. The use of the constructor `Nil` inside the `Cons` pattern in the second case is a nested pattern, which matches when `xs` is a `Cons` whose tail is `Nil`. If the case applies, then `x` is bound to the head of `xs` for use in the body of the `case` construct. The third case also uses a nested pattern. It matches when both `xs` and the tail of `xs` are `Cons` values, and if the case applies, then `x` and `y` are bound to `xs.head` and `xs.tail.head`, respectively. As usual in patterns, the underscore in the position of `xs.tail.tail` says that we don't care to introduce any bound variable to stand for that subexpression. The body of the third case evaluates to `true` when the first two elements of `xs` are ordered and the elements of `xs.tail` are pairwise ordered.

If you feel that a definition, like `Ordered` here, is complicated or non-obvious, it is a good idea to verify that it has some properties that we expect it to have. For a list

that satisfies `Ordered`, we expect that any element earlier in the list is below any element later in the list. We can state this precisely using the list function `At` (from Section 6.4), which retrieves a particular element of the list. That leads us to the following lemma:

```
lemma AllOrdered(xs: List<int>, i: nat, j: nat)
  requires Ordered(xs) && i <= j < Length(xs)
  ensures At(xs, i) <= At(xs, j)
```

The proof of this lemma does not go through automatically, so let's help the verifier along by writing the proof. By the definition of `At`, if `i` is non-zero (which, by `AllOrdered`'s precondition, implies that `j` is non-zero, too), then our proof goal is equivalent to

```
At(xs.tail, i - 1) <= At(xs.tail, j - 1)
```

so we start our proof of `AllOrdered` as follows:

```
{
  if i != 0 {
    AllOrdered(xs.tail, i - 1, j - 1);
```

If `i` is 0, then we need to argue that going down the list until we find the element at position `j` gives us an element that is no smaller than the head of `xs`. This is trivial if `j` is also 0:

```
  } else if i == j {
```

If `j` is non-zero, then the property follows from

```
At(xs, 0) <= At(xs, 1) <= At(xs, j)
```

We get the condition `At(xs, 0) <= At(xs, 1)` from the fact that the list is ordered. The condition `At(xs, 1) <= At(xs, j)` is equivalent to `At(xs.tail, 0) <= At(xs.tail, j - 1)`, which means we can obtain it from the induction hypothesis by calling the lemma recursively. Our proof of `AllOrdered` thus comes to an end like this:

```
    } else {
      AllOrdered(xs.tail, 0, j - 1);
    }
  }
```

We can now feel reasonably confident that we have stated the property `Ordered` correctly.

8.0.1. Same elements

Not every function that outputs a list of ordered elements is a sorting routine. We also want to describe some relation between the input elements and the output elements. For one, we want the output to consist of exactly the *same elements* as in the input, but arranged in a possibly different order. We define this property by saying: for any element p , the number of occurrences of p in the input equals the number of occurrences of p in the output. In other words, this is saying that the *multiset* of elements in the input equals the multiset of elements in the output. A multiset (sometimes known as a *bag*) is like a *set*, but the multiset keeps track of the multiplicity of each element.

Parenthetically, let me mention that this same-elements property of sorting routines is often described by saying: the output is a *permutation* of the input. If we tried to formalize what it means for one list to be a permutation of another, we would likely get ourselves into a complicated mess. So, while “permutation” mentally suggests a possible reordering, it really just means that the elements are the same.

To specify the same-elements property of a sorting routine, we could define a function that counts the number of occurrences of a given value p in a list:

```
ghost function Count(xs: List<int>, p: int): nat {  
  match xs  
  case Nil => 0  
  case Cons(x, tail) =>  
    (if x == p then 1 else 0) + Count(tail, p)  
}
```

Our sorting specification would then include a property like `Count(xs, p) == Count(MySortingFunction(xs), p)`

for every p .

However, instead of using `Count`, I will do something slightly more general, as I'll describe next.

Exercise 8.0.

Generalize the definition of `Count` beyond integer lists. What type characteristic is needed of the list's element type?

8.0.2. Stability

A property that a sorting algorithm may or may not have is *stability*. This property is of interest when the elements to be sorted have a key that is not the entire element. For example, if you are sorting a list of (student name, exam score) pairs by their exam score, then a stable sort would output any two students with the same exam score in the same order as these students were listed in the input. In other words, a stable sort preserves the order of records with the same key.

Suppose you wanted to output a list of (student name, exam score) pairs in the order of exam scores, and within each group of the same exam score, by the student names in alphabetical order. You can accomplish this by first sorting the list using the name as the key and then, using a stable sort, sorting by exam score.

In this chapter, I only present sorting for lists of integers, so the sorting key of an element is the element itself. In this case, stability does not matter; for example, you cannot distinguish the list `Cons(4, Cons(4, Nil))` from the list `Cons(4, Cons(4, Nil))`. 🤖 Nevertheless, I will define functions that can easily be adapted for lists where keys are computed from elements and where stability may matter. The following function will pick out the elements equal to (or, think: that have the key) p :

```
ghost function Project(xs: List<int>, p: int): List<int> {
```

```

match xs
case Nil => Nil
case Cons(x, tail) =>
  if x == p then Cons(x, Project(tail, p)) else Project(tail,
p)
}

```

Using this `Project` function, we can specify stability by the following *stability equation*: for every p ,

$$\text{Project}(xs, p) == \text{Project}(\text{MySortingFunction}(xs), p)$$

Note that the equality in this equation is on lists, not numbers like in the similar equation with `Count` above. Importantly, note also that stability (defined in this way) implies the same-elements property (cf. [79]). Therefore, even though stability does not make a difference in this chapter, I will phrase the same-elements property like we would have stated stability, namely as the stability equation above.

Exercise 8.1.

For any p and any two lists that, after projection to p , are equal, state and prove a lemma that says the two lists have the same occurrence count of p .

In the rest of this chapter, I define two classic sorting algorithms, Insertion Sort and Merge Sort. We will prove for each algorithm that its output is sorted and that the output has the same elements as the input. Both Insertion Sort and Merge Sort are in fact stable sorts, but we can't tell since I'm only using lists of integers. Nevertheless, as I mentioned above, I will specify the same-elements property using function `Project`.

8.1. Insertion Sort

The idea of Insertion Sort is simple. To sort a nonempty list, sort its tail and then insert its head into the appropriate

position. Here is a function that does that:

```
function InsertionSort(xs: List<int>): List<int> {  
  match xs  
  case Nil => Nil  
  case Cons(x, tail) => Insert(x, InsertionSort(tail))  
}
```

As for the insertion into a list, if the list has a head that is smaller than the element to be inserted, insert the element into tail. Otherwise, prepend the element to the list.

```
function Insert(y: int, xs: List<int>): List<int> {  
  match xs  
  case Nil => Cons(y, Nil)  
  case Cons(x, tail) =>  
    if y < x then Cons(y, xs) else Cons(x, Insert(y, tail))  
}
```

That's all there is. Let us verify that we wrote it correctly.

We start by proving the ordered property of the output. As is common, we will have one lemma per function in the implementation. These will look something like this:

```
lemma InsertionSortOrdered(xs: List<int>)  
  ensures Ordered(InsertionSort(xs))
```

```
lemma InsertOrdered(y: int, xs: List<int>)  
  ensures Ordered(Insert(y, xs))
```

The proof of the first calls the second, just like the InsertionSort function calls Insert:

```
{  
  match xs  
  case Nil =>  
  case Cons(x, tail) =>  
    InsertOrdered(x, InsertionSort(tail));  
}
```

The verifier is happy with this proof. Since Insert is a recursive function, we expect the InsertOrdered lemma to be recursive as well. And since the recursion is pretty simple, we may also have good hopes that Dafny will verify InsertOrdered automatically. If you supply the empty proof

body, {}, you will find this is not the case. To debug the situation, let's start filling in the proof manually. Following our standard technique of following the structure of the body of the function we are trying to prove something about, we write the following structure of cases in the lemma:

```
{
  match xs
  case Nil =>
  case Cons(x, tail) =>
    if y < x { // cannot prove postcondition on this return
path
    } else {
    }
}
```

This reveals that even when `Insert` returns `Cons(y, xs)`, in the then branch of the conditional, the verifier fails to prove the postcondition. But, of course! There is no reason to think that `Cons(y, xs)` is sorted just because `y < xs.head`—the rest of `xs` may contain elements in any order. We have thus discovered that the lemma needs to know more about the given list, `xs`. We state this property, namely `Sorted(xs)`, as a precondition of the lemma. Now, the lemma holds and, furthermore, Dafny verifies it automatically:

```
lemma InsertOrdered(y: int, xs: List<int>)
  requires Ordered(xs)
  ensures Ordered(Insert(y, xs))
{
}
```

Note that the function `Insert` itself does not need any precondition. Indeed, `Insert` is happy to insert `y` after the prefix of elements of `xs` that are smaller than `y`. The need for the precondition `Sorted(xs)` arises only when we are trying to prove that `Insert` outputs a sorted list.

One thing remains: proving the same-elements property of the sorting functions. To state this property, we need to talk about any value of the list's element type. We can do

that by parameterizing our lemmas with another parameter, call it p , like I did in the stability equation in Section 8.0.2. Since we state and prove the lemmas for any such p , we obtain a proof of the desired same-elements property.

```
lemma InsertionSortSameElements(xs: List<int>, p: int)
  ensures Project(xs, p) == Project(InsertionSort(xs), p)
lemma InsertSameElements(y: int, xs: List<int>, p: int)
  ensures Project(Cons(y, xs), p) == Project(Insert(y, xs), p)
```

The first lemma says that an element p occurs in $\text{InsertionSort}(xs)$ exactly as much as it does in the input xs , and the second lemma says that p occurs in $\text{Insert}(y, xs)$ exactly as much as it does in $\text{Cons}(y, xs)$.

To prove the first lemma, we simply call the second, just like function InsertionSort calls Insert :

```
{
  match xs
  case Nil =>
  case Cons(x, tail) =>
    InsertSameElements(x, InsertionSort(tail), p);
}
```

The proof of $\text{InsertSameElements}$ is automatic:

```
{
}
```

8.2. Merge Sort

The Merge Sort algorithm is faster than the Insertion Sort algorithm for large inputs. Asymptotically, for an input of length n , Merge Sort runs in time $\mathcal{O}(n \log n)$, whereas Insertion Sort requires time $\mathcal{O}(n^2)$. The difference in speed comes from the fact that Insertion Sort reduces its list size by just 1 with each recursive call, whereas Merge Sort divides its input in half with each recursive call. In more detail, Merge Sort first splits the given list into two halves,

then sorts each of those halves, and finally merges the two sorted halves.

8.2.0. The implementation

To split the input in two parts whose lengths are as equal as possible, it is convenient to know the length of the input. Rather than computing this length with each recursive call, we will compute it at the beginning and then keep track of the lengths of the halves. The bulk of the work of our Merge Sort will therefore be done in an auxiliary function that takes the length as a parameter. Here is the top-level function:

```
function MergeSort(xs: List<int>): List<int> {  
  MergeSortAux(xs, Length(xs))  
}
```

The auxiliary function is declared as follows:

```
function MergeSortAux(xs: List<int>, len: nat): List<int>  
  requires len == Length(xs)
```

If the list has length 0 or 1, it is already sorted, so we just return the input:

```
{  
  if len < 2 then  
    xs  
  else
```

Otherwise, we need two more functions, `Split` and `Merge`. Let's write those functions next and then return to `MergeSortAux`.

Function `Split` splits a list into a prefix and a suffix. We let the function take a parameter that indicates the desired length of the prefix.

```
function Split(xs: List, n: nat): (List, List)  
  requires n <= Length(xs)
```

The result type of `Split` is a pair of lists, as indicated by the type `(List, List)`. (See Section 6.8 if you wonder what

happened with the type argument to `List`, which is irrelevant to `Split`.) To give all callers of `Split` information about the two lists returned, we write an intrinsic specification in an **ensures** clause of the function:

```
ensures var (left, right) := Split(xs, n);
    Length(left) == n &&
    Length(right) == Length(xs) - n &&
    Append(left, right) == xs
```

This postcondition uses a `let` expression to give names to the two components of the pair that is being returned. The body of `Split` is defined like this:

```
{
  if n == 0 then
    (Nil, xs)
  else
    var (l, r) := Split(xs.tail, n - 1);
    (Cons(xs.head, l), r)
}
```

Function `Merge(xs, ys)` is like `Append(xs, ys)` (Section 6.2), except that `Append` always takes its next elements from `xs`, provided `xs` is nonempty. In contrast, `Merge` takes the next element from `ys` if it is smaller. To set this up, we match not on the list `xs`, but on the pair of lists `(xs, ys)`:

```
function Merge(xs: List<int>, ys: List<int>): List<int>
{
  match (xs, ys)
  case (Nil, Nil) => Nil
  case (Cons(_, _), Nil) => xs
  case (Nil, Cons(_, _)) => ys
  case (Cons(x, xtail), Cons(y, ytail)) =>
    if x <= y then
      Cons(x, Merge(xtail, ys))
    else
      Cons(y, Merge(xs, ytail))
}
```

Having defined `Split` and `Merge`, we return to `MergeSortAux`. We fill in the **else** branch where we had left off with

```
var (left, right) := Split(xs, len / 2);
```

```

    Merge(MergeSortAux(left, len / 2),
          MergeSortAux(right, len - len / 2))
}

```

This expression splits `xs` into the prefix `left` and the suffix `right`, whose lengths are $\text{len} / 2$ and $\text{len} - \text{len} / 2$, respectively. Two recursive calls to `MergeSortAux` sort these smaller lists and the sorted lists are then merged.

Here, the verifier complains that it cannot prove termination of either of the recursive calls. We have arranged to make these calls with lists of lengths $\text{len} / 2$ and $\text{len} - \text{len} / 2$, each of which is indeed smaller than len , since the recursive calls take place on the branch where $2 \leq \text{len}$. But the termination metric for `MergeSortAux` is the default—the lexicographic pair of the function's arguments: `xs, n`. It is not true that `left` is structurally included in `xs`, so the verifier has good reasons to complain about the first recursive call. (For the second call, `right` is actually structurally included in `xs`, but to communicate that to the verifier, we would have to prove a lemma.) Our way out is simple: we manually change the termination metric to `len`, which we do by adding the following clause after the type signature of `MergeSortAux`:

decreases `len`

Exercise 8.2.

The integer parameter to `Split` makes it easy to see what is happening. To use this `Split` function, we must start by computing the length of the original list (as we did when calling `MergeSortAux` from `MergeSort`). However, it is not necessary to use such an integer, because we can instead use the elements of a list to do the counting. Implement a function with the following specification:

```

function Split'(xs: List, nn: List): (List, List)
  requires Length(nn) <= Length(xs)
  ensures var (left, right) := Split'(xs, nn);
    var n := Length(nn) / 2;

```

```

    Length(left) == n &&
    Length(right) == Length(xs) - n &&
    Append(left, right) == xs

```

(The last three lines are the same as from the postcondition of `Split`.)

Exercise 8.3.

Write a new function `MergeSort'(xs: List<int>)` that is implemented directly in terms of `Split'` (from Exercise 8.2) and `Merge`. Prove `MergeSort(xs) == MergeSort'(xs)`. Hint: Lemma `AppendDecomposition` in Exercise 6.5 will be useful for the proof.

8.2.1. Ordered correctness

We prove the correctness of Merge Sort in two steps, just as for Insertion Sort. We start by showing that the output of `MergeSort` is ordered. Our implementation has four functions (`MergeSort`, `MergeSortAux`, `Split`, and `Merge`), so it is reasonable to expect that we will have four lemmas as well. As it turns out, we will need only three, because we captured the salient properties of `Split` by its intrinsic specification.

The first two lemmas mimic the structure of the corresponding functions:

```

lemma MergeSortOrdered(xs: List<int>)
  ensures Ordered(MergeSort(xs))
{
  MergeSortAuxOrdered(xs, Length(xs));
}

```

```

lemma MergeSortAuxOrdered(xs: List<int>, len: nat)
  requires len == Length(xs)
  ensures Ordered(MergeSortAux(xs, len))
  decreases len
{
  if 2 <= len {
    var (left, right) := Split(xs, len / 2);
    MergeOrdered(MergeSortAux(left, len / 2),
                  MergeSortAux(right, len - len / 2));
  }
}

```

```

    }
}

```

The lemma that shows the output of Merge to be ordered holds only if the inputs of Merge are already sorted, so the lemma also needs a precondition. The proof is done automatically by Dafny:

```

lemma MergeOrdered(xs: List<int>, ys: List<int>)
  requires Ordered(xs) && Ordered(ys)
  ensures Ordered(Merge(xs, ys))
{
}

```

Exercise 8.4.

We had to write part of the proof of MergeSortAuxOrdered above, but we didn't explicitly write calls to the induction hypothesis. Mark the lemma with `{:induction false}` and manually fill in the necessary calls.

Exercise 8.5.

Mark MergeOrdered with `{:induction false}` and write its proof without relying on Dafny's automatic induction.

8.2.2. Same-elements correctness

Our final proof obligation—which, recall, has the form of the stability equation in Section 8.0.2—requires more work. It starts off straightforwardly, following the structure of MergeSort, which calls the auxiliary function:

```

lemma MergeSortSameElements(xs: List<int>, p: int)
  ensures Project(xs, p) == Project(MergeSort(xs), p)
{
  MergeSortAuxSameElements(xs, Length(xs), p);
}

```

```

lemma MergeSortAuxSameElements(xs: List<int>, len: nat, p: int)
  requires len == Length(xs)
  ensures Project(xs, p) == Project(MergeSortAux(xs, len), p)
  decreases len

```

We need to help the verifier along for this proof, so we start off like we usually do, following the structure of the corresponding function `MergeSortAux` and starting a calculation from the more complicated side of the equality we are trying to prove:

```
{
  if 2 <= len {
    var (left, right) := Split(xs, len / 2);
    calc {
      Project(MergeSortAux(xs, len), p);
    == // def. MergeSortAux
      Project(Merge(MergeSortAux(left, len / 2),
                      MergeSortAux(right, len - len / 2)), p);
```

(The next step is the most difficult in our calculation, so prepare yourself for a longer discourse. The steps after that will be easier again.)

To simplify this `Project(..., p)` expression, we need some property about `Project(Merge(...), p)`. Let us switch our attention to such a lemma. What *can* we say about the p elements of a merge? Those elements come from the p elements of the two lists being merged. Let's call the two sorted lists being merged xs and ys and let's think about what `Merge(xs, ys)` does. The merge proceeds by repeatedly picking an element from either xs or ys and adding it to the result. It does this for all elements that are less than p before it does it for any element that is equal to or greater than p . Moreover, it does this for all elements that are equal to p before it does it for any element that is greater than p . Therefore, the effect of projecting the result of the merge to the elements that are equal to p is the same as if we ignore the non- p elements of the two lists. In other words, we expect `Project` to distribute over `Merge`:

```
Project(Merge(xs, ys), p) ==
Merge(Project(xs, p), Project(ys, p))
```

Furthermore, merging two lists of p 's is the same as appending them, which leads us to state the following

lemma:

```
lemma MergeSameElements(xs: List<int>, ys: List<int>, p: int)
  requires Ordered(xs) && Ordered(ys)
  ensures Project(Merge(xs, ys), p)
    == Append(Project(xs, p), Project(ys, p))
{
}
```

The proof of this lemma goes through automatically.

Next, we would like to apply this lemma in the calculation in lemma MergeSortAuxSameElements where we left off. That is, we would like to do the following step:

```
Project(Merge(MergeSortAux(left, len / 2),
                MergeSortAux(right, len - len / 2)),
        p);
== { MergeSameElements(
    MergeSortAux(left, len / 2),
    MergeSortAux(right, len - len / 2),
    p);
}
Append(
  Project(MergeSortAux(left, len / 2), p),
  Project(MergeSortAux(right, len - len / 2), p));
```

But the verifier complains, saying we can call MergeSameElements only on sorted lists. We can obtain this needed information by calling the lemma MergeSortAuxOrdered, which we proved above in Section 8.2.1. We make the two calls to this lemma inside the hint, before calling MergeSameElements in the hint. So, our long-awaited next step in the MergeSortAuxSameElements calculation is

```
Project(Merge(MergeSortAux(left, len / 2),
                MergeSortAux(right, len - len / 2)),
        p);
== { MergeSortAuxOrdered(left, len / 2);
    MergeSortAuxOrdered(right, len - len / 2);
    MergeSameElements(
      MergeSortAux(left, len / 2),
      MergeSortAux(right, len - len / 2),
      p);
}
Append(
```

```

    Project(MergeSortAux(left, len / 2), p),
    Project(MergeSortAux(right, len - len / 2), p));

```

Here, we have two subexpressions that project the result of MergeSortAux, so we can apply the induction hypothesis:

```

== { MergeSortAuxSameElements(left, len / 2, p);
    MergeSortAuxSameElements(right, len - len / 2, p); }
Append(Project(left, p), Project(right, p));

```

First projecting left and right on p and then appending them gives the same result as first appending left and right and then projecting on p, which we prove separately by another lemma:

```

lemma AppendProject(xs: List<int>, ys: List<int>, p: int)
  ensures Append(Project(xs, p), Project(ys, p))
    == Project(Append(xs, ys), p)
{
}

```

With this lemma, the proof calculation of MergeSortAuxSameElements is now complete:

```

== { AppendProject(left, right, p); }
   Project(Append(left, right), p);
==
   Project(xs, p);
}
}

```

Whew! We have now proved the correctness of our MergeSort function.

Exercise 8.6.

It is common for implementations of $\mathcal{O}(n \log n)$ sorting algorithms to defer to a simpler $\mathcal{O}(n^2)$ sorting algorithm when the input is small. This can lead to better performance, because $\mathcal{O}(n \log n)$ may incur greater overhead before setting up its recursive calls. Change the Merge Sort function above to call Insertion Sort on small inputs. More specifically, change **if** len < 2 **then** xs in MergeSortAux to

```
if len < 8 then InsertionSort(xs)
```

Verify that the new Merge Sort program outputs the same elements as in the input, but ordered.

8.3. Summary

In this chapter, we considered the task of sorting lists. We first wrote general specifications for sorting algorithms and then applied these to Insertion Sort and Merge Sort.

The specification of sorting has two parts: the output is sorted, and the output has the same elements as the input. A *stable* sort additionally says that elements with equal keys are arranged in the same order in the input and output.

It is critical in engineering to get the specifications right. With program proofs and an automated verifier, you can check that an implementation meets its specification. How you formulate a specification affects how easy it is for a human to read and understand the specification. How you formulate the specification also affects the complexity of proofs, and for an automated verifier, the formulation may affect the degree of automation. In this chapter, I formulated the “same elements” specification in terms of preserving the multiset of the list's elements. Not only do I find that specification more straightforward to understand, but it also avoids the complexity of (the common approach of) trying to define what a permutation is.

Notes

The gallery of verified programs available from the Why3 home page [[20](#)] contains many algorithms that sort a given List. (It also contains programs that sort mutable arrays, which in this book we'll get to in Chapter 15.) The literature contains many “proof pearls” (short articles that

demonstrate something interesting about proofs) of sorting algorithms. For example, the Natural Merge Sort algorithm, which is used in the Haskell standard library, has been verified in Isabelle/HOL [[119](#)] and in Dafny [[79](#)].

For imperative sorting algorithms in this book, see Chapter 15, which contains additional pointers and historical notes.

Chapter 9

Abstraction



Abstraction and information hiding are critical to good program design. They allow us to manage the complexity of interacting parts of a program. The chapters so far used small examples to illustrate how to reason about algorithms and how to write proofs. In those examples, there was no concern about (or need for, really) abstraction. In this chapter, I will show how to collect a type and various operations on that type into a *module*. One focus in this endeavor is how to draw the line between what clients of the module are allowed to know and what will remain as

private details of the implementation. The chapter illustrates these organizational principles by a module that implements a queue.

9.0. Grouping Declarations into Modules

One basic role of a module is to group together related declarations. For example, in Chapter 6, we defined a type `List` along with operations on lists and lemmas about those operations. We can place these declarations in a module, call it `ListLibrary`.

```
module ListLibrary {
  datatype List<T> = Nil | Cons(head: T, tail: List<T>)

  function Append(xs: List, ys: List): List
  // ...

  lemma AppendAssociative(xs: List, ys: List, zs: List)
    ensures Append(Append(xs, ys), zs) == Append(xs, Append(ys,
zs))
  // ...

  // many other declarations go here...
}
```

Not only does a module group together related declarations, but it also divides up the program's *namespace*. This means that the same identifier can be introduced in multiple modules. For example, a module `StackLibrary` can also declare a function (or type or lemma or whatever) called `Append`. To qualify which `Append` you are referring to, you prefix it with the name of the module and a `"."`. For example, to refer to the `ListLibrary` module's `Append` function from outside the `ListLibrary` module, you write `ListLibrary.Append`.

Two declarations placed alongside each other—that is, in the same *scope*—are known as *siblings*. Suppose there is a method `Main` that is a sibling of the module `ListLibrary`. The method can then use the name `ListLibrary`. Furthermore (and more interestingly), the method can refer to `ListLibrary` declarations by mentioning `ListLibrary` in qualified names. The following example illustrates:

```
method Main() {
  var xs, i := ListLibrary.Nil, 0;
  while i < 5 {
    xs, i := ListLibrary.Cons(i, xs), i + 1;
  }
  print xs, "\n";
}
```

9.1. Module Imports

A module cannot refer to another module willy-nilly. More precisely, the names of sibling modules are not automatically included in other modules. For example, the following program is not legal:

```
module ModuleA {
  function Plus(x: int, y: int): int {
    x + y
  }
}
module ModuleB {
  function Double(x: int): int {
    ModuleA.Plus(x, x) // error: unresolved identifier
    'ModuleA'
  }
}
```

To include the module name `ModuleA` inside the sibling module `ModuleB`, the latter must *import* the former. This is achieved by an **import** declaration inside `ModuleB`:

```
module ModuleB {
  import ModuleA
  function Double(x: int): int {
```

```
    ModuleA.Plus(x, x)
}
}
```

More generally, the **import** declaration gives the importing module the ability to indicate a different local name for referring to the imported module. For example, in

```
module ModuleB {
    import A = ModuleA
    function Double(x: int): int {
        A.Plus(x, x)
    }
}
```

the **import** declaration introduces the local name `A` inside `ModuleB` and binds this name to the module `ModuleA`. In fact, the simpler declaration **import** `ModuleA` is just a shorthand for **import** `ModuleA = ModuleA`. (Note that the right-hand side of the “=” in the **import** declaration is rather special, because it has the ability to refer to names that are not in scope inside the module.)

The module imports in a program are not allowed to be cyclic. In other words, the import relation in a legal program forms a hierarchy of modules. For example, if `ModuleB` imports `ModuleA`, then `ModuleA` is not allowed to import `ModuleB`.

9.2. Export Sets

Modules do more than divide up the namespace of a program. They facilitate *information hiding*. This means that certain declarations, or certain parts of declarations, are not accessible outside the module. By hiding some portions of its declarations, a module gains the ability to make private implementation decisions. Such private implementation decisions can freely be changed in future versions of the module without disturbing any client of the module.

In Dafny, which parts of a module are available outside the module is controlled by *export sets*. These are introduced by the **export** declaration. If a module does not include any **export** declaration, then everything in the module becomes available to importers of the module. For example, the `ListLibrary` module in Section 9.0 and `ModuleA` in Section 9.1 default to exporting everything.

Most every declaration consists of some kind of *signature* and some kind of *body*. The signature consists of the name introduced by the declaration, its type or parameters, and any specification clauses. The body is the rest of the declaration and typically consists of an expression or statement inside curly braces or some definition that follows an `=`.

For example, consider the following two declarations:

```
datatype Color = Blue | Yellow | Green | Red
function Double(x: int): nat
  requires 0 <= x
  ensures Double(x) % 2 == 0
{
  if x == 0 then 0 else 2 + Double(x - 1)
}
```

The signature of `Color` is just the name `Color` and the signature of `Double` is its type signature, precondition, and postcondition. The body of `Color` is what follows the `"="` and the body of `Double` is the expression in curly braces.

The export mechanism in Dafny gives control over the decision to export both the signature and body of a declaration, just the signature, or neither. In more detail, the **provides** clause of the **export** declaration lists the names of those declarations whose signature is to be exported and the **reveals** clause of the **export** declaration lists the names of those declarations whose signature and body are to be exported.

For illustration, consider the following modules:

```
module ModuleC {
```

```

export
  provides Color
  reveals Double
datatype Color = Blue | Yellow | Green | Red
function Double(x: int): nat
  requires 0 <= x
  ensures Double(x) % 2 == 0
{
  if x == 0 then 0 else 2 + Double(x - 1)
}
}

module ModuleD {
  import ModuleC
  method Test() {
    var c: ModuleC.Color;
    c := ModuleC.Yellow; // error: unresolved identifier
    'Yellow'
    assert ModuleC.Double(3) == 6;
  }
}

```

The export set of ModuleC provides the signature of the type Color and reveals both the signature and body of the function Double. Since ModuleD imports ModuleC, the type ModuleC.Color can be used in ModuleD. The example uses this type in the declaration of local variable c. However, since Color is only provided by ModuleC, not revealed, the constructors of Color are not known in ModuleD, so the particular assignment statement in ModuleD is not allowed. In fact, ModuleD does not even get to know that Color is an inductive datatype—all it can tell is that Color is a type. On the other hand, the export set in ModuleC reveals Double, so ModuleD can both refer to the function and reason about its value. In particular, the assertion in ModuleD can be verified.

An export set must provide a self-contained view of the module: if you delete what is not exported, then what remains must be a program that still type checks. For one, all identifiers mentioned must have an available declaration. To illustrate, consider the following module:

```

module ModuleE {
  datatype Parity = Even | Odd
  function F(x: int): Parity
  {
    if x % 2 == 0 then Even else Odd
  }
}

```

Suppose the export set of this module were to just provide the function F, which would be declared as follows:

```

export provides F // inconsistent export
                  // set for ModuleE

```

This would be illegal, because after deleting what is not exported, what remains is

```

function F(x: int): Parity

```

where Parity would be an unknown type. In other words, to a module that imports ModuleE, the declaration of F would seem malformed, because it mentions Parity, whose declaration is not known to the importer. Similarly, an export declaration

```

export provides Parity reveals F // inconsistent export
                               // set for ModuleE

```

would not be legal either. Here, what is made available by the export set is:

```

type Parity
function F(x: int): Parity
{
  if x % 2 == 0 then Even else Odd
}

```

where Even and Odd would be unresolved identifiers.

An example of a consistent export set for ModuleE is

```

export
  provides Parity, F

```

It makes the following declarations available:

```

type Parity
function F(x: int): Parity

```

Another legal export set is:

```
export  
  reveals Parity, F
```

which makes everything about type `Parity` and function `F` available to importers. Since those are the only two declarations in this module, the `export` declaration can also be given as

```
export  
  reveals *
```

which, as we have seen before, is also the default `export` set if there is no `export` declaration at all.

9.3. Modular Specification of a Queue

Let's try our hand at applying good information hiding to the implementation of an immutable queue. A *queue* is a list of elements with three operations: *initialization* creates an empty queue, *enqueue* appends an element to the queue, and *dequeue* removes and returns an elements from the head of the queue. The queues we consider here are *immutable*, so each operation returns a queue value rather than modifying some existing queue.

9.3.0. A basic queue interface

The most basic module describing a queue is the following:

```
module ImmutableQueue {  
  type Queue<A>  
  function Empty(): Queue  
  function Enqueue<A>(q: Queue, a: A): Queue  
  function Dequeue<A>(q: Queue): (A, Queue)  
    requires q != Empty()  
}
```

This is not yet a complete module—let me build it up one step at a time. What I have shown here is how a *client* of the module (that is, a user of the module) would see it. Such a view of a module is called the module *interface*,

because it describes how the module's clients and implementation interface with one another.

This module interface declares a type `Queue`, parameterized by some element type `A`. The module does not reveal anything about this type other than its name and number of type parameters. This is good information-hiding practice, because it gives the implementation freedom in how to represent values of the type. We say it *abstracts* over the implementation, which simply means that we can speak of queues in higher-level, more abstract terms.

The three operations are declared as functions. Since the `dequeue` operation cannot be applied to an empty queue, it has a precondition.

9.3.1. Abstraction functions

The basic queue interface above is too simple, because it does not give a client enough information to verify its use of the queue. For example, a client cannot tell from the basic interface whether or not the `Enqueue` operation may return an empty queue, which makes ever calling `Dequeue` difficult. To describe the operations in more detail, it would be helpful to be able to speak about what a queue really is.

A queue consists of a list of elements. If we could speak about that list, it would make for useful and intelligible specifications. It's that easy! Let's introduce a function from a queue to its list of elements:

```
ghost function Elements(q: Queue): List
```

The function is declared as `ghost`, because we intend to use this function only in specifications. It could be that the queue is implemented as a list, but just because we have this function does not constrain our implementation of the queue to be a list. We say that this function *abstracts* over the implementation and that the function is called an

abstraction function, because it lets us think of the queue at a level that is higher than its eventual implementation.

Whatever module you are designing, it is a good idea to think about what abstraction the module provides. Doing so gives you clarity of mind when you dip into the implementation details.

Using the abstraction function `Elements`, we can specify the queue operations in terms of lists of elements. The specifications can be given intrinsically (in the postcondition of the functions) or extrinsically (in separate lemmas). Let us do it extrinsically. Here is our revised module interface:

```
module ImmutableQueue {
  import LL = ListLibrary

  type Queue<A>
  function Empty(): Queue
  function Enqueue<A>(q: Queue, a: A): Queue
  function Dequeue<A>(q: Queue): (A, Queue)
    requires q != Empty()

  ghost function Elements(q: Queue): LL.List
  lemma EmptyCorrect<A>()
    ensures Elements(Empty<A>()) == LL.Nil
  lemma EnqueueCorrect<A>(q: Queue, x: A)
    ensures Elements(Enqueue(q, x)) == LL.Snoc(Elements(q), x)
  lemma DequeueCorrect(q: Queue)
    requires q != Empty()
    ensures var (a, q') := Dequeue(q);
      LL.Cons(a, Elements(q')) == Elements(q)
}
```

In order for the module to talk about lists, it imports the module `ListLibrary`. We are going to refer to the declarations in `ListLibrary` many times, so it will be convenient to have a shorter name to use in qualifications. For this reason, the **import** declaration introduces the local name `LL` to stand for the imported library.

For module `ImmutableQueue` to refer to declarations inside module `ListLibrary`, it prefixes them with “`LL.`”. For

example, inside module `ListLibrary`, the datatype constructor `Cons` can be referred to as just `Cons` (since the name does not clash with other declarations in `ListLibrary`) or as `List.Cons` or even `List<B>.Cons` for any type `B`. To refer to this constructor from `ImmutableQueue`, we can therefore write `LL.Cons`, `LL.List.Cons`, or `LL.List<B>.Cons`.

I chose to declare `Elements` as a ghost function, because we will use it only in specifications. This is common for abstraction functions. But if the function's body is compilable and you want to make it available at run time, just remove the keyword **ghost** from the function's declaration.

Exercise 9.0.

Function `Elements` provides an abstraction of the queue implementation. Another, more coarse-grained, abstraction would have been to instead introduce a function

```
ghost function Length(q: Queue): nat
```

that returns the number of elements in the queue. Write a module interface that uses `Length` instead of `Elements`.

9.3.2. Declaring an export set

We did not yet explicitly declare an export set in our module-under-development. By default, Dafny will therefore reveal everything. This would be fine for the body-less declarations we have written so far. But as we complete the module to fully define the type, functions, and lemmas, we do not want to reveal those definitions. By writing an export set explicitly, we get to control the clients' view of the module.

We use **provides** clauses in an **export** declaration to list the names of declarations we wish to make available outside the module. To make the export set self-contained, we must make sure to provide clients with the `ListLibrary` as well.

We accomplish this by listing its local name, `LL`, in the export declaration.

Here is the export declaration we add to the module:

```
export
  provides Queue, Empty, Enqueue, Dequeue
  provides LL, Elements
  provides EmptyCorrect, EnqueueCorrect, DequeueCorrect
```

Since these are all the names declared in our module so far, we could have written

```
export provides *
```

However, this would also provide other helper functions and lemmas that we might introduce later.

One more remark about exporting the imported module `ListLibrary`. Through the exported name `LL`, a client module that imports `ImmutableQueue` under the name `IQ` can refer to the list type as `IQ.LL.List<B>` for any type `B`. A client can also declare its own import of the `ListLibrary` module, say

```
import MyOwnLLImport = ListLibrary
```

and then use `IQ.LL.List<B>` and `MyOwnLLImport.List<B>` synonymously.

9.3.3. A sample client

Now that we have a module interface, let us test it to make sure it can be used by a simple client. Our client will create an empty queue, enqueue an element, and then dequeue that element:

```
module QueueClient {
  import IQ = ImmutableQueue
  method Client() {
    var q := IQ.Empty();
    q := IQ.Enqueue(q, 20);
    var (a, q') := IQ.Dequeue(q); // precondition violation
  }
}
```


The verifier complains that the call to `Dequeue` may violate its precondition. We had anticipated this before, which is why we introduced the lemmas. Let us make use of the lemmas—one for each function call (phew!)—and let us also add a couple of assertions at the end to check that we are getting the end results we'd expect:

```
module QueueClient {
  import IQ = ImmutableQueue
  method Client() {
    IQ.EmptyCorrect<int>(); var q := IQ.Empty();
    IQ.EnqueueCorrect(q, 20); q := IQ.Enqueue(q, 20);
    IQ.DequeueCorrect(q); var (a, q') := IQ.Dequeue(q);
    assert a == 20;
    assert q' == IQ.Empty(); // assertion cannot be verified
  }
}
```

The precondition of `Dequeue` can now be verified, so we can see that our lemmas are useful. But they don't do everything we would like, because the verifier complains about the last assertion.

To debug this situation, we can try to precede the failing assertion with the following assertion:

```
assert IQ.Elements(q') == IQ.LL.Nil;
```

This assertion goes through the verifier! So, if we know that `q'` represents the empty list of elements, then how come we cannot verify that it equals `IQ.Empty()`, which (as `EmptyCorrect()` tells us) also represents the empty list of elements?

The problem is that there may be several values of the type `Queue<int>` that represent the empty list of elements. Indeed, depending on what data structure the implementation of `ImmutableQueue` uses, there may be many `Queue<int>` values that represent the same list of elements. In general, it can be computationally very expensive to make sure that all values of a type are canonical, that is, that each one represents a different abstract value.

We have a decision to make. At one extreme, we can decide that a client should never compare a `Queue` value directly, but should always resort to comparisons via the abstraction function `Elements`. At the other extreme, we can decide that the implementation must make it possible to compare `Queue` values directly, which essentially comes down to making sure function `Elements` is *injective* (that is, that `Elements` is a one-to-one function). We will settle for a good choice in between these two extremes, namely to make `Elements` injective for the empty queue. Stated differently, we will make `Empty()` the only value of `Queue` that represents the empty list of elements. We accomplish this by adding another lemma in `ImmutableQueue`:

```
lemma EmptyUnique(q: Queue)
  ensures Elements(q) == LL.Nil ==> q == Empty()
```

and (don't forget!) adding `EmptyUnique` to the **provides** clause of the module's export clause. As usual, there are several ways to express the property of this lemma (see the discussion of `LessTransitive` in Section 7.1). Here, I chose to use an implication in the postcondition, rather than writing the antecedent in a precondition. This makes it possible to use the contrapositive of the lemma. That is, by calling the lemma with a `q` that is known not to be `Empty()`, a caller is able to conclude `Elements(q) != LL.Nil`.

Finally, our simple client can be verified:

```
module QueueClient {
  import IQ = ImmutableQueue
  method Client() {
    IQ.EmptyCorrect<int>(); var q := IQ.Empty();
    IQ.EnqueueCorrect(q, 20); q := IQ.Enqueue(q, 20);
    IQ.DequeueCorrect(q);    var (a, q') := IQ.Dequeue(q);
    assert a == 20;
    IQ.EmptyUnique(q');
    assert q' == IQ.Empty();
  }
}
```

This gets the job done, but it sure looks cluttered. We'll work on reducing clutter like this in Section 10.3. Next, we are ready to take on the implementation of `ImmutableQueue`.

Exercise 9.1.

Write a client module (like `QueueClient`) to test the module interface in Exercise 9.0. Adjust the queue interface so that your client module verifies.

9.3.4. Queue implementation

We return to module `ImmutableQueue`. To make it complete, we need to define the type `Queue`, the four functions, and the four lemmas.

One simple implementation for our queue would be to use a list, the same list that `Elements` returns. However, the cost of appending an element to the list, as the `Enqueue` operation would do, is proportional to the length of the queue. The key idea behind getting a better implementation is to use two lists, call them `front` and `rear`. A portion of the elements at the head of the queue are stored in `front`, where they can be dequeued in constant time. The rest of the elements are stored in `rear` in *reverse order*, which lets them be enqueued in constant time. When the `Dequeue` operation finds `front` empty, it copies *and reverses* the elements from `rear` into `front`. Consequently, each element dequeued will have undergone two constant-time operations and been part of one reversal. As we have seen in Section 6.6, list reversal requires linear time, so `Dequeue` will have an *amortized* running time that is constant. This means that as a queue evolves, the sum of the running times of `Dequeue` operations that created the queue is a constant multiple of the number of `Dequeue` invocations.

We start our implementation by defining the type of the data structure. We replace the declaration `type Queue<A>`

with the following:

```
datatype Queue<A> = FQ(front: LL.List<A>, rear: LL.List<A>)
```

Note, since our module only *provides*, not *reveals*, the type `Queue`, the fact that `Queue` is defined as a datatype remains a private implementation decision in the module.

The key idea of this queue implementation is captured in the definition of `Elements`:

```
ghost function Elements(q: Queue): LL.List {  
  LL.Append(q.front, LL.Reverse(q.rear))  
}
```

The other functions are also easy to write:

```
function Empty(): Queue {  
  FQ(LL.Nil, LL.Nil)  
}  
  
function Enqueue<A>(q: Queue, x: A): Queue {  
  FQ(q.front, LL.Cons(x, q.rear))  
}  
  
function Dequeue<A>(q: Queue): (A, Queue)  
  requires q != Empty()  
{  
  match q.front  
  case Cons(x, front') =>  
    (x, FQ(front', q.rear))  
  case Nil =>  
    var front := LL.Reverse(q.rear);  
    (front.head, FQ(front.tail, LL.Nil))  
}
```

Lemmas `EmptyCorrect` and `EmptyUnique` are done automatically—you need only supply the empty body, `{}`, so I won't show them here. The proofs of the other two lemmas make for good exercises, so I won't show them here, either. 🤖

Exercise 9.2.

Write the proof for `EnqueueCorrect`.

Exercise 9.3.

Write the proof for DequeueCorrect.

Exercise 9.4.

(a) Write a module `Blockchain` declaring a type `Ledger` with three operations that have the following type signatures:

```
function Init(): Ledger
function Deposit(ledger: Ledger, n: nat): Ledger
function Withdraw(ledger: Ledger, n: nat): Ledger
```

Write an abstraction function

```
ghost function Balance(ledger: Ledger): int
```

that reports the balance of a ledger, and make $n \leq \text{Balance}(\text{ledger})$ be a precondition of `Withdraw`. Declare an export set for the module.

(b) Write intrinsic specifications for the functions in terms of `Balance`. Write a simple client to test that they behave as you'd expect.

(c) Implement the module with a list of integers. Specifically, define `Ledger` to be a type synonym:

```
type Ledger = List<int>
```

where `List` is our usual list type. The implementations of `Deposit` and `Withdraw` prepend (a non-negative or non-positive number, respectively) to the list.

(Note that parameter `n` to `Deposit` and `Withdraw` is `nat`, whereas `Balance` returns an `int`. In order to declare `Balance` to return a `nat`, we need a data-structure invariant, which I will cover in Chapter 10, see Exercise 10.5.)

9.4. Equality-Supporting Types

There is one more operation that clients of our immutable queue may want to use and that we thus need to export: equality. If you don't feel too concerned about this, you may want to skip this section, because it goes into a number of picky details of the Dafny language.

9.4.0. Equality

In Dafny, every type has a built-in equality operator, written `==`. For example, equality on integers (`int`) is the expected arithmetic equality, and equality on sets of integers (`set<int>`) is mathematical set equality where the elements are compared using equality on integers. Equality can always be used in ghost contexts in Dafny, but equality cannot always be used in compiled contexts. This is because the values of some types may not be computable in finite time or at all. For example, to compare possibly infinite sets of integers (`iset<int>`) requires an infinite number of membership tests, and to compare functions on integers (`int -> int`) requires invoking the function on an infinite number of values.

For this reason, Dafny keeps track of which types are known to have a *compilable* equality operation. These are referred to as “equality-supporting types”—a slight misnomer, since all types support equality in ghost contexts. When the definition of a type is available, Dafny uses the definition to figure out whether or not the type supports equality. This is not possible for type parameters, opaque types, or types that are just *provided* by an export set. In such cases, we can explicitly declare that the type refers to an equality-supporting type. This is done by writing the *type characteristic* (`==`) after the name of the type in its declaration. (We saw an instance of this already, for function `Find` in Section 6.5.)

Consider a client of module `ImmutableQueue` that defines a variation of `Dequeue`, one that can be applied to any queue and that returns a default value when given an empty queue. Here is such a client module:

```
module QueueExtender {
  import IQ = ImmutableQueue
  function TryDequeue<A>(q: IQ.Queue, default: A): (A,
    IQ.Queue)
```

```

    {
      if q == IQ.Empty() then (default, q) else IQ.Dequeue(q)
    }
  }
}

```

With `ImmutableQueue` as we defined it, Dafny complains that function `TryDequeue` uses `==` to try to compare values of type `IQ.Queue<A>`. In module `QueueExtender`, it is not known whether or not `IQ.Queue<A>` supports equality. We will consider two possible ways out of this quagmire.

9.4.1. Explicating equality support

One way out of the quagmire is to export, from module `ImmutableQueue`, that type `Queue<A>` does indeed support equality. You might think this would be done by adding a `(==)` in the **datatype** declaration:

```

datatype Queue(==)<A> = // syntax error
  FQ(front: LL.List<A>, rear: LL.List<A>)

```

However, the `(==)` mark can be used (for type parameters, as we saw for `Find` in Section 6.5 or) only in **type** declarations. That's not a problem—we simply rename the **datatype** and use a **type** declaration to define a *type synonym* that we export:

```

type Queue(==)<A> = // claimed equality support is in error
  Queue'<A>
datatype Queue'<A> = FQ(front: LL.List<A>, rear: LL.List<A>)

```

This almost works. The problem is that the **datatype** `Queue'<A>` does not always support equality! It supports equality only if the type parameter `A` does. So, to claim that `Queue<A>` supports equality, we must restrict the definition to `A`'s that support equality. This is done by marking the type parameter with `(==)` as well:

```

type Queue(==)<A(==)> = Queue'<A>
datatype Queue'<A> = FQ(front: LL.List<A>, rear: LL.List<A>)

```

Exercise 9.5.

How come we didn't need `(==)` already when we compared `q' == IQ.Empty()` in method `Client()` in Section 9.3.3?

9.4.2. Exporting an emptiness predicate

Another way out of the quagmire is for `ImmutableQueue` to export not the `queue ==` operation itself but only a test that compares a queue with `Empty()`. After all, as I remarked in Section 9.3.3 where I motivated the introduction of the `EmptyUnique` lemma, equality on `Queue` values is not so useful, because there are several `Queue` values that represent the same list of elements. But equality is useful for the empty queue, because `ImmutableQueue` is using a unique representation for the empty queue. Actually, you could also argue that using the precondition `q != Empty()`—back in Section 9.3.0—was a mistake, because it already assumes a unique representation for the empty queue.

Under this quagmire exit strategy, we keep the **datatype** declaration of `Queue` as we had it before, and we add to module `ImmutableQueue` the following compiled predicate:

```
predicate IsEmpty(q: Queue)
  ensures IsEmpty(q) <==> q == Empty()
{
  q == FQ(LL.Nil, LL.Nil)
}
```

We also add `IsEmpty` in a **provides** clause of the **export** declaration in `ImmutableQueue`.

Looking at the body of `IsEmpty`, you may wonder how it can compare `q` using equality, since `q` is of a type that supports equality only if the element type of `q` supports equality. The explanation is that, as a special case, Dafny allows comparisons with expressions that consist of just literals, like `FQ(LL.Nil, LL.Nil)`. After all, such a comparison could also be done with **match** expressions or with the datatype discriminators `FQ?` and `Nil?`. (As a reminder, the

comparison `q == Empty()` in the postcondition is also allowed, because Dafny always supports equality in ghost contexts.)

Exercise 9.6.

Without using `(==)`, rewrite module `QueueExtender` to use `IsEmpty`.

Exercise 9.7.

Export two new members of module `ImmutableQueue`: a compiled function `Length` that returns the number of elements in a given queue, and a correctness lemma about `Length(q)` that says it returns the length of the list `Elements(q)`. From these, it follows that the test `Length(q) == 0` gives the same result as would `q == Empty()`. Rewrite function `TryDequeue` in module `QueueExtender` to use `Length`.

Exercise 9.8.

The implementation of `TryDequeue` as suggested in Exercise 9.7 is not very efficient. Why?

9.5. Summary

Modular design is crucial in software engineering. It leads to a program decomposition where implementation decisions can be hidden inside modules. David Parnas pointed this out, in what can be summarized in the slogan “every module has one secret” [108]. To make it possible to write specifications and do verification in the presence of such information hiding, we abstract over the details. In this chapter, I showed how that can be done using an abstraction function (`Elements`, in the example) that maps the values we are implementing to values of other, known, simpler types.

Declarations are divided up into modules, and a module imports the modules it relies on. By declaring an export set, a module decides which declarations are to be available to

clients. For each declaration in an export set, the export set can choose to include just the signature of the declaration (using **provides** clauses) or both the signature and body of the declaration (using **reveals** clauses).

Notes

Many fun and useful implementations of functional data structures, such as the queue implementation in Section 9.3.4, are found in Chris Okasaki's book on *Purely Functional Data Structures* [104].

To support information hiding, programming languages provide mechanisms of *access control* that limit the visibility of declarations. As we have seen, access control for the declarations in a module in Dafny is defined via export sets. Many class-based object-oriented languages, including Java and C#, allow members of a **class** to be declared with access modifiers like **private** (meaning, visible only in the implementation of the class) and **public** (meaning, usable in any context that has access to the class). C++ and Eiffel additionally allow members of one class to declare “friend” classes that get privileged access to the members. Other languages, including SPARK [117], Oberon-2, and Go, use a private/public mechanism at the module level.

In many programming languages, access-control mechanisms govern only whether or not a client has access to a given declaration. This is sufficient for compilation. But when specifications and proofs are involved, it is also important to be able to restrict how much of the *meaning* of a declaration is visible to a client. For this reason, Dafny offers two levels of exporting declarations (**provides** and **reveals**) [80], and F* [53] allows its module interfaces to contain semantic information.

Each module in this book has only one export set, which is the common case. In other cases, a module may need to present different clients with different views of the module. For example, a module may want to reveal some detail of its representation to some “friend” module. For this purpose, Dafny allows a module to declare several export sets [80]. This kind of flexibility is also provided by the modules and interfaces in Modula-3 [100].

Abstraction can be achieved in various ways. In this chapter, we used abstraction functions—ghost functions that speak of some values without constraining the implementation to any particular representation of those values. Abstraction functions are common in verification tools. In Chapters 16 and 17, we'll achieve abstraction by using ghost fields in combination with a validity predicate. The SPARK language [117] has special support for declaring abstract state (`Abstract_State`) and the concrete counterpart (`Refined_State`). There are many great texts that teach techniques for abstracting over data; to learn from true masters of abstractions, scope out Morgan using specification statements [93], Abrial using Event-B [0], and Lamport using TLA⁺ [73].

Chapter 10

Data-Structure Invariants



Thank goodness for types! They let us describe the skeleton of a data structure and they give a simple way for a program analysis to check that a program maintains that skeleton. This means we can rely on, for example, the fact that a value of the type `Tree` defined in Section 4.5 is either a `Leaf` or a `Node`.

For most data structures, there is more to the design than just the skeleton. Every value of the data structure matches the skeleton, but not every value that matches the skeleton is a value that the data-structure operations can ever produce. For example, a *binary search tree* is a data structure whose skeleton is a tree and whose payload values are arranged in sorted order. Constraints like these that go beyond the skeleton are called *data-structure*

invariants. Invariants are critical to good program design and program correctness.

To illustrate data-structure invariants in this chapter, I will use a *priority queue* as the running example. Whereas an ordinary queue is a collection of elements that are accessed in a first-in first-out fashion (as we saw in Section 9.3), a priority queue gives access only to a minimal element of the collection. We start with the specification and then go into the implementation.

10.0. Priority-Queue Specification

Other than creating and checking for an empty priority queue, the operations on a priority queue are insertion of an element and removal of the minimal element. We will consider priority queues that store integers, so “minimal” is defined in terms of arithmetic comparisons. We start our specification by declaring the following simple (and incomplete) module:

```
module PriorityQueue {  
  type PQueue  
  function Empty(): PQueue  
  predicate IsEmpty(pq: PQueue)  
  function Insert(pq: PQueue, y: int): PQueue  
  function RemoveMin(pq: PQueue): (int, PQueue)  
    requires !IsEmpty(pq)  
}
```

So far, the module looks a lot like our starting point for ordinary queues in Section 9.3. Here, I have decided right away that we don't want to commit to a unique representation of values, not even the empty priority queue, and therefore the module introduces an `IsEmpty` predicate (as described in Section 9.4.2).

10.0.0. Abstraction

To say more about what is returned by these functions, we need some abstraction of what each `PQueue` value represents. As in Section 9.3, we introduce for this purpose an abstraction function that maps a `PQueue` to its elements. Here, we have a choice. One possibility is to abstract the elements into a sorted sequence (of type `seq<int>`). This will make it easy to write the specification for `RemoveMin`, but will complicate the specification of `Insert`. Another possibility (and this is what we will do) is to abstract the elements into a multiset (of type `multiset<int>`). This makes the specification of `Insert` simple, but complicates (only slightly) the specification of `RemoveMin`.

We introduce such a (specification-only) abstraction function `Elements` in our module. Using it, we write specifications for our functions extrinsically, as separate lemmas:

```
ghost function Elements(pq: PQueue): multiset<int>
lemma EmptyCorrect()
  ensures Elements(Empty()) == multiset{ }
lemma IsEmptyCorrect(pq: PQueue)
  ensures IsEmpty(pq) <==> Elements(pq) == multiset{ }
lemma InsertCorrect(pq: PQueue, y: int)
  ensures Elements(Insert(pq, y))
    == Elements(pq) + multiset{y}
lemma RemoveMinCorrect(pq: PQueue)
  requires !IsEmpty(pq)
  ensures var (y, pq') := RemoveMin(pq);
    IsMin(y, Elements(pq)) &&
    Elements(pq') + multiset{y} == Elements(pq)
```

The lemma about `RemoveMin` uses a predicate `IsMin`, which says that `y` is a minimal element in the multiset `Elements(pq)`. It's possible to define `IsMin` recursively, but a much more straightforward way is to use a *universal quantifier* (**forall**, written $\forall$ in mathematical texts) that can say something about every elements in the multiset. I will illustrate quantifiers in greater detail in Chapter 13; for our purposes here, we only need a head-nodding understanding of that `IsMin` is defined correctly.

```
ghost predicate IsMin(y: int, s: multiset<int>) {
  y in s && forall x :: x in s ==> y <= x
}
```

In words, `IsMin(y, s)` says that `y` is an element of the multiset `s` and that for each element `x` in `s`, element `y` is below element `x`.

Our module so far looks pretty good from the point of view of clients.

Exercise 10.0.

Write a small test harness in a client module (cf. `QueueClient` in Section 9.3.3). Make the test harness insert two elements into a newly created priority queue and then use `RemoveMin` twice to retrieve them. Verify that the two elements are returned in the correct order.

Hint: To convince the verifier of this, you will need to use some lemmas. Of course, you will need to invoke the lemmas declared in `PriorityQueue`, but you will also need some lemmas to help the verifier reason about multisets. In particular, you will need to write some inline assertions about what is known about the elements of the priority at intermediate program points in test harness.

10.0.1. Export set

As we continue the development of the `PriorityQueue` module, let us be clear about what clients get to see and what remains private implementation details. We do that by adding an explicit **export** declaration. The module provides everything we have declared so far, except for `IsMin`, which it reveals in full.

```
export
  provides PQueue, Empty, IsEmpty, Insert, RemoveMin
  provides Elements
  provides EmptyCorrect, IsEmptyCorrect
  provides InsertCorrect, RemoveMinCorrect
  reveals IsMin
```

This export set uses a mixture of `provides` and `reveals` clauses, which I introduced in Section 9.2.

10.1. Designing the Data Structure

To implement the priority queue, we will use a *binary heap*. A binary heap is a binary tree with two properties. First, a skeletal property: unlike the tree in Section 9.3, which placed the tree's payload in the leaves, the binary heap places its payload in the interior nodes of the tree and the leaves are empty. Second, a property beyond the skeleton, a data-structure invariant: every node in the tree satisfies what is called the *heap property*—the value stored in a node is minimal among all the values stored in the subtree rooted at that node.

The heap property makes it possible to obtain good performance: methods `Insert` and `RemoveMin` can each operate in time $\mathcal{O}(\log n)$ (that is, in time proportional to the logarithm of the size of the priority queue), *provided* the tree is balanced. A tree is *balanced* if every left subtree has roughly the same size as its neighboring right subtree.

An elegant data structure that ensures that the binary-heap tree is balanced is a *Braun tree*. This is what we are going to implement. The central idea of a Braun tree is that either the left and right subtrees of a node have the same size or the left subtree has one more element than the right subtree.

So, we have three properties altogether. There's the skeletal property that places values in interior nodes, there's the heap property, and there's the Braun-tree balance property. We call the latter two properties *data-structure invariants*, since they go beyond the skeletal property.

To ensure the skeletal property of the binary heap, we define type `PQueue` to be the following datatype:


```

type PQueue = BraunTree
datatype BraunTree =
  | Leaf
  | Node(x: int, left: BraunTree, right: BraunTree)

```

(Dafny allows a `|` before the first constructor name, which looks nice when you declare the constructors on separate lines, as I did here.) To ensure the data-structure invariant of Braun trees, we need to involve some logical expressions.

10.1.0. Valid trees

The correctness of our priority queue relies on the heap property. Thus, for verification, we need to distinguish those trees that have the heap property from other trees that (in principle) could be built from the constructors `Leaf` and `Node`. We do this by introducing a predicate that we shall call `Valid`. The idea is that a tree satisfies `Valid` if and only if it's a tree that may be returned from our `PriorityQueue` module.

We provide the `Valid` predicate to clients so it can be used in specifications, but we will keep the details private to the module.

```

export // ...
provides Valid

ghost predicate Valid(pq: PQueue)

```

Next, we have a choice to make: `Valid` can be used intrinsically or extrinsically. That is, we can either use `Valid` in the specifications of the priority-queue operations themselves (intrinsically) or we can use `Valid` only in the correctness lemmas (extrinsically). If the crash-free functioning of the tree operations will rely on the validity condition, then we *have* to use it intrinsically, but otherwise it is a choice like other extrinsic-versus-intrinsic choices we have seen before (see Sections 6.2). For now, I will use

Valid extrinsically, but I will return to this issue in Section 10.3.

Using `Valid`, we update the definitions of the lemmas as follows:

```
lemma EmptyCorrect()
  ensures var pq := Empty();
    Valid(pq) &&
    Elements(pq) == multiset{}
lemma IsEmptyCorrect(pq: PQueue)
  requires Valid(pq)
  ensures IsEmpty(pq) <==> Elements(pq) == multiset{}
lemma InsertCorrect(pq: PQueue, y: int)
  requires Valid(pq)
  ensures var pq' := Insert(pq, y);
    Valid(pq') &&
    Elements(pq') == Elements(pq) + multiset{y}
lemma RemoveMinCorrect(pq: PQueue)
  requires Valid(pq) && !IsEmpty(pq)
  ensures var (y, pq') := RemoveMin(pq);
    Valid(pq') &&
    IsMin(y, Elements(pq)) &&
    Elements(pq') + multiset{y} == Elements(pq)
```

Exercise 10.1.

Revisit Exercise 10.0 to make sure that the test harness still works, now that the correctness lemmas take into consideration the data-structure invariant.

10.2. Implementation

10.2.0. Defining the invariant

We define `Valid` to include all the properties we want to make sure are invariants of our data structure. In addition to the heap property, our invariant says that the data structure satisfies the Braun-tree balance property. Here are those definitions:

```
ghost predicate Valid(pq: PQueue) {
  IsBinaryHeap(pq) && IsBalanced(pq)
```

```

}

ghost predicate IsBinaryHeap(pq: PQueue) {
  match pq
  case Leaf => true
  case Node(x, left, right) =>
    IsBinaryHeap(left) && IsBinaryHeap(right) &&
    (left == Leaf || x <= left.x) &&
    (right == Leaf || x <= right.x)
}

ghost predicate IsBalanced(pq: PQueue) {
  match pq
  case Leaf => true
  case Node(_, left, right) =>
    IsBalanced(left) && IsBalanced(right) &&
    var L, R := |Elements(left)|, |Elements(right)|;
    L == R || L == R + 1
}

```

Since predicate `IsBinaryHeap` is defined to hold recursively for the subtrees `left` and `right`, it suffices to state for each node the heap property between the node's value and the value stored in its direct children (see Exercise 10.2 after the definition of `Elements` below).

As in the queue example in Section 9.3, abstraction function `Elements` is central to all our descriptions of correctness. It abstracts over the implementation's data structure to present clients with a simple mental model of the priority queue's operation. We define the function as the multiset union of all the values stored in the tree:

```

ghost function Elements(pq: PQueue): multiset<int> {
  match pq
  case Leaf => multiset{}
  case Node(x, left, right) =>
    multiset{x} + Elements(left) + Elements(right)
}

```

Exercise 10.2.

Predicate `IsBinaryHeap` is defined to hold recursively for the subtrees `left` and `right`; the property “a node holds

the minimum value of the node's entire subtree" is stated only as comparisons between the node's value and the values stored in its direct children. Prove the definition correct. In other words, prove that the value stored in any node that satisfies `IsBinaryHeap` is minimal in the tree rooted at that node. That is, prove the following lemma:

```
lemma {:induction false} BinaryHeapStoresMinimum(pq: PQueue,  
y: int)  
  requires IsBinaryHeap(pq) && y in Elements(pq)  
  ensures pq.x <= y
```

The statement of the lemma itself is curious. The postcondition selects `.x`, so you might expect the precondition to have a conjunct `pq.Node?`. Why is the statement of the lemma still okay?

The proof of this lemma is also curious. (Spoiler alert!) The proof is by induction over the structure of `pq`. Most such proofs break into the structural cases. Not this one. Here, `pq` is required to be a `Node`. The proof's cases instead break down like the 3-part union in the definition of `Elements`.

Next, we write the implementation and correctness proofs for emptiness, insertion, and removal.

10.2.1. Emptiness

An empty priority queue is represented as a leaf. We define the body of `Empty` as follows:

```
function Empty(): PQueue {  
  Leaf  
}
```

In fact, in our implementation, the empty priority queue has a unique representation. So, we define predicate `IsEmpty` as an equality comparison with `Leaf`:

```
predicate IsEmpty(pq: PQueue) {  
  pq == Leaf  
}
```

Next, we prove the correctness of `Empty` and `IsEmpty` by filling in the bodies of lemmas `EmptyCorrect` and `IsEmptyCorrect`:

```
lemma EmptyCorrect()
  ensures var pq := Empty();
    Valid(pq) &&
    Elements(pq) == multiset{ }
{
}

lemma IsEmptyCorrect(pq: PQueue)
  requires Valid(pq)
  ensures IsEmpty(pq) <==> Elements(pq) == multiset{ }
{
}
```

Both of these lemmas are verified automatically.

10.2.2. Insertion

To insert an element into an empty binary heap is just a matter of returning a tree storing the element in a node with empty subtrees:

```
function Insert(pq: PQueue, y: int): PQueue {
  match pq
  case Leaf => Node(y, Leaf, Leaf)
```

To insert an element into a `Node` takes some more work. Recall that a binary heap is a tree where the element stored in each node compares below the elements stored in the node's subtrees. Thus, if the element stored in the node, call it x , is below the one we want to insert, y , then the insertion operation proceeds by creating a node that stores x and inserting y into a subtree. Otherwise, if y is below x , then the insertion operation does it the other way around: creating a node that stores y and inserting x into a subtree.

When x or y is inserted into a subtree, which subtree do we want to insert it into? As for maintaining the heap property, it does not matter. We can do the recursive

insertion in either the left or right subtree. As for maintaining the balance property, however, it matters. We want to do the insertion into whichever subtree is smaller.

How do we determine which of the two subtrees is smaller? Here is where the key insight behind Braun trees comes into play. In a Braun tree, the right subtree is never larger than the left, so we want to insert it there. But if the two subtrees start off having the same size, then inserting into the right subtree would make it bigger. The beautiful observation in Braun trees is that we can maintain the Braun-tree invariant by (inserting into the right subtree and then) swapping the left and right subtrees. This will result in a tree where, again, either the left and right subtrees have the same size or the left subtree is one larger.

Here is the rest of the code in the Insert method:

```
case Node(x, left, right) =>
  if y < x then
    Node(y, Insert(right, x), left)
  else
    Node(x, Insert(right, y), left)
}
```

The correctness proof of Insert is by induction, following the cases in the function. In fact, it is so simple that the verifier handles it automatically:

```
lemma InsertCorrect(pq: PQueue, y: int)
  requires Valid(pq)
  ensures var pq' := Insert(pq, y);
    Valid(pq') &&
    Elements(pq') == Elements(pq) + multiset{y}
{
}
```

10.2.3. Removal of the minimum

Because of the precondition `!IsEmpty(pq)`, function `RemoveMin(pq)` always applies to a Node, never to a Leaf. The

element of that node is returned in the first component of RemoveMin's return tuple, because a binary-heap stores its minimal element in the root node (see also Exercise 10.2). The other component of the return tuple is pq but with the minimum deleted. Let's write an auxiliary recursive function, DeleteMin, to do the latter. Then, we implement RemoveMin as follows:

```
function RemoveMin(pq: PQueue): (int, PQueue)
  requires !IsEmpty(pq)
{
  (pq.x, DeleteMin(pq))
}
```

```
function DeleteMin(pq: PQueue): PQueue
  requires !IsEmpty(pq)
```

We define a correctness lemma for DeleteMin. This will make sure that we are clear about what DeleteMin is supposed to do. Plus, it lets us prove the correctness of RemoveMin.

```
lemma RemoveMinCorrect(pq: PQueue)
  requires Valid(pq) && !IsEmpty(pq)
  ensures var (y, pq') := RemoveMin(pq);
    Valid(pq') &&
    IsMin(y, Elements(pq)) &&
    Elements(pq') + multiset{y} == Elements(pq)
{
  DeleteMinCorrect(pq);
}
```

```
lemma DeleteMinCorrect(pq: PQueue)
  requires Valid(pq) && pq != Leaf
  ensures var pq' := DeleteMin(pq);
    Valid(pq') &&
    Elements(pq') + multiset{pq.x} == Elements(pq)
```

It seems magical that the proof of RemoveMinCorrect only calls lemma DeleteMinCorrect, since the latter does not mention anything about IsMin. The magic comes from Dafny's automatic induction, which lets the verifier discharge the proof obligation about IsMin without further

guidance from us. (To confirm that induction is used, you can temporarily disable automatic induction for `RemoveMinCorrect` by marking it with the attribute `{:induction false}`. This will cause the verifier to complain that it no longer can prove the lemma.) I cannot expect you to write the proof about `IsMin` yourself at this time, because I have not yet said anything about how to write a manual proof of the universal quantifier in `IsMin`. So, if you want to understand in more detail why `y` is the minimum among the values in `Elements(pq)`, I instead recommend you do Exercise 10.2.

10.2.4. Auxiliary function `DeleteMin`

For a small tree where one or both subtrees are empty, the deletion of the root results in just the left subtree:

```
function DeleteMin(pq: PQueue): PQueue
  requires !IsEmpty(pq)
{
  if pq.left == Leaf || pq.right == Leaf then
    pq.left
```

Consider a node with two nonempty subtrees. To delete its minimum, we put in its place the root of one of the subtrees. We want to be sure that the element we “promote” in this way is the smaller of the roots of the two subtrees, since this is what is needed to maintain the heap property. So we do something like this:

```
    else if pq.left.x <= pq.right.x then
      Node(pq.left.x, ...)
    else
      Node(pq.right.x, ...)
```

In the first of these two cases, we need to delete the element we promoted from the left subtree, so we call `DeleteMin(pq.left)` recursively. This will maintain the heap property of the Braun tree. To correctly maintain the balance condition of the Braun tree, we also swap the two

subtrees, essentially reversing what the Insert operation does. This makes the first of the two cases into:

```

else if pq.left.x <= pq.right.x then
  Node(pq.left.x, pq.right, DeleteMin(pq.left))

```

The second case is not symmetric to the first, because if we called function `DeleteMin(pq.right)`, we would produce a subtree that could potentially contain two fewer elements than `pq.left`, which would not satisfy the balance property of Braun trees. Instead, we first swap the roots of the left and right subtrees and then proceed as in the previous case. We do the swap with yet one more auxiliary method, `ReplaceRoot`:

```

else
  Node(pq.right.x, ReplaceRoot(pq.right, pq.left.x),
    DeleteMin(pq.left))
}

```

```

function ReplaceRoot(pq: PQueue, y: int): PQueue
  requires !IsEmpty(pq)

```

```

lemma ReplaceRootCorrect(pq: PQueue, y: int)
  requires Valid(pq) && !IsEmpty(pq)
  ensures var pq' := ReplaceRoot(pq, y);
    Valid(pq') &&
    Elements(pq) + multiset{y} ==
    Elements(pq') + multiset{pq.x}

```

Exercise 10.3.

The very first case of `DeleteMin` handles the situation when one or both subtrees are empty. In the `if` condition, I wrote the test as a disjunction (an “or”):

```

pq.left == Leaf || pq.right == Leaf

```

If the tree is balanced, then the condition `pq.left == Leaf` implies `pq.right == Leaf`, so the disjunction is equivalent to the simpler test `pq.right == Leaf`. Why didn't I write it that way? What happens if you change the condition to the simpler one?

We start the proof of DeleteMinCorrect off with a usual case study, following the cases of the DeleteMin function:

```
lemma DeleteMinCorrect(pq: PQueue)
  requires Valid(pq) && pq != Leaf
  ensures var pq' := DeleteMin(pq);
    Valid(pq') &&
    Elements(pq') + multiset{pq.x} == Elements(pq)
{
  if pq.left == Leaf || pq.right == Leaf {
  } else if pq.left.x <= pq.right.x {
    DeleteMinCorrect(pq.left);
```

The first case is handled automatically. For the second case, the definition of function DeleteMin makes a recursive call on pq.left, so we expect that the lemma will need an analogous recursive call. Indeed, that does the trick.

The third case is more involved. Let's start off by giving names to subexpressions we'll use:

```
  } else {
    var left, right :=
      ReplaceRoot(pq.right, pq.left.x), DeleteMin(pq.left);
    var pq' := Node(pq.right.x, left, right);
```

That's a lot of symbols. To make sure we didn't introduce any mistakes, we let the verifier check that pq' is indeed the result of DeleteMin:

```
    assert pq' == DeleteMin(pq);
```

We expect to need the correctness properties of left and right, so we add calls to the relevant lemmas:

```
    ReplaceRootCorrect(pq.right, pq.left.x);
    DeleteMinCorrect(pq.left);
```

If we were lucky, the verifier would now finish the proof. It doesn't, so we have more work to do. There are various ways to proceed. If you were on your own, you would probably (like I did) try proving each of the different parts of the lemma's conclusion. When I tried it, I then ended up with a long and ugly proof. After looking at various pieces of the proof, refactoring it, and trying many different ways

to prove each part, I ended up with a reasonable proof. Here, I will only show my final proof.

It starts off by a proof calculation that shows the `Elements` correctness property of `DeleteMin`. It is a property that shows the equality of two multisets. For such proofs, the verifier typically requires a fair amount of hand-holding. To understand how to approach such a proof, I have two pieces of advice.

First piece of advice for proofs that use multisets

The verifier may be able to prove the equality of two multisets, but it is not very creative when it comes to figuring out which equalities it should try to prove. Therefore, if you have an expression like `...A...` where `A` denotes a multiset and you want to show that it is equal to an expression `...B...` for some other multiset `B`, then you typically have to structure your proof calculation like this:

```
...A...;  
== { assert A == B; }  
...B...;
```

In the hint, the assertion will tell the verifier to prove the equality of multisets `A` and `B`. The verifier does this by trying to prove the multisets have the same elements (more precisely, that the multisets have the same multiplicity of every element). Once it has verified that `A` and `B` are indeed equal, it will use that equality to check the step in the proof calculation.

There's a name for the kind of equality in the assertion above: *extensionality*. By asking the verifier to check `A == B`, it does so by checking that `A` and `B` have the same elements, which it often can do, even if it didn't think of the need for the equality `A == B` by itself.

Second piece of advice for proofs that use multisets

Just like addition for arithmetic, multiset union is associative and commutative. For example, $A + B + C$ is the same multiset as $A + C + B$. The verifier can check this, but you may have to guide it by bringing the equality to its attention, like I showed above. When it substitutes equals for equals, the verifier treats multiset union as being left associative and not necessarily commutative. Consequently, suppose a subexpression $A + B + C$ appears in a formula and you want to show that the formula is unchanged if you replace that subexpression with $A + D + E$, where you have some way to proving $B + C == D + E$. If you tried a proof like

```
...A + B + C...;
== { assert B + C == D + E; } // I'm supposing you have
                                // some reason for this to hold
...A + D + E...; // you get an error that the proof
                  // step to here does not hold
```

then the verifier would still complain, because what it sees is:

```
...(A + B) + C...;
== { assert B + C == D + E; }
    ...(A + D) + E...; // you get an error that the proof
                        // step to here does not hold
```

Instead, you have to manually rearrange the parts of the formula. This is most comfortably done in separate steps.

```
...A + B + C...;
== { assert A + B + C == A + (B + C); }
    ...A + (B + C)...;
== { assert B + C == D + E; } // again, I'm supposing
                                // you have some reason for why this holds
    ...A + (D + E)...;
== { assert A + (D + E) == A + D + E; }
    ...A + D + E...;
```

Enough advice, let's get back to the proof

With these two pieces of advice in mind, we can now do the proof calculation for `DeleteMin`. In the steps without hints, all

I have done is to rearrange the operands of the multiset union.

```

calc {
  Elements(pq') + multiset{pq.x};
== // def. Elements, since pq' is a Node
  multiset{pq.right.x} + Elements(left) +
  Elements(right) + multiset{pq.x};
==
  Elements(left) + multiset{pq.right.x} +
  Elements(right) + multiset{pq.x};
== { assert Elements(left) + multiset{pq.right.x}
      == Elements(pq.right) + multiset{pq.left.x}; }
  Elements(pq.right) + multiset{pq.left.x} +
  Elements(right) + multiset{pq.x};
==
  Elements(right) + multiset{pq.left.x} +
  Elements(pq.right) + multiset{pq.x};
== { assert Elements(right) + multiset{pq.left.x}
      == Elements(pq.left); }
  Elements(pq.left) + Elements(pq.right) +
  multiset{pq.x};
==
  multiset{pq.x} + Elements(pq.left) +
  Elements(pq.right);
== // def. Elements, since pq is a Node
  Elements(pq);
}

```

To finish the proof of DeleteMin, we need some information about the sizes of $pq'.left$ and $pq'.right$. For pq' to be balanced, the size of $pq'.left$ should be equal to or one more than the size of $pq'.right$. Since pq is balanced, the property we need follows from the fact that the recursive call to DeleteMin reduces the size by one and the call to ReplaceRoot does not change the size. We can prove these properties from the postconditions we wrote for lemmas DeleteMinCorrect and ReplaceRootCorrect. However, it will be more convenient if these lemmas had postconditions that mentioned these size properties explicitly. So, let's change the postcondition of DeleteMinCorrect to add the conjunct $|Elements(pq')| == |Elements(pq)| - 1$ and change

the postcondition of `ReplaceRootCorrect` to add the conjunct $|\text{Elements}(\text{pq}')| == |\text{Elements}(\text{pq})|$.

The proof of `DeleteMinCorrect` now goes through without further interaction:

```
    }  
  }
```

So, after strengthening the lemmas to explicitly mention how `DeleteMin` and `ReplaceRoot` affect the size of a tree, the only non-boilerplate proof we had to supply was the proof calculation about `Elements` in the third branch of `DeleteMin`.

10.2.5. Auxiliary function `ReplaceRoot`

The finish line is in sight—we have just one function left. Function `ReplaceRoot(pq, y)` deletes the minimum element of `pq` and inserts `y`. If neither subtree of `pq` has an element that is smaller than `y`, then we return a node like `pq` but with `y` as its root. This happens in several cases, depending on how many nonempty subtrees `pq` has:

```
function ReplaceRoot(pq: PQueue, y: int): PQueue  
  requires !IsEmpty(pq)  
  {  
    if pq.left == Leaf ||  
      (y <= pq.left.x && (pq.right == Leaf || y <= pq.right.x))  
    then  
      Node(y, pq.left, pq.right)
```

Of the small-tree cases, only one remains, namely when `pq` has one nonempty subtree with an element that is smaller than `y`. In that case, we promote that element and place `y` in a subtree by itself:

```
    else if pq.right == Leaf then  
      Node(pq.left.x, Node(y, Leaf, Leaf), Leaf)
```

In the remaining cases, we promote the root of whichever subtree has the minimal element and then replace its minimum with `y`:

```

else if pq.left.x < pq.right.x then
  Node(pq.left.x, ReplaceRoot(pq.left, y), pq.right)
else
  Node(pq.right.x, pq.left, ReplaceRoot(pq.right, y))
}

```

The proof of `ReplaceRoot` starts off in the usual way, following the cases of the function. The first two cases are handled automatically. (Recall that the proof of `DeleteMinCorrect` compelled us to add to `ReplaceRootCorrect` the conjunct that says `ReplaceRoot` does not change the size of the tree.)

```

lemma ReplaceRootCorrect(pq: PQueue, y: int)
  requires Valid(pq) && !IsEmpty(pq)
  ensures var pq' := ReplaceRoot(pq, y);
    Valid(pq') &&
      Elements(pq) + multiset{y} == Elements(pq') +
multiset{pq.x} &&
    |Elements(pq')| == |Elements(pq)|
  {
    if pq.left == Leaf ||
      (y <= pq.left.x && (pq.right == Leaf || y <=
pq.right.x))
    {
      } else if pq.right == Leaf {

```

We start the third case by introducing names for the subexpressions of the result of `ReplaceRoot`, as we also did for the third case of `DeleteMin` in Section 10.2.4.

```

    } else if pq.left.x < pq.right.x {
      var left := ReplaceRoot(pq.left, y);
      var pq' := Node(pq.left.x, left, pq.right);

```

We follow this up with a check that we wrote the expressions correctly:

```

      assert pq' == ReplaceRoot(pq, y);

```

The proof is surely going to use the induction hypothesis, suitably parameterized, so we add that call, too.

```

      ReplaceRootCorrect(pq.left, y);

```

Next, we prove the lemma's conclusion about Elements. This is another example of where the verifier can check multiset properties, but we have to lead it through which properties to check.

```

calc {
  Elements(pq) + multiset{y};
  == // def. Elements, since pq is a Node
    multiset{pq.x} + Elements(pq.left) +
    Elements(pq.right) + multiset{y};
  ==
    Elements(pq.left) + multiset{y} +
    Elements(pq.right) + multiset{pq.x};
  == { assert Elements(pq.left) + multiset{y}
      == Elements(left) + multiset{pq.left.x}; } // I.H.
    multiset{pq.left.x} + Elements(left) +
    Elements(pq.right) + multiset{pq.x};
  == // def. Elements, since pq' is a Node
    Elements(pq') + multiset{pq.x};
}

```

The fourth and final case is similar:

```

} else {
  var right := ReplaceRoot(pq.right, y);
  var pq' := Node(pq.right.x, pq.left, right);
  assert pq' == ReplaceRoot(pq, y);
  ReplaceRootCorrect(pq.right, y);
  calc {
    Elements(pq) + multiset{y};
    == // def. Elements, since pq is a Node
      multiset{pq.x} + Elements(pq.left) +
      Elements(pq.right) + multiset{y};
    ==
      Elements(pq.right) + multiset{y} +
      Elements(pq.left) + multiset{pq.x};
    == { assert Elements(pq.right) + multiset{y}
        == Elements(right) + multiset{pq.right.x}; }
      multiset{pq.right.x} + Elements(pq.left) +
      Elements(right) + multiset{pq.x};
    == // def. Elements, since pq' is a Node
      Elements(pq') + multiset{pq.x};
  }
}
}

```


Well, it took us a while, but we have now implemented all the functions in module `PriorityQueue`, and we have proved it all to be correct. The code is subtle, and there are more cases to consider than a human reviewer or test suite could confidently check. In contrast, the relatively simple specification and our accompanying proof give us high assurance that we have written the code correctly.

Sidebar 10.0

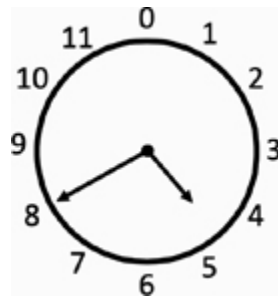
In the 12-hour system of clocks, many people get 12am and 12pm confused. The confusing thing is that an hour after 11am is 12pm (noon). If instead of using

$1 \leq \text{hour} \leq 12$

you replace the 12 on the clock by 0, so you get

$0 \leq \text{hour} < 12$

then it would be easy to remember that 0pm (noon) is an hour before 1pm.



Choose your invariants wisely and it will simplify your design.

10.3. Making Intrinsic from Extrinsic

In Section 10.0.0, we made the decision in the development of our `PriorityQueue` module that all specifications of our

functions, except the non-emptiness precondition of `RemoveMin`, would be extrinsic. This brings the benefit of not having to clutter up the function bodies with proof elements. A drawback is that the functions cannot rely on the data-structure invariant. For some functions, this is not a big deal. Indeed, all the functions in the queue example in Chapter 9 work without spelling out a data-structure invariant at all. For the Braun tree example, it makes a difference. Let's see how.

10.3.0. Optimizing `DeleteMin`

We started off our function `DeleteMin` like this:

```
function DeleteMin(pq: PQueue): PQueue
  requires !IsEmpty(pq)
{
  if pq.left == Leaf || pq.right == Leaf then
    pq.left
```

For a tree that satisfies the Braun-tree balance condition, this `if` test is a bit redundant. Any Braun tree satisfies:

`pq.left == Leaf ==> pq.right == Leaf`

so the disjunction in the `if` test can be simplified to just `pq.right == Leaf`. In other words, there is no case where only the left tree is a leaf, so the code might as well just inspect the right tree.

Exercise 10.4.

Formulate and prove a lemma that shows the Braun-tree balance condition to entail the implication above.

It is now tempting to change our `DeleteMin` implementation to instead start off

```
function DeleteMin(pq: PQueue): PQueue
  requires !IsEmpty(pq)
{
  if pq.right == Leaf then
    pq.left
```

```

    else if pq.left.x <= pq.right.x then // error: .x requires
pq.left
                                     // to be a Node

```

but that does not work, because of the next line of the function, which I included here. Since this `DeleteMin` function could, in principle, be called on any nonempty tree, even trees that do not satisfy the Braun-tree balance property, the verifier complains that the code tries to access member `x` of `pq.left`. (Well, there! I just gave you the answer to Exercise 10.3.)

To optimize our code for Braun trees, we need to give it a precondition that restricts it to Braun trees. That's what our decision of making the specifications extrinsic did not consider. Let's change that decision, at least partially, and add a precondition to `DeleteMin`.

```

function DeleteMin(pq: PQueue): PQueue
  requires IsBalanced(pq) && !IsEmpty(pq)
{
  if pq.right == Leaf then
    pq.left
  else if pq.left.x <= pq.right.x then
    Node(pq.left.x, pq.right, DeleteMin(pq.left))
  else
    Node(pq.right.x, ReplaceRoot(pq.right, pq.left.x),
        DeleteMin(pq.left))
}

```

With this change of specifications, the function body verifies even with the simplified `if` test. Good. But the verifier complains about the call to `DeleteMin` in `RemoveMin`. Let's change its specification accordingly. We might try:

```

function RemoveMin(pq: PQueue): (int, PQueue)
  requires IsBalanced(pq) && !IsEmpty(pq)

```

This does not work, because our module exports `RemoveMin`, but not `IsBalanced`. We could consider exporting `IsBalanced` as well, but that would burden priority-queue clients with yet another concept. Instead, let's strengthen the precondition further:

```
function RemoveMin(pq: PQueue): (int, PQueue)
  requires Valid(pq) && !IsEmpty(pq)
```

This does the trick. We have now optimized the code in `DeleteMin`, strengthened its precondition, and propagated its change in specifications to its callers.

10.3.1. The intrinsic-extrinsic spectrum

To apply the optimization in `DeleteMin`, we had to strengthen some specifications. We imposed as few additional preconditions as possible, culminating in the addition of `Valid(pq)` to the precondition of the exported function `RemoveMin`. Aesthetically, this leaves something to be desired, since `RemoveMin` is now the only one of the exported functions to mention `Valid`. In other words, we have gotten ourselves into a situation where the module interface uses a mix between intrinsic and extrinsic specifications.

We could consider making all of the specifications fully intrinsic. In other words, instead of writing the function specifications in separate correctness lemmas, we could write the specifications on the functions directly. In this approach, function `RemoveMin` would be declared as

```
function RemoveMin(pq: PQueue): (int, PQueue)
  requires Valid(pq) && !IsEmpty(pq)
  ensures var (y, pq') := RemoveMin(pq);
    Valid(pq') &&
    IsMin(y, Elements(pq)) &&
    Elements(pq') + multiset{y} == Elements(pq)
```

The choice between intrinsic and extrinsic specifications is a spectrum. An intermediate point in the design space is to always include the data-structure invariant, `Valid`, in the intrinsic specifications, but, to the extent possible, omit other functional-correctness specifications. In this approach, function `RemoveMin` would be declared as

```
function RemoveMin(pq: PQueue): (int, PQueue)
  requires Valid(pq) && !IsEmpty(pq)
  ensures var (y, pq') := RemoveMin(pq);
```

Valid(pq')

10.3.2. Externally intrinsic, internally extrinsic

Regardless of which point of the intrinsic-extrinsic spectrum we choose, as soon as a function includes a nontrivial postcondition, we face the extrinsic-specification disadvantage of having to pollute the function body with additional lemma calls. But there is a division of labor that gives us a simple intrinsic (or as-intrinsic-as-we-want) module interface and yet lets us separate function definitions from their correctness proofs. It is to layer a suite of intrinsically specified functions on top of the extrinsically specified functions. A clean way to do this is to write the two layers as separate modules.

First, let us change the name of the `PriorityQueue` module that we have written so far to `PriorityQueueExtrinsic`.

```
module PriorityQueueExtrinsic {  
  export  
    provides PQueue, Empty, IsEmpty, Insert, RemoveMin  
    provides Valid, Elements  
    reveals IsMin  
    provides EmptyCorrect, IsEmptyCorrect, InsertCorrect  
    provides RemoveMinCorrect  
  
  // the rest of the module as before...  
}
```

Next, we define another module as a layer above `PriorityQueueExtrinsic`, where types and functions are defined directly in terms of the extrinsic counterparts. This module does not contain the correctness lemmas, because we are about to write the specifications as part of the functions. We give this module the straightforward name `PriorityQueue`, because we expect it to be the module that clients will use. Here, you can see how it works:

```
module PriorityQueue {  
  export
```

```

provides PQueue, Empty, IsEmpty, Insert, RemoveMin
provides Valid, Elements
reveals IsMin
provides Extrinsic

import Extrinsic = PriorityQueueExtrinsic

type PQueue = Extrinsic.PQueue

ghost predicate Valid(pq: PQueue) {
  Extrinsic.Valid(pq)
}

ghost function Elements(pq: PQueue): multiset<int> {
  Extrinsic.Elements(pq)
}

ghost predicate IsMin(m: int, s: multiset<int>) {
  Extrinsic.IsMin(m, s)
}

function Empty(): PQueue {
  Extrinsic.Empty()
}

predicate IsEmpty(pq: PQueue) {
  Extrinsic.IsEmpty(pq)
}

function Insert(pq: PQueue, y: int): PQueue {
  Extrinsic.Insert(pq, y)
}

function RemoveMin(pq: PQueue): (int, PQueue)
  requires Valid(pq) && !IsEmpty(pq)
{
  Extrinsic.RemoveMin(pq)
}

```

Note that the module export set provides the symbol `Extrinsic`. This is necessary if we want to, like in

PriorityQueueExtrinsic, reveal function IsMin. (By providing Extrinsic, we also make it possible to reveal, not just provide, type PQueue. This can be useful in some advanced situations where we expect clients to want to use both modules.)

Sidebar 10.1

As I've shown them here, modules PriorityQueue and PriorityQueueExtrinsic are sibling declarations. An alternative organization is to make PriorityQueueExtrinsic a nested module, which you do by writing it inside module PriorityQueue. You can then also remove the **import** declaration in PriorityQueue that introduced the local name Extrinsic and give the inner module the name Extrinsic.

Finally, we add the specifications and proofs. This involves copying each correctness-lemma specification from PriorityQueueExtrinsic to the respective function in PriorityQueue and adding a call from the function to the lemma.

```
function Empty(): PQueue
  ensures var pq := Empty();
    Valid(pq) &&
    Elements(pq) == multiset{ }
{
  Extrinsic.EmptyCorrect();
  Extrinsic.Empty()
}

predicate IsEmpty(pq: PQueue)
  requires Valid(pq)
  ensures IsEmpty(pq) <==> Elements(pq) == multiset{ }
{
  Extrinsic.IsEmptyCorrect(pq);
```

```

    Extrinsic.IsEmpty(pq)
}

function Insert(pq: PQueue, y: int): PQueue
  requires Valid(pq)
  ensures var pq' := Insert(pq, y);
    Valid(pq') &&
    Elements(pq') == multiset{y} + Elements(pq)
{
  Extrinsic.InsertCorrect(pq, y);
  Extrinsic.Insert(pq, y)
}

function RemoveMin(pq: PQueue): (int, PQueue)
  requires Valid(pq) && !IsEmpty(pq)
  ensures var (y, pq') := RemoveMin(pq);
    Valid(pq') &&
    IsMin(y, Elements(pq)) &&
    Elements(pq') + multiset{y} == Elements(pq)
{
  Extrinsic.RemoveMinCorrect(pq);
  Extrinsic.RemoveMin(pq)
}

```

In this example, where I showed how to layer an intrinsic-specification module on top of an extrinsic-specification module, I used the full extrinsic specifications. If you want a different point on the intrinsic-extrinsic spectrum, it would follow the same form.

10.4. Summary

An *invariant* is a condition that always holds. In this chapter, the invariant of interest described the consistency condition of Braun trees: the heap property and the Braun-tree balance property. We used this invariant to prove the functional correctness of priorityqueue operations, and also to prove that the operations produce only balanced trees. We encoded the invariant as a predicate `Valid`. More precisely, `Valid` is a predicate that holds when a given

skeleton value satisfies the desired data-structure invariant. We then used `Valid()` in pre- and postconditions (for the correctness lemmas in Section 10.1.0 and for the intrinsically specified functions in Section 10.3.2).

By using `Valid()` in the pre- and postcondition of the priority-queue operations, clients of our `PriorityQueue` module are made aware of the existence of a validity condition. By only providing, not revealing, `Valid()` in the module interface, we abstract over the details of the invariant. In effect, this means that clients cannot fabricate their own tree values from scratch, because the client doesn't know what `Valid()` entails. Instead, clients can obtain priority-queue values only by calling the operations we provided in the module interface. In this way, we have created a verified interface with desirable information hiding.

If there's a property you think always holds of your data structure, you can use the invariant to check it. To do that, write and prove a lemma that the data-structure invariant (`Valid()`) implies the property of interest (see Exercise 10.4).

From the perspective of program structure and specification, what the example I showed has in common with so many other modules that provide immutable data structures is that it captures the data-structure invariant in a predicate `Valid` and uses some number (here, one) of abstraction functions (like `Elements`) to describe the effect of the various operations of the module. We'll see a similar, slightly more complicated pattern when we get to mutable data structures in the next Part of the book.

Exercise 10.5.

Revisit the `Blockchain` module of Exercise 9.4 and change the result type of the `Balance` function from the previous `int` to `nat`. To prove that the function does indeed return a

non-negative number, you need to use a data-structure invariant. Add a predicate

```
ghost predicate Valid(ledger: Ledger)
```

to the module, like we have done in this chapter. Use this validity predicate in the specifications of the `Init`, `Deposit`, `Withdraw`, and `Balance` functions, and prove that the implementations of the functions meet their specifications. As usual, write and verify a test harness to make sure your specifications are usable in the way you'd expect.

Notes

It was Aaron Stump's delightful introductory book on verification in Agda [[120](#)] that introduced me to Braun trees [[24](#)].

The validity predicates we used in this chapter (and those we'll see in Chapters 16 and 17) let us describe the expected state of a data structure without divulging details of that state to clients. This achieves information hiding. But couldn't we hide even more? The way we created our module interface, a client will never be able to obtain a non-valid skeleton. So, then, is it really necessary to bother clients with the existence of a validity condition? The general answer to this question depends on if there are ways for clients to detect mutations in or interactions between parts of the implementation. Since we wrote our module using functional programming, where values are immutable, the answer is that we can in fact hide the existence of `Valid` as well. Many verifiers (including Dafny) support *subset types* (aka *predicate subtypes* or *refinement types*) for this purpose, but I won't cover them in this book.

Some verification languages and proof assistants, including Coq [[16](#)], PVS [[107](#)], Agda [[22](#)], F* [[53](#)], and LiquidHaskell [[123](#)], are primarily based around such

subset types. In those languages, it is customary (or even necessary) to formulate datastructure invariants as parts of types. Advanced forms of such programming and verification are treated in Adam Chlipala's book on *Certified Programming* [[27](#)].

Part 2

Imperative Programs

In imperative programming, the value of a variable can be changed during the execution of the program. This gives us a natural way to represent evolving real-life state, like your social-media contact list or the speed of a locomotive. It also means that we don't have to immediately produce the desired value of a computation, but we can proceed in steps, gradually changing the values of variables until we obtain what we want. Imperative programming is sometimes hailed as achieving efficiency, because a step of a computation can update just what needs to be changed rather than making a copy of all pieces of the state that do not change.

In this Part, I will show you how to reason about three kinds of imperative state: local variables, arrays of values, and dynamically allocated, evolving data structures. Common to reasoning about each of these is some form of *invariant*, like the datastructure invariants we already saw in Chapter 10.

Chapter 11

Loops



Many interesting computations can be described as repetitions of steps. The common construct for repeating a part of a computation is the *loop*. There are many forms of loop constructs, but in this chapter—indeed, in this entire book—I will consider just one form of loop, the **while** statement.

Central to reasoning about any loop is the *loop invariant*. Let's start with loop specifications, then loop

implementations, termination, and many examples.

11.0. Loop Specifications

The specification of a loop consists of two parts. The *loop invariant* restricts the state space that the loop is allowed to explore. The *loop guard* controls when the loop continues performing steps and when it stops doing so.

In the pre-state of a loop, there is a proof obligation that the invariant hold. I'll refer to this as the *loop-use proof obligation*. The *effect* of a loop is to go from one state where the invariant holds to a state where both the invariant and the negation of the guard hold.

11.0.0. Non-technical examples

Let's see how this works. Here's an illustrative loop specification:

```
while not in Seattle
  invariant on the train
```

What follows the **while** keyword is the guard, and what follows the **invariant** keyword is the invariant. A loop also has a *loop body*, whose execution is repeated as long as the loop guard holds. However, throughout this section, I will intentionally omit the loop body so that we can focus just on the loop *specification*.

According to the loop-use proof obligation, you must be on the train initially in order to use this loop. So, you'll want to arrange for the code before the loop to get you on the train. After the loop, by the invariant and the negation of the guard, you are both on the train and in Seattle. So, I expect you'll want to arrange for the code that follows the loop to get you off the train and start enjoying your day in the city.

Note that the invariant only says you're on the train, so by the loop specification, you may be in different cars on entry and exit of the loop. Also, note that you are allowed to start the loop in Seattle, as long as you are on the train. That is, it is the loop invariant that must hold on entry to the loop, and the value of the loop guard is irrelevant. After the loop, both the invariant and the negation of the guard hold.

Here's another example:

```
while homework is not done
  invariant awake
```

To use this loop, you must start off awake (even if your homework happens to be done already). After the loop, you find yourself both awake and with your homework done. Compare that loop specification with the following:

```
while awake
  invariant homework is not done
```

To use this loop, you must start off with your homework not done. After the loop, you find yourself asleep and your homework is still not done.

11.0.1. Floyd logic for loop specifications

I've stated the rule for loop specifications in words. We can state it formally in Floyd logic, as we did in Chapter 2. For boolean expressions B and J , the semantics of a loop specification is given by this Hoare triple:

```
{{ J }}
while B
  invariant J
{{ J && !B }}
```

It says that to use the loop, you must prove that its invariant holds initially. This is the loop-use proof obligation. After the loop, you get to find out that the invariant and the negation of the loop guard hold.

11.0.2. Numeric examples

Here's an example loop specification that uses integers:

```
while x < 300  
  invariant x % 2 == 0
```

To use this loop, you must prove that x is even. After the loop, you get to assume that x is still even and that $300 \leq x$.

What about this one?

```
while x % 2 == 1  
  invariant 0 <= x <= 100
```

This loop comes with the proof obligation that x be in the range from 0 through 100. After the loop, you get to find out that x is an even number from 0 through 100.

Here's a loop specification preceded by an assignment and followed by an assertion:

```
x := 2;  
while x < 50  
  invariant x % 2 == 0  
assert 50 <= x && x % 2 == 0;
```

The loop-use proof obligation that $x \% 2 == 0$ hold on entry is provable, because x has the value 2 at that time. After the loop, we get to know that the invariant holds and that the negation of the guard holds. This is exactly the condition in the assertion, so the assertion is provable as well.

Here's another example:

```
x := 0;  
while x % 2 == 0  
  invariant 0 <= x <= 20  
assert x == 19; // not provable
```

On entry, x is 0, so the proof obligation that x be in the range from 0 through 20 is met. To prove the assertion, we need to prove

$$0 \leq x \leq 20 \ \&\& \ x \% 2 \neq 0 \implies x == 19$$

We cannot prove this implication. For example, x could be 3 on exit from the loop, since 3 is odd and is in the range

from 0 through 20.

Exercise 11.0.

For each of the following uses of loop specifications, indicate whether or not the loop-use proof obligation is met and whether or not the assertion following the loop can be proved to hold.

a) <code>x := 0;</code> while <code>x != 100</code> invariant <code>true</code> assert <code>x == 100;</code>	b) <code>x := 20;</code> while <code>10 < x</code> invariant <code>x % 2 == 0</code> assert <code>x == 10;</code>
c) <code>x := 20;</code> while <code>x < 20</code> invariant <code>x % 2 == 0</code> assert <code>x == 20;</code>	d) <code>x := 3;</code> while <code>x < 2</code> invariant <code>x % 2 == 0</code> assert <code>x % 2 == 0;</code>
e) if <code>50 < x < 100</code> { while <code>x < 85</code> invariant <code>x % 2 == 0</code> assert <code>x < 85 && x % 2 == 1;</code> }	f) if <code>0 <= x</code> { while <code>x % 2 == 0</code> invariant <code>x < 100</code> assert <code>0 <= x;</code> }
g) <code>x := 0;</code> while <code>x < 100</code> invariant <code>0 <= x < 100</code> assert <code>x == 25;</code>	

11.0.3. Attaining equality

A common situation with loops is that we want a variable to start at one end of an interval and terminate at the other end of that interval. For example, we may initialize a variable *i* with 0 and want to loop until *i* has reached 100. In such situations, we often refer to variable *i* as the *loop index*.

Here is such a program fragment:

```
i := 0;
while i != 100
    invariant 0 <= i <= 100
assert i == 100;
```

From the negation of the loop guard alone, we can prove the assertion. So, the proof of this assertion does not depend on the invariant. For example, the assertion is also provable in the following program:

```
i := 0;
while i != 100
    invariant true
assert i == 100;
```

Another variation of the first loop specification is

```
i := 0;
while i < 100
    invariant 0 <= i <= 100
assert i == 100;
```

This assertion is also provable. But here, unlike in the two examples above, the guard is $i < 100$, so it is not possible to prove the assertion from just the negation of the guard ($100 \leq i$). This is where the invariant $i \leq 100$ comes in. From $100 \leq i$ (the negation of the guard) and $i \leq 100$ (the invariant), we can conclude $i == 100$, which makes the assertion provable.

As a fourth example, consider

```
i := 0;
while i < 100
    invariant true
assert i == 100; // not provable
```

This assertion is not provable, because all we know after the loop is $\text{true} \ \&\& \ 100 \leq i$.

The conclusion from these four little examples is that there are several ways to write a loop specification that reaches a particular value. If the loop guard is $i \neq 100$, then we immediately know that i is 100 after the loop. If the loop guard is $i < 100$, then the loop invariant also needs to stipulate $i \leq 100$ in order for us to conclude $i == 100$ after the loop.

Moreover, in my experience, understanding these four examples and why each one verifies or does not verify is

key to working with loops. Everything I've said so far comes from the loop *specification* (the invariant and the guard). Indeed, reasoning about a loop uses the loop specification, not the loop body. Try not to forget this when I introduce loop bodies in Section 11.1.

Exercise 11.1.

For each program, give a possible value of `i` of type `int` after the loop that shows that the assertion is not provable.

a) <code>i := 0;</code> while <code>i < 100</code> invariant <code>0 <= i</code> assert <code>i == 100;</code>	b) <code>i := 100;</code> while <code>0 < i</code> invariant <code>true</code> assert <code>i == 0;</code>
c) <code>i := 0;</code> while <code>i < 97</code> invariant <code>0 <= i <= 99</code> assert <code>i == 99;</code>	d) <code>i := 22;</code> while <code>i % 5 != 0</code> invariant <code>10 <= i <= 100</code> assert <code>i == 55;</code>

Exercise 11.2.

For each program in Exercise 11.1, strengthen the invariant so that the invariant both holds on entry to the loop and suffices to prove the assertion.

11.0.4. Variable relations

Loops are much more interesting when they involve more than one variable. Here's a loop specification where the loop invariant relates two variables:

```
x, y := 0, 0;
while x < 300
    invariant 2 * x == 3 * y
assert 200 <= y;
```

Following our rule for loop specifications, the relation between `x` and `y` stated in the invariant holds after the loop, as does the negation of the guard, that is, `300 <= x`. This lets

us conclude $200 \leq y$ after the loop, so we can prove the assertion.

Here's another example:

```
x, y := 0, 191;
while !(0 <= y < 7)
  invariant 7 * x + y == 191
  assert x == 191 / 7 && y == 191 % 7;
```

This program computes the integer quotient and modulus of 191 and 7, as the (provable) assertion confirms. A variation of this loop specification is

```
x, y := 0, 191;
while 7 <= y
  invariant 0 <= y && 7 * x + y == 191
  assert x == 191 / 7 && y == 191 % 7;
```

This invariant constrains y to non-negative values, so we can simplify the guard.

As a final example, the following program computes the sum of the first 33 natural numbers:

```
n, s := 0, 0;
while n != 33
  invariant s == n * (n - 1) / 2
```

Through a familiar arithmetic identity, the invariant expresses that s is the sum of the first n natural numbers. After the loop, we can infer that n is 33, and therefore we also know that s is the sum of the first 33 natural numbers.

11.0.5. Loop frames

Consider the following loop specification, where the loop computes r to be the integer square root of N :

```
r, N := 0, 104;
while (r+1)*(r+1) <= N
  invariant 0 <= r && r*r <= N
  assert 0 <= r && r*r <= N < (r+1)*(r+1);
```

Before doing anything else, let's pause for a moment to appreciate the beauty of this program fragment. For r to be

the integer square root of N means that N lies in the range from the square of r to the square of $r+1$, and r itself is non-negative. We want these three inequalities to be provable after the loop. One way to express that in a loop specification is to put two of the conditions into the invariant and to put the negation of the third condition into the loop guard. Aaaaaah, what a beaut!

By constructing the loop specification from the pieces of the assertion, we made sure the assertion in our example is provable. So, then, can we conclude that r is 10 after the loop? If N is still 104 after the loop, then sure, r will be 10. But the loop specification does not say N will still be 104. For example, the loop specification allows r and N on exit from the loop to be 2 and 5, respectively.

From this realization, we see that our loop specifications have a glaring omission: they don't say which variables are allowed to be changed and which must stay the same. The part of a specification that talks about what can be modified and what has to be unchanged is called a *frame*—in our context, more precisely, a *loop frame*. We could add an explicit loop frame to the loop invariant and loop guard that are already part of a loop specification. But let's not, because already in the next section, I have a plan for figuring out the loop frame by looking at the loop implementation. So, for now, let's just remember that there is such a thing as a loop frame that is part of the specification, and unless I say otherwise, the implicit loop frame allows all local variables to be modified.

11.1. Loop Implementations

If you've written loops before (which I *am* assuming you have), then you may be puzzled after reading the previous section, wondering if the loops I'm talking about are the loops you're familiar with. Yes, they are the same. But so

far, I've only talked about loop *specifications*, and the loops you've written are probably loop *implementations*. Let's combine these two notions. As I now bring in the familiar, don't you go forgetting about what you just learned about loop specifications—I'll be sure to remind you.

A loop derives its name from the fact that its specification is carried out through a repetition of steps. In particular, a loop repeats the execution of its *loop body* for as long as the loop guard holds. The loop body is the *implementation* of the loop. Syntactically, this implementation is given in curly braces after the loop specification.

11.1.0. Quotient modulus

As our first example, let's return to the loop in the previous section for computing the quotient and modulus of 191 and 7. Here is the specification for that loop, including an initializing assignment before the loop and an assertion after the loop:

```
x, y := 0, 191;
while 7 <= y
  invariant 0 <= y && 7 * x + y == 191
assert x == 191 / 7 && y == 191 % 7;
```

The loop body is a sequence of statements that maintains the loop invariant, given that the loop guard holds in its initial state. That is, starting from a state where both the invariant and the guard hold, the body is to produce a state where the invariant holds. Let's think of some ways we can do this for the quotient-modulus programs. That is, let's think of what we can write for the ? in the following Hoare triple:

```
{ { 0 <= y && 7 * x + y == 191 && 7 <= y } }
?
{ { 0 <= y && 7 * x + y == 191 } }
```

A loop body

One way to replace the `?` is to increase `x` by 1, and maybe we'll need to change `y` as well. Using the weakest-precondition computations for assignment in Section 2.3 and the bookkeeping notation for program proofs from Section 2.3.1, we have:

```
{ { 0 <= y && 7 * x + y == 191 && 7 <= y } }  
?  
{ { 0 <= y && 7 * (x + 1) + y == 191 } }  
x := x + 1  
{ { 0 <= y && 7 * x + y == 191 } }
```

The formula in the middle can be simplified:

```
7 * (x + 1) + y == 191  
=  
7 * x + 7 + y == 191
```

If we substitute `y - 7` for `y` in this formula, we get the same condition as at the start of the Hoare triple. So, we have:

```
{ { 0 <= y && 7 * x + y == 191 && 7 <= y } }  
{ { 0 <= y - 7 && 7 * x + 7 + (y - 7) == 191 } }  
y := y - 7  
{ { 0 <= y && 7 * x + 7 + y == 191 } }  
{ { 0 <= y && 7 * (x + 1) + y == 191 } }  
x := x + 1  
{ { 0 <= y && 7 * x + y == 191 } }
```

It's best to double-check this computation, especially because it's easy to get confused with the direction of the substitution. Starting from the *last* of these lines and reading *upward*, whenever we come across an assignment, we replace the variable being assigned with the right-hand side of the assignment (replacing `x` by `x + 1`, and replacing `y` by `y - 7`), and whenever there are two lines with no program statement in between, we need to check that the first of the two implies the second. (Review Section 2.3.1 as needed.)

That's how it's done, but maybe this felt a bit formal. Let's think about it again, precisely but informally. We are

supposed to write a statement that maintains the condition $7 * x + y == 191$ (along with various range conditions, yada yada). If we increase x by 1, then it's clear from this formula that we need to decrease y by 7. This is what it means to maintain an invariant.

Here's our program:

```
x, y := 0, 191;
while 7 <= y
  invariant 0 <= y && 7 * x + y == 191
  {
    y := y - 7;
    x := x + 1;
  }
assert x == 191 / 7 && y == 191 % 7;
```

If you look just at the loop body, it is not at all clear that this program computes the integer square root. But thanks to the loop invariants (and the help of the verifier), we can be sure that the loop establishes the method postcondition.

Leap to the answer

There's more than one way to implement the loop body for the quotient-modulus program. Here's a Hoare triple that shows a different way to maintain the invariant:

```
{ { 0 <= y && 7 * x + y == 191 && 7 <= y } }
{ { true } }
{ { 0 <= 2 && 7 * 27 + 2 == 191 } }
x, y := 27, 2
{ { 0 <= y && 7 * x + y == 191 } }
```

So, we have discovered another way to implement our loop:

```
x, y := 0, 191;
while 7 <= y
  invariant 0 <= y && 7 * x + y == 191
  {
    x, y := 27, 2;
  }
assert x == 191 / 7 && y == 191 % 7;
```

Going twice as fast

Let's dwell on this first program with a loop body some more, to consider something that does not work. How about we try to combine two loop bodies into one? Instead of incrementing x by 1 and decrementing y by 7, let's try incrementing x by 2 and decrementing y by 14. The Hoare triple that expresses the proof obligation of the loop implementation is then:

```
{ { 0 <= y && 7 * x + y == 191 && 7 <= y } }
{ { 14 <= y && 7 * x + y == 191 } } // error: does not follow
                                     // from the previous
annotation
{ { 0 <= y - 14 && 7 * (x + 2) + (y - 14) == 191 } }
x, y := x + 2, y - 14
{ { 0 <= y && 7 * x + y == 191 } }
```

where I added two more annotations above the proposed assignment statement. Our “quicker” update does maintain the relation $7 * x + y == 191$. However, it gives rise to the proof obligation $14 <= y$, which does not follow from the top line. So, this attempt to speed up the loop is not correct.

Exercise 11.3.

Introduce an **if** statement in the body of the loop, where one branch is $x, y := x + 2, y - 14$ and the other is $x, y := x + 1, y - 7$. What guard condition do you need in the **if** statement to make the loop correct?

The easiest way to maintain anything

If all we have to do is establish a condition that already holds, then there's no simpler way to proceed than to do nothing. Indeed, this would maintain any loop invariant. But no number of repetitions of such a loop body would get us any closer to reaching a state where the loop guard no longer holds. So, unless the guard didn't hold in the first place, such a loop would never terminate.

Loop termination is important, but I'll postpone the details until Section 11.2.

11.1.1. Being formal about maintaining the loop invariant

For a loop

```
while B
  invariant J
{
  Body
}
```

the proof obligation for maintaining the loop invariant is captured in Floyd logic by the following Hoare triple:

```
{{ J && B }}
Body
{{ J }}
```

Note that there is no repetition of `Body` in this proof obligation. The Hoare triple gives you the assumption `J && B` and that's it! If you need something stronger to prove that `J` holds after `Body`, then you need to strengthen the invariant. Of course, if you strengthen the loop invariant from `J` to some `J'`, then you also need to prove that `J'` (not just `J`) holds after `Body`.

In addition to this proof obligation about maintaining the invariant, there's a loop proof obligation about termination. We'll see it in Section 11.2.

11.1.2. Computing sums

Let's do one more example from the previous section, the one that sums up the first 33 natural numbers. Here is the specification:

```
while n != 33
  invariant s == n * (n - 1) / 2
```

To implement this loop, we need to fill in the `?` in the following Hoare triple:

```
{{ s == n * (n - 1) / 2 && n != 33 }}
?
```

$\{ \{ s == n * (n - 1) / 2 \} \}$

Incrementing n by 1 seems reasonable. Let's work out what change that entails for s :

```
(s == n * (n - 1) / 2)[n := n + 1]
= { substitution }
s == (n + 1) * n / 2
=
s == (n*n + n) / 2
=
s == (n*n - n + 2*n) / 2
=
s == (n*n - n) / 2 + 2*n / 2
=
s == n * (n - 1) / 2 + n
```

Since, by the loop invariant, s starts off as $n * (n - 1) / 2$, we see that we need to increment s by n . So, our whole program, including an assignment to n and s before the loop that establishes the invariant initially, is:

```
n, s := 0, 0;
while n != 33
  invariant s == n * (n - 1) / 2
  {
    s := s + n;
    n := n + 1;
  }
```

Exercise 11.4.

Write a different (but still correct) initializing assignment for the loop above.

Exercise 11.5.

Write an initializing assignment and a loop implementation for the following loop specifications:

a) **while** $x < 300$ b) **while** $x \% 2 == 1$
 invariant $x \% 2 == 0$ **invariant** $0 \leq x \leq 100$

Exercise 11.6.

Consider the following program fragment:

```
x := 0;
while x < 100
{
  x := x + 3;
}
assert x == 102;
```

Write a loop invariant that holds initially, is maintained by the loop body, and allows you to prove the assertion after the loop.

11.1.3. Loop frames inferred

In Section 11.0.5, I mentioned that a loop specification also needs a loop frame, that is, an indication of what the loop is allowed to modify and what it must leave unchanged. At this time, I have two things to say about that.

One thing is that in-parameters in Dafny are immutable. Thus, it is clear that any in-parameter mentioned in a loop specification (or, for that matter, any in-parameter omitted from a loop specification) is not changed by the loop.

The other thing is that Dafny does have a way to work out which local variables (and out-parameters) are inside the loop frame. It does so not from an explicit part of the loop specification, but instead from peering into the loop implementation. More precisely, a local variable is inside the loop frame if the loop body syntactically contains an assignment to the variable. What this means in practice is that once you've written the loop specification and loop body, the verifier knows that any local variable that's not assigned in the loop body will have the same value after the loop as it did before the loop. For any local variable that your loop does assign to, you'll want to say something about that variable in the loop invariant—otherwise, the verifier won't have any information about it after the loop.

The following example illustrates:

```
method LoopFrameExample(X: int, Y: int)
  requires 0 <= X <= Y
```

```

{
  var i, a, b := 0, X, Y;
  while i < 100 {
    i, b := i + 1, b + X;
  }
  assert a == X;
  assert Y <= b; // not provable without an invariant for the
loop
}

```

Since there is no assignment to *a* in the loop body and *x* is an in-parameter (so, of course there's no assignment to *x*, either), the assertion *a* == *x* is provable after the loop. On the contrary, *b* is assigned in the loop body, so without a loop invariant about *b*, the verifier knows nothing about *b* after the loop. To prove the second assertion, you need to add

invariant *Y* <= *b*

or

invariant *b* == *Y* + *i* * *X*

to the loop specification. Such a loop invariant is provable without the further invariant that $0 \leq x$, since *x* is an in-parameter.

11.2. Loop Termination

There's one more proof obligation for loop implementations, which is to show termination. Of course, the loop body itself must be shown to terminate. For example, any recursive call from the loop body to the enclosing method must be shown to terminate, in the way we saw back in Chapter 3. In addition, we must show that the repetitions of the loop body eventually come to an end. In other words, we want to prevent infinite repetitions, or *infinite looping*.

Infinite looping is prevented by labeling the loop iterations by elements of a well-founded order, just as

infinite recursion is prevented by labeling the activation records of functions and methods by elements of a well-founded order. But instead of showing a decrease from a caller to a callee, we show a decrease from one iteration to the next. This is accomplished by computing the next iteration's label at the time the loop's "back edge" is taken. Stated more simply, we check that the body causes a decrease of the loop's termination metric.

The loop's termination metric is declared using a **decreases** clause. For a loop

```
while B
  invariant J
  decreases D
{
  Body
}
```

the proof obligation of termination, phrased as a Hoare triple, is:

```
{ { J && B } }
ghost var d0 := D;
Body
{ { d0 > D } }
```

As a convenient way to record the value of D on entry to the loop body, so that we can refer to it again in the Hoare-triple postcondition, I introduced a ghost local variable d_0 . The Hoare triple says that we must show that d_0 exceeds the value of D after the loop body.

11.2.0. Termination of quotient modulus

In Section 11.1.0, we considered several implementations of the loop that is specified to compute the quotient and modulus of 191 and 7. Let's check termination of some attempted implementations of that loop specification.

Simple body

The body of our most straightforward program is

```
y := y - 7;  
x := x + 1;
```

Since y is being decreased, let's choose it as our termination metric. We then have to prove

```
{ { 0 <= y && 7 * x + y == 191 && 7 <= y } }  
ghost var d0 := y;  
y := y - 7;  
x := x + 1;  
{ { d0 > y } }
```

We're working with integers, so by the definition of the well-founded order $>$ on integers (Section 3.2), we have to prove:

```
y < d0 && 0 <= d0
```

The first conjunct follows because $y == d_0 - 7$, and the second conjunct follows from the invariant $0 \leq y$ (and it also follows from the guard $7 \leq y$). Done!

Here's how we write the whole loop—specification, implementation, and initializing assignment:

```
x, y := 0, 191;  
while 7 <= y  
  invariant 0 <= y && 7 * x + y == 191  
  decreases y  
{  
  y := y - 7;  
  x := x + 1;  
}
```

Quick body

Recall the leap-to-the-answer implementation of the quotient-modulus loop:

```
x, y := 27, 2;
```

What is it that decreases here? To see it more clearly, here is the proof obligation in Hoare-triple form, where the two

question marks should be filled in with the same expression:

```
{ { 0 <= y && 7 * x + y == 191 && 7 <= y } }  
ghost var d0 := ?;  
x, y := 27, 2;  
{ { d0 > ? } }
```

Notice that the Hoare triple allows us to start with $7 \leq y$. Therefore, **decreases** y works to prove termination of this loop, too.

Switching directions

Here is an attempt at implementing the loop that we didn't try before:

```
while 7 <= y  
  invariant 0 <= y && 7 * x + y == 191  
{  
  x, y := x - 1, y + 7;  
}
```

Instead of incrementing x and decrementing y , this loop body does it the other way around. This still maintains the invariant. But to prove termination, we would need a lower bound on x or an upper bound on y . We have neither. Indeed, if y starts off satisfying the guard, then it always will, so this program really won't terminate.

We can *prove* that this loop does *not* terminate. One way to do that is to initialize y to satisfy the guard and to add $7 \leq y$ as an invariant:

```
x, y := 0, 191;  
while 7 <= y  
  invariant 0 <= y && 7 * x + y == 191  
  invariant 7 <= y  
{  
  x, y := x - 1, y + 7;  
}
```

(Analogous to multiple **requires** and **ensures** clauses, writing multiple **invariant** clauses is the same as writing one that

conjoins all the conditions.) Though it does not terminate, this program does maintain the invariant. More generally, whenever we have a situation where the invariant implies the guard, that means the loop will not terminate.³

11.2.1. Default decreases clauses for loops

In Section 3.4, I explained that Dafny provides a default **decreases** clause for recursive functions or methods, if you don't supply one. Dafny also supplies a default **decreases** clause for any loop where you don't explicitly write one.

Dafny's rule for constructing a default **decreases** clause is foolishly simple and surprisingly effective. In a nutshell, if the loop guard is an arithmetic comparison $E < F$ or $E \leq F$, then the default is

decreases $F - E$

And if the loop guard is an arithmetic comparison $E \neq F$, then the default is the absolute difference between E and F :

decreases **if** $E < F$ **then** $F - E$ **else** $E - F$

Even with these simple rules, you'll find that Dafny will prove a large share of your terminating loops without you having to do anything.

You don't need to know the exact rules for how the default **decreases** clause is constructed, but you can easily inspect what Dafny came up with by hovering over the **while** keyword of a loop in your Dafny IDE. If you don't like the default **decreases** clause (which probably happens only if the default is not good enough to prove termination of your loop), you can just supply your own.

11.3. Summarizing the Loop Rule

Let's collect all the proof obligations of a loop in one place. For a loop

```

while B
  invariant J
  decreases D
{
  Body
}

```

the context that uses the loop views it as

```

{{ J }}
while B
  invariant J
  decreases D
{
  Body
}
{{ J && !B }}

```

which has nothing to do with Body (or with D, for that matter), and the correctness of the loop implementation comes down to proving

```

{{ J && B }}
ghost var d0 := D;
Body
{{ J && d0 > D }}

```

which has nothing to do with the fact that the loop construct will repeatedly execute the Body.

Exercise 11.7.

For each of the following methods, write a loop invariant and an (explicit) **decreases** clause that allow you to prove the method.

```

method UpWhileLess(N: int) returns (i: int)
  requires 0 <= N
  ensures i == N
{
  i := 0;
  while i < N {
    i := i + 1;
  }
}

```

```

method UpWhileNotEqual(N: int) returns (i: int)
  requires 0 <= N
  ensures i == N
{
  i := 0;
  while i != N {
    i := i + 1;
  }
}

method DownWhileNotEqual(N: int) returns (i: int)
  requires 0 <= N
  ensures i == 0
{
  i := N;
  while i != 0 {
    i := i - 1;
  }
}

method DownWhileGreater(N: int) returns (i: int)
  requires 0 <= N
  ensures i == 0
{
  i := N;
  while 0 < i {
    i := i - 1;
  }
}

```

You should expect to have to write your loop invariants, but Dafny does a wee bit of inference for you. If you're surprised that you don't need to write any loop invariant for two of these examples, it's because it infers them for you. Still, go ahead and write them yourself, too.

11.4. Integer Square Root

I'll wrap up this chapter with one more example, the integer-square-root program that we got a brief taste of in Section 11.0.5. We'll start with a method specification:

```
method SquareRoot(N: nat) returns (r: nat)  
  ensures  $r * r \leq N < (r + 1) * (r + 1)$ 
```

Note that I restricted N to be a natural number, so that we won't get stuck trying to compute the square root of a negative number. I also declared r to be of type `nat`, which saves us from having to write $0 \leq r$ in the postcondition and, later, in the loop invariant.

11.4.0. A technique for solving the problem

During the course of the next chapters, we'll come across a number of techniques for designing loops to solve problems of certain forms. Here is the first such technique.

Loop design technique 11.0. (Omit a conjunct)

To solve a problem of the form $A \ \&\& \ B$, pick the loop invariant to be A and pick the loop guard to be $\neg B$. That is, use a loop specification like this:

```
while  $\neg B$   
  invariant  $A$ 
```

The square-root problem consists of two conjuncts. As we had seen before, let's use $r * r \leq N$ as the loop invariant and let's use the negation of $N < (r + 1) * (r + 1)$ as the loop guard. To establish the invariant initially, we can set r to 0:

```
{  
  r := 0;  
  while  $(r + 1) * (r + 1) \leq N$   
    invariant  $r * r \leq N$ 
```

We can implement this loop by an increment to r .

```
  {  
    r := r + 1;  
  }  
}
```

That's all there is! The resulting program is a linear search for the right value of r .

11.4.1. A more efficient program

Every iteration of our program evaluates the expression $(r + 1) * (r + 1)$. If efficiency is a concern, we may instead consider computing the expression from the value of the expression in a previous iteration. Invariants can help us make sure we transform the program correctly. We'll do it through what I like to call a *wish*. It goes like this:

I wish we had a variable s whose value was always $(r + 1) * (r + 1)$. If we did, we could change the loop guard to be $s \leq N$. Let's keep track of our wish in a loop invariant, because a loop invariant gives us proof obligations that will prompt us to turn the wish into reality.

invariant $s == (r + 1) * (r + 1)$

Very well. Now that our loop specification has this invariant, one of our obligations is to establish it initially. Since r is initially 0, the loop invariant directly tells us that we must initialize s to 1.

var $s := 1;$

The other proof obligation is to maintain the invariant in the body. The body increments r , so we can compute the weakest precondition of that assignment with respect to our new invariant. That is, we will *work backward* from the desired loop invariant. This will tell us what we have to establish about s before the assignment to r .

```
{ {  $s == (r + 1 + 1) * (r + 1 + 1)$  } }  
 $r := r + 1$   
{ {  $s == (r + 1) * (r + 1)$  } }
```

By arithmetic, we have:

$$\begin{aligned} & (r + 1 + 1) * (r + 1 + 1) \\ = & \\ & r * r + 4 * r + 4 \\ = & \\ & r * r + 2 * r + 1 + 2 * r + 3 \\ = & \\ & (r + 1) * (r + 1) + 2 * r + 3 \end{aligned}$$

This is good news, because we know, by the invariant, that s is $(r + 1) * (r + 1)$ on entry to the loop body. So, all we need to do is increment s by $2 * r + 3$.

```
s := s + 2*r + 3;
```

We had to think about establishing the new invariant initially and about maintaining it.

So, here is our final program to compute the integer square root:

```
method SquareRoot(N: nat) returns (r: nat)
  ensures r * r <= N < (r + 1) * (r + 1)
{
  r := 0;
  var s := 1;
  while s <= N
    invariant r * r <= N
    invariant s == (r + 1) * (r + 1)
  {
    s := s + 2*r + 3;
    r := r + 1;
  }
}
```

The technique we used in constructing the program is captured as follows:

Loop design technique 11.1. (Programming by wishing)

If a problem can be made simpler by having a precomputed quantity Q , then introduce a new variable q with the intention of establishing and maintaining the invariant $q == Q$.

I'll expand on this technique in Section 12.1.

11.5. Summary

A loop has two parts, a specification and an implementation. The loop specification itself has three

parts:

- the loop guard, which controls when the loop's repetition continues
- the loop invariant, which constrains the state the loop iterations may explore
- the loop frame, which usually is implicit

When you *use* a loop specification, you have to meet the loop-use proof obligation (that is, you have to prove that the loop invariant holds initially). In return, you get to find out that, after the loop, the invariant and the negation of the loop guard hold.

The loop implementation is a block statement known as the loop body. When you *implement* a loop, you must show that the loop body maintains the loop invariant from a state where the loop guard holds. To ensure that the loop's repetitions eventually end, you must also show that the loop body decreases some termination metric.

Many loop invariants fall into common patterns. In this chapter, I identified two such patterns (as Loop Design Techniques), and I will continue to point out such patterns in chapters to come. These patterns give you techniques that you can try when designing your own loops and invariants to solve problems.

Loops are a hallmark of imperative programming and loop invariants are a critical part of program proofs. Now that you have seen several example loops and invariants, it's a good idea to review the body-less loop specifications at the beginning of this chapter. If on your first reading you had found it strange to reason about a loop by *ignoring* its body, you may now start to appreciate how the loop specification separates the concerns of reasoning about the *use* of the loop from the concerns of reasoning about the *implementation* of the loop.

Exercise 11.8.

Using datatype `List` and function `Length` from Chapter 6, write a loop that constructs a `List` with a given value `d` repeated `n` times. Prove as a postcondition of the method that the returned list has length `n`.

Exercise 11.9.

Write a method `Duplicate` that takes a `List` (see Chapter 6) and returns a list twice as long. The elements of the new list can be anything you'd like—this exercise is concerned only with the length of the list. Prove the postcondition that the returned list is indeed twice as long as the original.

Notes

Invariants are ubiquitous in program proofs. For example, we saw data-structure invariants in Chapter 10 and we'll see object invariants in Chapter 16. This chapter introduced loop invariants. From a learning perspective, loops offer a friendly playground for internalizing the concept of invariants, because you don't have to write many lines of code to experiment with different loop invariants. Use the simple examples of this chapter and the next to develop an understanding of how invariants are used to reason about programs without needing to know how many iterations a loop performs.

Although loop invariants are the most prevalent way to reason about loops, there are other ways to reason about repeated, imperative behavior. A sentiment about this is captured in a classic paper by Eric Hehner [[60](#)].

³Unless there is another exit from the loop, as would be facilitated by a `return` or `break` statement.

Chapter 12

Recursive Specifications, Iterative Programs



We often use functions in method specifications, and functions are built up using recursion. If we implement such a method using a loop, then the loop invariant is also going to be expressed using functions. In this chapter, I give some examples that demonstrate issues that arise when you connect recursion and iteration in this way.

12.0. Iterative Fibonacci

In Section 3.1, I defined the famous Fibonacci sequence

0, 1, 1, 2, 3, 5, 8, ...

by a recursive function:

```

function Fib(n: nat): nat {
  if n < 2 then n else Fib(n - 2) + Fib(n - 1)
}

```

It is simple to inspect this function to see that it defines the Fibonacci sequence correctly: it starts with 0 and 1, and all subsequent numbers are defined as the sum of the preceding two Fibonacci numbers. Attempting to compute Fibonacci numbers this way at run time, however, would be disastrously slow. Let's write a far more efficient version ($O(n)$ addition operations for $\text{Fib}(n)$) using a loop.

We use function Fib to specify our method, ComputeFib:

```

method ComputeFib(n: nat) returns (x: nat)
  ensures x == Fib(n)

```

Note that Fib is just part of the specification and will not be evaluated when ComputeFib is executed. In other words, the **ensures** clause is ghost, explains what we want to verify, and is erased by the compiler. In fact, since we don't intend for function Fib to be used directly in a compiled program, it would make sense to declare it as ghost; indeed, in the rest of this chapter, I will declare such specification-only functions as ghost.

We will implement ComputeFib using a loop with loop index i , which will range from 0 through n . The initialization and loop specification thus looks like this:

```

{
  x := 0;
  var i := 0;
  while i != n
    invariant 0 <= i <= n
    invariant x == Fib(i)
}

```

This loop specification says that, after the loop, $i == n$ and $x == \text{Fib}(n)$. This tells us we're on the right track—we have reduced the problem of implementing the method to a problem of implementing the loop.

What we just did is a cornerstone technique of designing a loop specification from a desired postcondition: we replaced the constant n in the postcondition with the loop index i . Here is a summary of that technique:

Loop design technique 12.0. (Replace constant by variable)

For a loop to establish a condition $\dots c \dots$, where c is an expression that is held constant throughout the loop, use a variable k that the loop changes until it equals c , and make $\dots k \dots$ a loop invariant.

Applied to our program, the constant c in this loop design technique is our inparameter n and the variable k is our loop index i .

As part of implementing the loop, we immediately see that an assignment

```
i := i + 1;
```

will maintain the first loop invariant and will let us prove termination.

The loop body must also update x . The idea that will let us do this efficiently is to keep track of not just one Fibonacci number, but two consecutive Fibonacci numbers. We'll store one in x , and let's store the other in a local variable y . But now we have a choice. We can either design our loop with

```
invariant x == Fib(i) && y == Fib(i + 1)
```

or design it with

```
invariant y == Fib(i - 1) && x == Fib(i)
```

The latter is not well-defined when i is 0 (since Fib is defined to take a **nat**). We can salvage the situation by writing that invariant as

```
invariant (i == 0 || y == Fib(i - 1)) && x == Fib(i)
```

This is not a good idea. If we start off with a more complicated invariant, chances are we'll end up with a more complicated program. The loop invariant captures the design of the loop we're writing, so by writing down the loop invariant first, before attempting to write the loop body, we can detect some design flaws earlier.

So, let's use the invariant where y is the Fibonacci number after x . We introduce y and initialize it to establish the invariant:

```
var y := 1;
```

We have already decided to increment i in the loop, so to obtain the condition we need before the update of i , we work backward from the loop invariant. As we've seen before, we get there by replacing i by $i + 1$ in the condition we want after the assignment to i . This gives us

```
{ { x == Fib(i) && y == Fib(i + 1) && i != n } }  
?  
{ { x == Fib(i + 1) && y == Fib(i + 1 + 1) } }  
i := i + 1  
{ { x == Fib(i) && y == Fib(i + 1) } }
```

where the $?$ shows where we need to update x and y . We want to assign x so that it equals $\text{Fib}(i + 1)$. This is easy, since we can see that y has that value before the $?$. At the same time, we want to assign y so that it equals $\text{Fib}(i + 2)$. By the definition of Fib , this is the same as $\text{Fib}(i) + \text{Fib}(i + 1)$, and those two terms are found in x and y before the $?$. So, we now know that the $?$ should be replaced by

```
x, y := y, x + y;
```

Our implementation of `ComputeFib` is complete. For the sake of reminding you of what we just did, I'm showing you the whole method here, including Hoare-triple annotations we used in the proof (except the pieces for termination, which Dafny takes care of automatically in this program):

```
method ComputeFib(n: nat) returns (x: nat)  
  ensures x == Fib(n)
```

```

{
  {{ true }}
  x := 0;
  var i := 0;
  var y := 1;
  {{ 0 ≤ i ≤ n && x == Fib(i) && y == Fib(i + 1) }}
  while i != n
    invariant 0 ≤ i ≤ n
    invariant x == Fib(i) && y == Fib(i + 1)
    {
      {{ 0 ≤ i ≤ n && x == Fib(i) && y == Fib(i + 1) && i != n }}
    }
    {{ 0 ≤ i < n && x == Fib(i) && y == Fib(i + 1) }}
    {{ 0 ≤ i < n && y == Fib(i + 1) && x + y == Fib(i) + Fib(i + 1) }}
    x, y := y, x + y;
    {{ 0 ≤ i < n && x == Fib(i + 1) && y == Fib(i) + Fib(i + 1) }}
    {{ 0 ≤ i < n && x == Fib(i + 1) && y == Fib(i + 2) }}
    {{ 0 ≤ i + 1 ≤ n && x == Fib(i + 1) && y == Fib(i + 1 + 1) }}
    i := i + 1;
    {{ 0 ≤ i ≤ n && x == Fib(i) && y == Fib(i + 1) }}
  }
  {{ 0 ≤ i ≤ n && x == Fib(i) && y == Fib(i + 1) && i == n }}
  {{ x == Fib(n) }}
}

```

Exercise 12.0.

(Advanced) Write an imperative program that computes $\text{Fib}(n)$ using only $O(\log n)$ arithmetic operations.

12.1. Fibonacci Squared

In this section, I employ the powerful technique of *programming by wishing* (Loop Design Technique 11.1) to let invariants guide us in gradually improving a program. We already used this technique in introducing a little optimization (Section 11.4.1).

We'll implement a method that computes the square of a Fibonacci number:

```
method SquareFib(N: nat) returns (x: nat)  
  ensures x == Fib(N) * Fib(N)
```

Suppose our program will execute on a machine that has no support for multiplication, only addition. So, let's implement the method without using multiplication (but we are free to use multiplication in specifications, since they are not compiled). I know we just implemented `ComputeFib`, where we made certain design decisions, but let's start afresh here to see how the wishing technique plays out.

12.1.0. A simple start

We know the technique of replacing a constant by a variable, so let's introduce a loop index n and write a loop invariant where the postcondition's constant N is replaced by n :

```
{  
  x := 0;  
  var n := 0;  
  while n != N  
    invariant 0 <= n <= N  
    invariant x == Fib(n) * Fib(n)  
}
```

The loop body will increment n , so let's work backward to figure out what to assign to x :

```
{{ x == Fib(n + 1) * Fib(n + 1) }}  
n := n + 1  
{{ x == Fib(n) * Fib(n) }}
```

Apparently, we need to set x to the square of $\text{Fib}(n + 1)$ before we update n . In a situation like this, it is often possible to expand the function applied to $n + 1$ and obtain another expression in terms of the function applied to n . But we can't do that with `Fib`, because its recurrence is defined in terms of the previous *two* values, $\text{Fib}(n - 1)$ and $\text{Fib}(n)$, and if $n - 1$ is negative, we can't call `Fib` on it.

12.1.1. A wish

No worries. Let's just wish we had the square of $\text{Fib}(n + 1)$ already computed in a variable, say y . Recording our wish as an invariant and finding an initial value for y that satisfies that invariant, our updated program is

```
x := 0;
var n, y := 0, 1;
while n != N
  invariant 0 <= n <= N
  invariant x == Fib(n) * Fib(n)
  invariant y == Fib(n + 1) * Fib(n + 1)
{
  x, y := y, ?;
  n := n + 1;
}
```

Notice that our new program uses y in the assignment to x , which was precisely the reason we wished for such a y .

How should we assign y in the loop body to maintain the invariant? Let's work backward again:

```
{ { y == Fib(n + 2) * Fib(n + 2) } }
n := n + 1
{ { y == Fib(n + 1) * Fib(n + 1) } }
```

This time, we are able to apply the definition of Fib , because we're starting with an argument that is at least 2. Thus, we rewrite the expression $\text{Fib}(n + 2) * \text{Fib}(n + 2)$ into parts that we have already computed. We could do this calculation on paper, but we might as well get the verifier's help in making sure we're doing the calculation correctly. To do this on the computer, you must omit the nonsensical assignment $y := ?$ that I wrote above—once we have figured out what to assign to y , you will include a proper assignment of y into the program. You can then write the following proof calculation immediately inside the loop body (remember proof calculations from Chapter 5 and Part 1?):

```
calc {
```

```

    Fib(n + 2) * Fib(n + 2);
== // def. Fib
    (Fib(n) + Fib(n + 1)) * (Fib(n) + Fib(n + 1));
== // cross multiply
    Fib(n) * Fib(n) + 2 * Fib(n) * Fib(n + 1) +
    Fib(n + 1) * Fib(n + 1);
== // invariant
    x + 2 * Fib(n) * Fib(n + 1) + y;
}

```

This worked out well. The **calc** statement proved that the square of $\text{Fib}(n + 2)$ can be expressed as the sum of three terms. The first term is the square of $\text{Fib}(n)$, which, by the loop invariant, is already stored in x . The third term is the square of $\text{Fib}(n + 1)$, which on entry to the loop body is stored in y . Wishing we had a variable, say k , for the middle term, our assignment to y would be

```
y := x + k + y;
```

The **calc** statement we introduced as a scratch pad has served its function, so you can now delete it. However, my recommendation is to leave it there until we're done with the whole method. In fact, you may even choose to keep the calculation permanently, since it provides a readable explanation as to why we introduced k . If you do keep it for documentation, you can add

```

== // invariant
    x + k + y;

```

to the end of it, once you've introduced k .

12.1.2. Another wish

Our wish above introduced k with the invariant

```
invariant k == 2 * Fib(n) * Fib(n + 1)
```

and initial value 0. The value we need to assign k to match the increment of n is $2 * \text{Fib}(n + 1) * \text{Fib}(n + 2)$. Let's use another **calc** statement just inside the loop body to try to simplify this expression:


```

calc {
  2 * Fib(n + 1) * Fib(n + 2);
== // def. Fib
  2 * Fib(n + 1) * (Fib(n) + Fib(n + 1));
== // distribute arithmetic
  2 * Fib(n + 1) * Fib(n) + 2 * Fib(n + 1) * Fib(n + 1);
}

```

This is our lucky day. The first of these terms is the value of k on entry to the loop body and the second term is twice the value of y . This tells us that we need the update $k := k + y + y$.

In summary, here is the full method body of our completed program:

```

x := 0;
var n, y, k := 0, 1, 0;
while n != N
  invariant 0 <= n <= N
  invariant x == Fib(n) * Fib(n)
  invariant y == Fib(n + 1) * Fib(n + 1)
  invariant k == 2 * Fib(n) * Fib(n + 1)
{
  x, y, k := y, x + k + y, k + y + y;
  n := n + 1;
}
}

```

The **calc** statements we wrote during the development were placed immediately on entry to the loop body, but I have deleted them from this final program.

Notice that the simultaneous assignment is necessary, since the variables are updated in terms of each other. Alternatively, we could have introduced temporary variables and done the assignments sequentially:

```

var prevX := x;
var prevY := y;
x := prevY;
y := prevX + k + prevY;
k := k + prevY + prevY;

```

12.1.3. Reflection

I hope the final program leaves you with a sense of wonder. I find it marvelous that, in the end, our wishes were all resolved by values we already had. This won't always be the case, but wishing is a technique worth trying.

Exercise 12.1.

Implement

```
method Cube(n: nat) returns (c: nat)
  ensures c == n * n * n
```

with a loop that iterates n times and only does addition (no multiplication).

12.2. Powers of 2

We can define a function that computes 2^n . The function uses the facts $2^0 == 1$ and, for any other exponent n , $2^n == 2 * 2^{n-1}$:

```
ghost function Power(n: nat): nat {
  if n == 0 then 1 else 2 * Power(n - 1)
}
```

I intentionally declared this function as ghost, since we will use it only in specifications.

Here is a method for computing $\text{Power}(n)$:

```
method ComputePower(n: nat) returns (p: nat)
  ensures p == Power(n)
```

Next, we'll consider the method's implementation.

12.2.0. The usual invariant

It is straightforward to write a loop that implements this method, so let's go through the steps once more to make sure you feel comfortable with how we're building up our invariants and programs.

The idea will be to use a loop where the loop index, i , goes from 0 through n , maintaining the condition that $p ==$

Power(i). With i starting off at 0, we initialize p to 1 in order to establish the invariant on entry to the loop.

```
{
  p := 1;
  var i := 0;
  while i != n
    invariant 0 <= i <= n
    invariant p == Power(i)
}
```

Exercise 12.2.

How does the verifier respond if you

- a) change $p := 1$ to $p := 2$?
- b) change $p := 1$ to $p := 2$ and change the invariant to $p == \text{Power}(i + 1)$?
- c) change $p := 1$ to $p := 2$ and change $i := 0$ to $i := 1$?
- d) do (c) and also change $i != n$ to $i < n$?

We'll increment i by 1 in the loop body. This ensures termination and lets us annotate the program to focus in on the update of p in the loop:

```
{ { 0 <= i < n && p == Power(i) } }
?
{ { 0 <= i + 1 <= n && p == Power(i + 1) } }
i := i + 1
{ { 0 <= i <= n && p == Power(i) } }
```

Here and from now on, I'm immediately simplifying $0 \leq i \leq n$ && $i \neq n$ to the half-open interval $0 \leq i < n$.

You can see from around the question mark in the annotations above that p is $\text{Power}(i)$ on entry to the loop body and we need to change it to $\text{Power}(i + 1)$. By its definition, $\text{Power}(i + 1)$ is $2 * \text{Power}(i)$, so our update of p needs to be

```
p := 2 * p;
```

12.2.1. An alternative invariant

The invariant we wrote about p focuses on what has been computed so far. This is not the only way we could have written the invariant. An alternative way is to focus on what is left to do. In words, I'll say "if you multiply p by $\text{Power}(n - i)$, you'll get what we're looking for". As a loop specification, we write this as

```
p := 1;
var i := 0;
while i != n
  invariant 0 <= i <= n
  invariant p * Power(n - i) == Power(n)
```

The initial assignments to p and i establish the invariant. After the loop, i equals n , so we have

```
p * Power(0) == Power(n)
```

By the definition of $\text{Power}(0)$, the left-hand side of this equation simplifies to p , so the method's postcondition follows.

You can't always phrase your invariants in this way, but other times, this is the only way to phrase some invariants.

Loop design technique 12.1. (Yet to be done)

If you're trying to solve a problem of the form $p == F(n)$, you may be able to do so with a loop index i satisfying $0 \leq i \leq n$ and either the what-has-been-done invariant

```
invariant p == F(i)
```

or the what's-yet-to-be-done invariant

```
invariant p  $\oplus$  F(n - i) == F(n)
```

where $\oplus$ is some kind of combination operation.

Exercise 12.3.

Generalize function `Power` to any base. That is, define a function

```
ghost function Exp(b: nat, n: nat): nat
```

that evaluates to b^n . Write a method `ComputeExp` that computes $\text{Exp}(b, n)$ in $\mathcal{O}(n)$ iterations.

Exercise 12.4.

Prove the following two lemmas about function `Exp` from Exercise 12.3:

```
lemma ExpAddExponent(b: nat, m: nat, n: nat)
  ensures Exp(b, m + n) == Exp(b, m) * Exp(b, n)
lemma ExpSquareBase(b: nat, n: nat)
  ensures Exp(b * b, n) == Exp(b, 2 * n)
```

Exercise 12.5.

(Advanced) Write a method `FastExp` that computes function `Exp` from Exercise 12.3 with only $\mathcal{O}(\log n)$ iterations.

Some hints and a caution are in order. The hints are to use a loop invariant of the what's-yet-to-be-done form, to make use of the lemmas in Exercise 12.4, and to use the following fact about `div` and `mod`, for any integer k :

$$k == 2 * (k / 2) + k \% 2$$

The caution is that this exercise involves non-linear arithmetic. In linear arithmetic, the only forms of multiplication are by constants, like $3 * x$, which are really just a nice notation for repeated additions, like $x + x + x$. Non-linear arithmetic means that there are multiplications that involve more than one variable occurrence, like $x * y$. Non-linear arithmetic puts a heavier burden on the mechanical verifier. For you, that translates to requiring more patience. Likely, you'll have to wait several seconds for the verifier to complete, and to get the proof to go through, you may have to help the verifier along in small steps with some assertions about arithmetic that you and I have taken for granted since grade school.

12.3. Sums

For the next example, I'm using a function F on integers. The example doesn't depend on what F does, so I'll just leave off its body. (Logicians would call this an *uninterpreted function*. If you want to compile and run this example, add a body to F . But to just verify the program, you won't need a body.)

```
function F(x: int): int
```

Sidebar 12.0

Half-open intervals are nice to work with. The number of values for i in the half-open interval

$$2 \leq i < 7$$

is $7 - 2$, that is, 5. Likewise, since ranges in Dafny are written with half-open intervals, the number of elements in the subsequence $a[2..7]$ is $7 - 2$. Here, I'm writing summations over half-open intervals, like

$$2^2 + 3^2 + 4^2 + 5^2 + 6^2 = \sum_{i=2}^{<7} i^2$$

So, to get the size of a half-open interval, subtract the lower bound from the upper bound. If the interval starts at 0, the size of the interval simply equals the upper bound. For example, there are 23 values for i in

$$0 \leq i < 23$$

12.3.0. Summing up and down

Function F can be applied to any integer. Let's apply it to a range of integers from a lower bound lo to an upper limit hi and add up what we get. In math notation, this is written as the summation

$$\sum_{i:=lo}^{<hi} F(i)$$

Without a built-in construct for Σ , we'll have to define our own recursive function for this expression. There are two ways to do that—the recursive calls can move the lower bound upward:

```
ghost function SumUp(lo: int, hi: int): int
  requires lo <= hi
  decreases hi - lo
{
  if lo == hi then 0 else F(lo) + SumUp(lo + 1, hi)
}
```

or they can move the upper bound downward:

```
ghost function SumDown(lo: int, hi: int): int
  requires lo <= hi
  decreases hi - lo
{
  if lo == hi then 0 else SumDown(lo, hi - 1) + F(hi - 1)
}
```

Note that we need explicitly to supply the **decreases** clause for both of these functions.

In math formulas, the definitions through these two **else** branches express

$$\sum_{i:=lo}^{<hi} F(i) = F(lo) + \sum_{i:=lo+1}^{<hi} F(i)$$

and

$$\sum_{i:=lo}^{<hi} F(i) = \left(\sum_{i:=lo}^{<hi-1} F(i) \right) + F(hi - 1)$$

If I wrote them correctly, we expect the two functions to return the same result. Let's prove that to be so.

```
lemma SameSums(lo: int, hi: int)
  requires lo <= hi
  ensures SumUp(lo, hi) == SumDown(lo, hi)
  decreases hi - lo
{
  if lo != hi {
    PrependSumDown(lo, hi);
  }
}

lemma PrependSumDown(lo: int, hi: int)
  requires lo < hi
  ensures F(lo) + SumDown(lo + 1, hi) == SumDown(lo, hi)
  decreases hi - lo
{
}
```

All appeals to the induction hypotheses of these two lemmas are handled by Dafny's automatic induction, but for this to work out, you do need to supply the appropriate **decreases** clauses (which are the same as for the functions involved). If you forgot to do so, you would get an error that the verifier can't prove the lemmas. You would then discover the need for these **decreases** clauses when you tried to call the lemmas recursively in your manual proofs.

Exercise 12.6.

What are the default **decreases** clauses for functions `SumUp` and `SumDown`? Explain why they do not work to prove termination.

Exercise 12.7.

Mark both of the lemmas with `{:induction false}` and fill in the proofs.

Exercise 12.8.

Without relying on automatic induction, prove

```
lemma AppendSumUp(lo: int, hi: int)
```



```
requires lo < hi  
ensures SumUp(lo, hi - 1) + F(hi - 1) == SumUp(lo, hi)
```

12.3.1. Computing sums by iteration

There are also two ways to write a loop to compute the sum. Assuming $lo \leq hi$, here is a loop implementation where the loop index increases. I will refer to this loop as LoopUp.

```
s, i := 0, lo;  
while i != hi  
  invariant lo <= i <= hi  
{  
  s := s + F(i);  
  i := i + 1;  
}
```

And here is a loop implementation where the loop index decreases. I will refer to this loop as LoopDown.

```
s, i := 0, hi;  
while i != lo  
  invariant lo <= i <= hi  
{  
  i := i - 1;  
  s := s + F(i);  
}
```

If we want to prove that these loops compute the sum into s , then, for each loop, we'll also need a loop invariant about s .

12.3.2. Verifying the sum

Let's focus on LoopUp. Using the technique of replacing a constant (hi) with a variable (i), a reasonable candidate for its loop invariant about s is

```
invariant s == SumUp(lo, i)
```

This loop invariant is established on entry to the loop, since we initialized i to lo . It lets us conclude $s == \text{SumUp}(lo, hi)$ after the loop, since i is hi there. But the verifier complains

that the invariant is not maintained by the loop body. ☹ To figure out why, let's write out some annotations, working backward from the desired loop invariant at the end of the loop body and using substitutions as we encounter assignments:

```
{ { s == SumpUp(lo, i) } }
{ { s + F(i) == SumpUp(lo, i + 1) } }
s := s + F(i);
{ { s == SumpUp(lo, i + 1) } }
i := i + 1;
{ { s == SumpUp(lo, i) } }
```

Our proof obligation is to show that the first annotation implies the second. We can now see the problem: `SumUp(lo, i)` is defined in terms `SumUp(lo + 1, i)`, but we need to conclude something about `SumUp(lo, i + 1)`. So, the summation goes the wrong way.

There are three ways to fix this problem.

One way to fix the problem is to use a lemma that tells us what happens when you increase the upper bound passed to `SumUp`. By calling

```
AppendSumUp(lo, i + 1);
```

(see Exercise 12.8) on entry to the loop body (or calling `AppendSumUp(lo, i)` after the increment of `i`), the verification goes through. Analogously, for `LoopDown`, you need to call `PrependSumDown(i, hi)` at the end of the loop body (or `PrependSumDown(i - 1, hi)` if you call it before the decrement of `i`).

A second way to fix the problem is to change the loop invariant. Instead of writing it in terms of `SumUp`, we can write it in terms of `SumDown`, since the definition of `SumDown` directly says what happens when you increase the upper bound.

```
invariant s == SumDown(lo, i)
```

What you can see is that `LoopUp` needs `SumDown` in its invariant. That is, when you iterate up, the function you

need recurses downward. Conversely, to prove LoopDown, the invariant you need is

invariant $s == \text{SumUp}(i, hi)$

This is a good thing to remember. But there's one more possible fix, and it lets you use SumUp with LoopUp and use SumDown with LoopDown.

The third way to fix the problem is to replace the what-has-been-done invariant with a what's-yet-to-be-done invariant, as I mentioned in the *yet to be done* Loop Design Technique 12.1. For LoopUp, the invariant then becomes

invariant $s + \text{SumUp}(i, hi) == \text{SumUp}(lo, hi)$

What-has-been-done invariants are far more common, so you may not have thought about this possible formulation. But when you look at it, it's quite natural. Essentially what one loop iteration does is move the term $F(i)$ from $\text{Sum}(i, hi)$ into s .

Exercise 12.9.

Prove the following lemma.

```
lemma SumRanges(lo: int, mid: int, hi: int)  
  requires lo <= mid <= hi  
  ensures SumUp(lo, mid) + SumUp(mid, hi) == SumUp(lo, hi)
```

12.3.3. Summary of programs

For side-by-side comparisons, here are the different versions of the sum method:

```
method LoopUp0(lo: int, hi:
int)
```

```
    returns (s: int)
    requires lo <= hi
    ensures s == SumUp(lo, hi)
{
    s := 0;
    var i := lo;
    while i != hi
        invariant lo <= i <= hi
        invariant s == SumUp(lo,
i)
    {
        s := s + F(i);
        i := i + 1;
        AppendSumUp(lo, i);
    }
}
```

```
method LoopUp1(lo: int, hi:
int)
```

```
    returns (s: int)
    requires lo <= hi
    ensures s == SumDown(lo, hi)
{
    s := 0;
    var i := lo;
    while i != hi
        invariant lo <= i <= hi
        invariant s == SumDown(lo,
i)
    {
        s := s + F(i);
        i := i + 1;
    }
}
```

```
method LoopUp2(lo: int, hi:
int)
```

```
    returns (s: int)
    requires lo <= hi
    ensures s == SumUp(lo, hi)
{
    s := 0;
    var i := lo;
    while i != hi
```

```
method LoopDown0(lo: int, hi:
int)
```

```
    returns (s: int)
    requires lo <= hi
    ensures s == SumDown(lo, hi)
{
    s := 0;
    var i := hi;
    while i != lo
        invariant lo <= i <= hi
        invariant s == SumDown(i,
hi)
    {
        i := i - 1;
        s := s + F(i);
        PrependSumDown(i, hi);
    }
}
```

```
method LoopDown1(lo: int, hi:
int)
```

```
    returns (s: int)
    requires lo <= hi
    ensures s == SumUp(lo, hi)
{
    s := 0;
    var i := hi;
    while i != lo
        invariant lo <= i <= hi
        invariant s == SumUp(i, hi)
    {
        i := i - 1;
        s := s + F(i);
    }
}
```

```
method LoopDown2(lo: int, hi:
int)
```

```
    returns (s: int)
    requires lo <= hi
    ensures s == SumDown(lo, hi)
{
    s := 0;
    var i := hi;
    while i != lo
```

<pre> invariant lo <= i <= hi invariant s + SumUp(i, hi) == SumUp(lo, hi) { s := s + F(i); i := i + 1; } </pre>	<pre> invariant lo <= i <= hi invariant SumDown(lo, i) + s == SumDown(lo, hi) { i := i - 1; s := s + F(i); } </pre>

One additional point you may notice in these programs is that loops where the loop index goes upward perform the increment of the loop index at the end of the loop body, after the rest of the body uses it. This is called *post-incrementing*. Loops where the loop index goes downward perform the decrement of the loop index at the beginning of the loop body, before the rest of the body uses it. This is called *pre-decrementing*. It is typical to do post-increments and pre-decrements.

12.4. Summary

We frequently write imperative programs whose specifications are given by functional definitions. When we do, we need to make sure the direction of recursion matches up with the way we do iterations. Any discrepancies can be solved by proving appropriate lemmas, but it may also be possible to phrase loop invariants in a way that fits both the recursive definitions and the loop implementations.

While comparing loops that go in different directions, we saw the use of post-increments of the loop index in upward loops and pre-decrements of the loop index in downward loops.

Among the programs we wrote, I also presented two more loop design techniques. These techniques are essential for mastery of program proofs.

Notes

In this chapter, I showed both what-has-been-done invariants (which are most common) and what's-yet-to-be-done invariants. To more directly facilitate reasoning in terms of what has yet to be done, one can use invocable block statements with specifications. As Eric Hehner shows, such *specified blocks* sometimes provide a natural alternative to **while** statements [\[62\]](#).

Chapter 13

Arrays and Searching



An *array* is a simple mutable data structure. Its elements are accessed by numeric values, called *indices*, that are computed at run time. In this chapter, we look at how to write and verify programs that use arrays. This ushers us into a wealth of useful, interesting, and sometimes subtle algorithms.

13.0. About Arrays

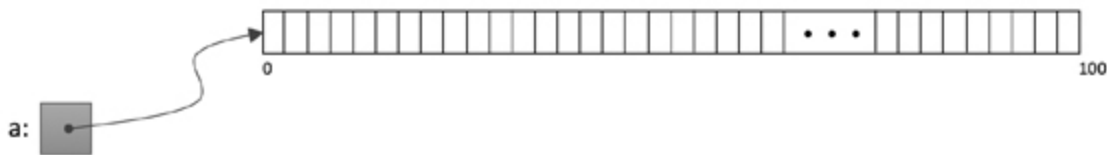
I assume you have seen and used arrays before. But just to make sure we're talking about the same kinds of arrays, let's go over the essentials.

13.0.0. Array allocation and length

An array stores a fixed number of elements, addressable by consecutive indices starting from 0. The *length* of an array is determined when the array is allocated. The following program statement allocates an array of length 100 and sets local variable `a` to reference this array.

```
var a := new int[100];
```

Here's a diagram that illustrates variable `a`, whose value is a reference to the allocated array:



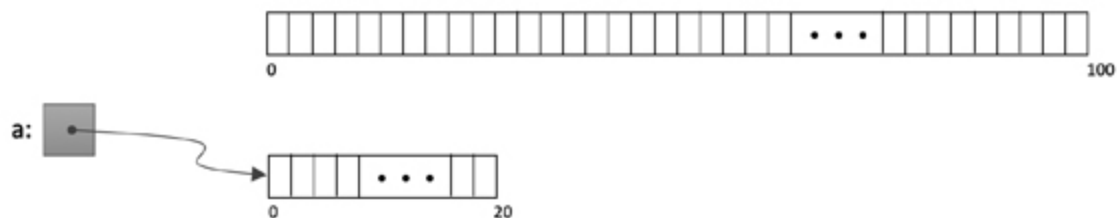
We can obtain the length of `a` by the array's `Length` member. For example, after the assignment above, we can prove

```
assert a.Length == 100;
```

The type of each of the 100 elements is `int`. The type of `a` is `array<int>`, which is independent of the length of `a`. This means we can update `a` to reference another `int`-array, even if that array has a different length. For example,

```
a := new int[20];
```

allocates an array of length 20 and sets `a` (from above) to reference it. The new situation is illustrated as follows:



13.0.1. Array elements

The element at index i of an array a is accessed by the expression $a[i]$. It is a proof obligation that i be a proper index of a , that is, we have to prove

```
0 <= i < a.Length
```

Such an array access can be used as the left-hand side of an assignment. For example,

```
a[9] := a[9] + 5;
```

increments element 9 of a by 5.

The update of one array element has no effect on the other elements of the array. For example, the assertion in the following program is provable:

```
a[6] := 2;
a[7] := 3;
assert a[6] == 2 && a[7] == 3;
```

Of course, if two integer expressions evaluate to the same value, then using those expressions when indexing an array will access the same array element. For example,

```
method TestArrayElements(j: nat, k: nat)
  requires j < 10 && k < 10
{
  var a := new int[10];
  a[j] := 60;
  a[k] := 65;
  if j == k {
    assert a[j] == 65;
  } else {
    assert a[j] == 60;
  }
}
```

This method illustrates that the second assignment overwrites the first when $j == k$, and that it has no effect on the first if $j != k$.

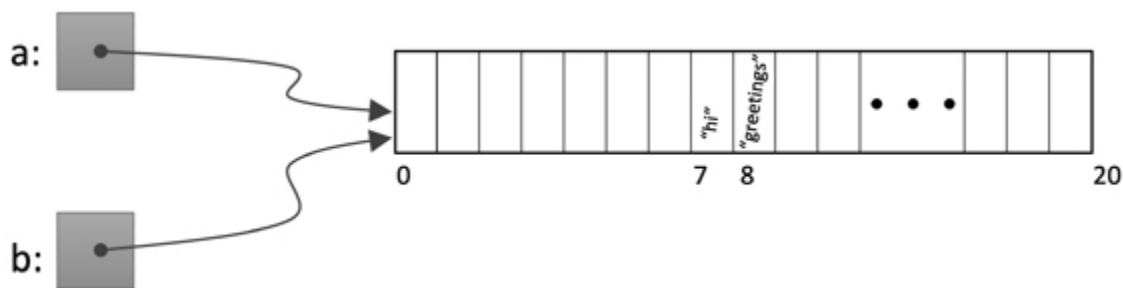
13.0.2. Arrays are references

An important aspect of arrays in Dafny is that they are *references* to their elements. Consequently, when you copy

an array, you're copying the array reference, not the array elements. This program fragment illustrates:

```
var a := new string[20];  
a[7] := "hello";  
var b := a;  
assert b[7] == "hello";  
b[7] := "hi";  
a[8] := "greetings";  
assert a[7] == "hi" && b[8] == "greetings";
```

The following diagram shows the situation after the program fragment:



There is only one array in this example. It is allocated to have length 20 and its element type is `string`. Local variable `a` is set to reference the array, and element 7 is set to `"hello"`. The assignment to `b` makes `b` reference the same array as `a` does. Thus, for the rest of the example, since `a` and `b` are equal (that is, they are the same array reference), it doesn't matter which local variable is used to access the array.

Continuing this example, note that `a` and `b` are different local variables. Therefore, if `b` is set to reference a different array, then `a` still references the previous array.

```
b := new string[8];  
b[7] := "long time, no see";  
assert a[7] == "hi";  
assert a.Length == 20 && b.Length == 8;
```

13.0.3. Multi-dimensional arrays

An array in Dafny can have more than one dimension. Here's an example that allocates a 2-dimensional array (usually called a *matrix*):

```
var m := new bool[3, 10];  
m[0, 9] := true;  
m[1, 8] := false;  
assert m.Length0 == 3 && m.Length1 == 10;
```

The element type of this matrix is **bool**, and the type of `m` is **array2<bool>**. Each dimension has a length, and these lengths are obtained using the members `Length0` and `Length1`.

I will mostly use 1-dimensional arrays.

13.0.4. Sequences

A cousin of arrays are *sequences*. Whereas arrays are mutable and are accessed via references, sequences are immutable values, just like booleans, integers, and datatypes are. For this reason, array types are often called *reference types* and sequence types *value types*.

The elements of a sequence must have a common type, just as for arrays. The sequence type constructor is written **seq**, so **seq<int>** denotes integer sequences and **seq<bool>** denotes sequences of booleans.

Since an array is a reference to elements stored in the heap, an array is created using **new**, as we saw above. The simplest way to write down a sequence value is to list its elements between square brackets. For example, `[]` denotes the empty sequence, `[58]` denotes the singleton sequence with the integer element 58, and

```
["hey", "hola", "tjena"]
```

is a sequence of three strings.

Indexing into a sequence has the same syntax as for arrays. For example, if `greetings` denotes the sequence of strings above, then `greetings[2]` is `"tjena"`. The length of an

array `a` is written `a.Length`, but the length of a sequence `a` is written `|a|`. For example, `|[]|` is 0 and `|greetings|` is 3.

Sequences can be concatenated using `+`. Here is an example with some pentagonal numbers:

```
[1, 5, 12] + [22, 35] == [1, 5, 12, 22, 35]
```

Another way to obtain a sequence is as a subsequence of another sequence. The expression `a[lo..hi]` denotes the subsequence of `a` of length `hi - lo` obtained by taking the first `hi` elements of `a` and then dropping the first `lo` elements thereof. The operation requires

```
0 <= lo <= hi <= |a|
```

If the lower bound is 0, it can be omitted, and if the upper bound is the length of the sequence, it can be omitted. Here are some examples, where I'm using `p` to denote the sequence of five pentagonal numbers above:

```
p[2..4] == [12, 22]
p[..2] == [1, 5]
p[2..] == [12, 22, 35]
greetings[..1] == ["hey"]
greetings[1..2] == ["hola"]
```

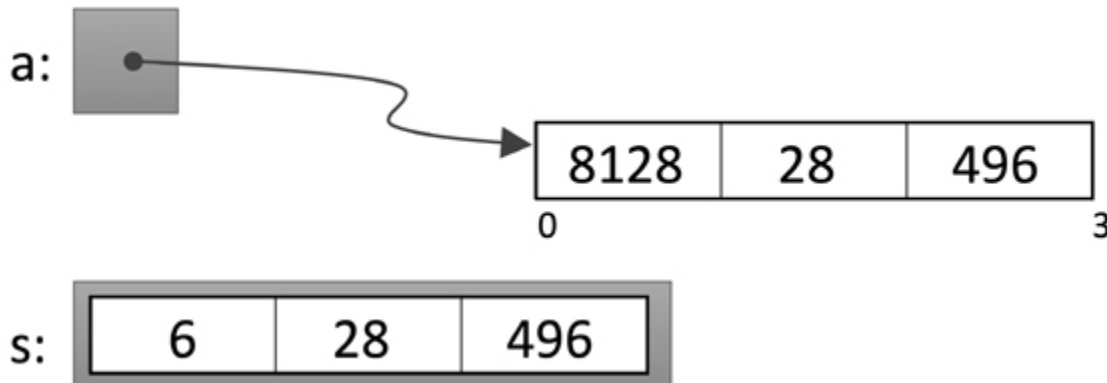
An operation like `p[2..]` applied to `p` is pronounced “drop 2” and the operation `p[..2]` is pronounced “take 2”.

It is also possible to obtain a sequence from a stretch of array elements. This uses the same syntax as the take and drop operations for subsequences. To obtain a sequence of all the elements of an array `a`, you can drop both the lower and upper bounds, writing `a[..]`. Note that the sequence value is created from the current array elements and is not sensitive to future changes of the array. For example, the assertions in the following code snippet all hold:

```
var a := new int[3];
a[0], a[1], a[2] := 6, 28, 496;
assert a[..2] == [6, 28] && a[1..] == [28, 496];
var s := a[..];
assert s == [6, 28, 496];
a[0] := 8128;
```

```
assert s[0] == 6 && a[0] == 8128;
```

The following diagram shows the final state of the code snippet. Note that `a` contains a reference to an array, whereas `s` contains a sequence value.



Enough background on arrays and sequences. It's time for some programs that use them.

13.1. Linear Search

Let's consider a method for finding an element in an array. To make the method usable for various search applications, we'll parameterize it with a predicate `P` and write the method to look for an array element that satisfies `P`. The type signature of the method is thus

```
method LinearSearch<T>(a: array<T>, P: T -> bool) returns (n: int)
```

The method takes a type parameter `T`, which denotes the element type of the array. We can tell from the type of `P` that it takes an argument of type `T` and returns a boolean. The method has an out-parameter `n`, which we'll use to return the findings of our search.

I haven't talked about using functions as values before. I won't say much here, either, other than to point out that if you declare a function, then you can use the name of that

function as a first-class value. For example, suppose you declare

```
predicate NearPi(r: real) {  
    3.14 <= r <= 3.15  
}
```

and `rls` is a variable of type `array<real>`, then you can call the method above as follows:

```
var n := LinearSearch(rls, NearPi);
```

Note that, since parameter `P` is non-ghost, the value passed in at the call site of `LinearSearch` must also be non-ghost, so `NearPi` is declared to be a compiled predicate.

The method implementations we'll consider for `LinearSearch` go through the array elements one by one. The name for such an algorithm is Linear Search.

I will go through several variations of the specification and corresponding program, which will let me introduce a common construct for writing specifications of array programs: logical quantifiers.

13.1.0. A simple specification

As a warm-up, let's write a simple specification that returns `n` in the inclusive range $0 \leq n \leq a.Length$, where a value strictly less than `a.Length` indicates an index where the array element satisfies `P`.

```
method LinearSearch0<T>(a: array<T>, P: T -> bool) returns (n: int)  
    ensures 0 <= n <= a.Length  
    ensures n == a.Length || P(a[n])
```

For our implementation, we will use the out-parameter `n` as our loop index. Here is the loop specification:

```
{  
    n := 0;  
    while n != a.Length  
        invariant 0 <= n <= a.Length  
}
```

This loop specification is enough to establish the method postcondition, because we can conclude $n == a.Length$ after the loop, which the method specification allows.

The loop body is going to increment n , and before doing so, it will check if P holds of the current array element. But suppose it does, what do we do then? Well, our mission is then done, so we'd rather not bother with loop invariants and other obligations. To put it bluntly, we're ready to head straight to the exit. We can do that with a **return** statement, which abruptly terminates the method body. At the time of a **return**, you immediately have to prove that the method postcondition holds. Alternatively, and with the same consequences, you can think of the **return** statement as jumping to the very end of the method body, at which point you need to prove that the method post-condition holds.

So, here is the full method body of `LinearSearch0`:

```
n := 0;
while n != a.Length
  invariant 0 <= n <= a.Length
{
  if P(a[n]) {
    return;
  }
  n := n + 1;
}
```

Instead of using a **return** to jump to the end of the method body, you can use a **break** statement, which abruptly terminates the loop, continuing the execution of the method body immediately after the loop. This is common in many programs. You do not need to prove that the loop invariant holds at the time you do a **break**, and you're allowed to do a **break** regardless of if the loop guard holds at that moment. I've told you that you can rely on the invariant and the negation of the guard to hold immediately after the loop, but this is not true if the loop uses a **break**. For every **break** that a loop body uses, there's an additional set of paths through the loop body to its exit. Other than

these differences, a **break** poses no problems, and the verifier understands all about the additional loop exits. Nevertheless, to keep things a bit more consistent, I will avoid the use of **break** in this book.

13.1.1. Having to justify defeat

The specification of method `LinearSearch0` admits an implementation that is even easier than using a loop: just change the entire method body into `n := a.Length`. (If there was an emoji for the Winnie the Pooh trombone fanfare of disappointment, I would insert it here.) We don't want our specification to allow an implementation to give up so easily. If the method returns `a.Length`, then we'd like to say that there is no element in `a` that satisfies `P`. Doing so requires a logical quantifier. I'll use a *universal quantifier*, written **forall** in Dafny and $\forall$ in logic. It introduces a bound variable and says that a condition holds universally for—that is, for all values of—that bound variable.

Here is an example quantifier that speaks of all integers `x`, where `Fib` is the Fibonacci function (Section 3.1):

```
forall x: int :: 5 <= x ==> 5 <= Fib(x)
```

This expression has the value **true** if the *body* of the quantifier—that is, the expression to the right of the “`::`”—holds for every integer `x`, and the value **false** otherwise.

Exercise 13.0.

What is the value, **false** or **true**, of the example quantifier above?

As for local variables, we can leave off the type “`: int`” if the type can be inferred.

Here is our improved specification for the linear-search method:

```
method LinearSearch1<T>(a: array<T>, P: T -> bool) returns (n: int)
```



```

ensures 0 <= n <= a.Length
ensures n == a.Length || P(a[n])
ensures n == a.Length ==>
  forall i :: 0 <= i < a.Length ==> !P(a[i])

```

This says that, in the event that `n` is returned as `a.Length`, then `!P(a[i])` holds for every `i` in the range `0 <= i < a.Length`. This looks pretty good, because it prevents the implementation from returning `a.Length` when there is an element satisfying `P`.

With the improved method specification, what will our loop invariant look like? Let's apply the *replace constant by variable* Loop Design Technique 12.0. After the loop, we want the condition

```

forall i :: 0 <= i < a.Length ==> !P(a[i])

```

We already have a variable `n` that will equal the constant `a.Length` after the loop, so by replacing that constant by the variable `n`, we arrive at the following loop invariant:

```

invariant forall i :: 0 <= i < n ==> !P(a[i])

```

Our loop implementation will be exactly the same as in `LinearSearch0`, but let's take a detailed look at why it works. Writing annotations around the loop body's increment of `n`, we get:

```

{{ forall i :: 0 <= i < n + 1 ==> !P(a[i]) }}
n := n + 1
{{ forall i :: 0 <= i < n ==> !P(a[i]) }}

```

Next, we'll use three equality transformations that are common when we work with quantifiers. Since `n` is of type `nat`, one is the arithmetic equality

```

0 <= i < n + 1
=
0 <= i < n || i == n

```

The second equality is a quantifier *Range Split* (cf. Section B.7), which has the general form

```

forall x :: A || B ==> C
=

```

(forall x :: A ==> C) && (forall x :: B ==> C)

We apply Range Split with x and A and B and C as i and $0 \leq i < n$ and $i == n$ and $!P(a[i])$, respectively, which gives us:

```
forall i :: 0 <= i < n || i == n ==> !P(a[i])
=
(forall i :: 0 <= i < n ==> !P(a[i])) &&
(forall i :: i == n ==> !P(a[i]))
```

The third equality is known as the *One-Point Rule* (cf. Section B.7) and gives us a way to eliminate a quantifier. Its general form is

```
forall x :: x == E ==> A
=
A[x := E]
```

where E is an expression that does not mention x . The last line uses a substitution (which I first introduced in Section 2.3, but see also Section B.6) that replaces every occurrence of x in A by E . In our situation, we apply the One-Point Rule with x and E and A as i and n and $!P(a[i])$, respectively:

```
forall i :: i == n ==> !P(a[i])
=
!P(a[n])
```

Going back to the annotations around $n := n + 1$, the three equalities above allow us to write the condition before the increment of n as follows:

```
{ { (forall i :: 0 <= i < n ==> !P(a[i])) && !P(a[n]) } }
{ { forall i :: 0 <= i < n + 1 ==> !P(a[i]) } }
n := n + 1
{ { forall i :: 0 <= i < n ==> !P(a[i]) } }
```

This is good news. The first of the conjuncts holds on account of the loop invariant on entry to the loop body, and the second conjunct holds after the **if** statement (because the then branch uses a **return** to jump to the end of the method if $P(a[n])$ does hold).

That proves the correctness of `LinearSearch1`.

13.1.2. Finding the first element

We can easily be more precise in our method specification even in the case where an element is found: we can promise to return the first occurrence of an element satisfying P . To that end, we'll say that there's no element satisfying P among the first n elements of the array. The postcondition we added in `LinearSearch1` was a weaker form of this, so we replace that **ensures** clause with a new one:

```
method LinearSearch2<T>(a: array<T>, P: T -> bool) returns (n:
int)
  ensures 0 <= n <= a.Length
  ensures n == a.Length || P(a[n])
  ensures forall i :: 0 <= i < n ==> !P(a[i])
```

The loop specification and body from `LinearSearch1` also work for our new method.

13.1.3. Knowing it's there

Let's do one more variant of Linear Search, one where we're given that the search item exists. This means we can simplify the postcondition to remove the special case $n == a.Length$. To specify this new assumption, we use a precondition. The most straight-forward way to write the condition is to use an *existential quantifier*, written **exists** in Dafny and $\exists$ in logic.

Here is an example that existentially quantifies over the integers:

```
exists x :: 0 <= x && Fib(x) == 143
```

This expression has the value **true** if there is some integer for which the body holds, and the value **false** otherwise.

Exercise 13.1.

What is the value, **false** or **true**, of the example quantifier above?

With an existential quantifier in its precondition, our new linear-search specification looks like this:

```
method LinearSearch3<T>(a: array<T>, P: T -> bool) returns (n: int)  
  requires exists i :: 0 <= i < a.Length && P(a[i])  
  ensures 0 <= n < a.Length && P(a[n])
```

The precondition says there exists an index i such $P(a[i])$ holds, and the postcondition says that n is an index such that $P(a[n])$ holds.

An important difference in how we formulate universal and existential quantifiers is that the main connective of the body of universal quantifiers is an implication, whereas it is a conjunction for existential quantifiers. For example, to say that $P(a[i])$ holds for *all* indices, we write

```
forall i :: 0 <= i < a.Length ==> P(a[i])
```

whereas to say that it holds for *some* index, we write

```
exists i :: 0 <= i < a.Length && P(a[i])
```

If we changed the $\Rightarrow$ to an $\&\&$, then the **forall** expression would say that all integers are in the range from 0 to `a.Length`, which isn't true (for example, -1 isn't). Thus, the **forall** expression would have the value **false**. Conversely, if we changed the $\&\&$ to an $\Rightarrow$, then the **exists** expression would say there's an integer i such that *if* the integer is in the range from 0 to `a.Length`, *then* yada yada. This is trivially true for any integer not in that range, of which there are plenty (for example, -1 is one). Thus, the **exists** expression would have the value **true**.

You may be surprised that the same method implementation and loop invariant as for `LinearSearch1` and `LinearSearch2` also work for `LinearSearch3`. Why is that? Well, it's no surprise that the postcondition is met at the **return** statement. What may be surprising is what happens after the loop, where the negation of the guard tells us that n is `a.Length`, which our new specification does not allow us to

return. After the loop, we also know the invariant, so we can prove the following condition to hold:

```
forall i :: 0 <= i < a.Length ==> !P(a[i])
```

Here's another useful property, known as *De Morgan's Law for Quantifiers* (see also Section B.8):

```
(forall x :: !R) == !(exists x :: R)
```

From it and other properties of Boolean logic, we get:

```
forall x :: A ==> B
=
forall x :: !A || B
= { De Morgan's Law }
forall x :: !(A && !B)
= { De Morgan's Law for Quantifiers }
!exists x :: A && !B
```

Using x and A and B as i and $0 \leq i < a.Length$ and $P(a[i])$, we thus have

```
forall i :: 0 <= i < a.Length ==> !P(a[i])
=
!exists i :: 0 <= i < a.Length && P(a[i])
```

where the first line is what we get from the invariant and negation of the guard, and the second expression is the negation of the precondition. In other words, after the loop, we are able to derive a condition that is the negation of the precondition. This can only mean one thing—control never reaches this point!

13.1.4. An invariant that says where to look

There is a different and interesting way to write the loop invariant for `LinearSearch3`. At the top of every iteration, we can say that the element we're looking for is at some index n or higher.

```
exists i :: n <= i < a.Length && P(a[i])
```

Working backward from this invariant across the update of n , we get the following annotations:

```

{{ !P(a[n]) && exists i :: n <= i < a.Length && P(a[i]) }}
// Range Split and One-Point Rule
{{ !P(a[n]) && (P(a[n]) || exists i :: n + 1 <= i < a.Length &&
P(a[i])) }}
// logic
{{ exists i :: n + 1 <= i < a.Length && P(a[i]) }}
n := n + 1
{{ exists i :: n <= i < a.Length && P(a[i]) }}

```

The first conjunct holds on account of that the then branch does a **return**, and the second conjunct holds on behalf of the invariant on entry to the loop. So, the loop from above maintains this invariant.

We can perform two more simplifications. First, the existential quantifier implies $n < a.Length$, since it says that i lies in between. This means we can strengthen the invariant that mentions the range of n :

```
0 <= n < a.Length
```

The loop guard is always evaluated in a state where the loop invariant holds. Under this range of n , the condition $n \neq a.Length$ can be simplified to **true**. However, if we do, then Dafny's foolishly simple scheme for a default **decreases** clause of the loop (see Section 11.2.1) doesn't come up with anything, so we'll need to supply one ourselves.

Here is our final implementation of `LinearSearch3`:

```

n := 0;
while true
  invariant 0 <= n < a.Length
  invariant exists i :: n <= i < a.Length && P(a[i])
  decreases a.Length - n
  {
    if P(a[n]) {
      return;
    }
    n := n + 1;
  }

```

The pattern in this loop invariant is a variation of replacing a constant by a variable, but the replacement is in the assumption, not in the proof goal:

Loop design technique 13.0. (Replace constant by variable in precondition)

To establish a postcondition from a precondition of the form $\dots c \dots$ for some constant c , use a loop invariant $\dots k \dots$ where k is a local variable updated by the loop to gradually make the assumption more specific. We may call this *replace constant by a variable in the precondition*.

13.1.5. Summary of quantifier properties

In this subsection on Linear Search, we came across several useful properties of quantifiers. Here is a summary of those properties in various forms:

Range Split:

- $(\text{forall } x :: A \mid\mid B \implies C) \iff (\text{forall } x :: A \implies C) \ \&\& \ (\text{forall } x :: B \implies C)$
- $(\text{exists } x :: (A \mid\mid B) \ \&\& \ C) \iff (\text{exists } x :: A \ \&\& \ C) \mid\mid (\text{exists } x :: B \ \&\& \ C)$

One-Point Rule:

- $(\text{forall } x :: x == E \implies A) \iff A[x := E]$
- $(\text{exists } x :: x == E \ \&\& \ A) \iff A[x := E]$

De Morgan's Law for Quantifiers:

- $(\text{forall } x :: A \implies B) \iff \neg(\text{exists } x :: A \ \&\& \ \neg B)$
- $\neg(\text{forall } x :: A \implies \neg B) \iff (\text{exists } x :: A \ \&\& \ B)$

Appendix B contains additional properties of quantifiers.

Exercise 13.2.

Write a linear-search specification for a method that always returns a value strictly less than `a.Length` and uses a negative value (instead of `a.Length`) to signal that no element satisfies P .

13.2. Binary Search

In this section, we'll write another staple from the computer-science pantry, Binary Search. The idea is that if the input is sorted, then we can halve the search space with each iteration of the loop. I'll use an integer array and we'll search for a given key. For this program, we'll get more practice with the *replace constant by variable* Loop Design Technique 12.0.

13.2.0. Specifying sortedness

We start with the question of how to specify that the input is sorted. Here is the answer:

```
forall i, j :: 0 <= i < j < a.Length ==> a[i] <= a[j]
```

It says that if you take two indices into the array, *i* and *j* where *i* is “to the left” of *j*, then the elements at those two indices are ordered properly. (It would also be fine to have written *i* <= *j* instead of *i* < *j*.)

What I wrote is not the only way to express that the array is sorted. One alternative way would be to nest the quantifications over *i* and *j*. For example, like

```
forall i :: 0 <= i ==>  
  forall j :: i < j < a.Length ==> a[i] <= a[j]
```

This works, but unnested quantifiers tend to be easier to work with.

Another way to specify sortedness is to say that adjacent elements are ordered properly:

```
forall i :: 0 <= i < a.Length - 1 ==> a[i] <= a[i + 1]
```

This expression is logically equivalent to the ones above, but it doesn't immediately make it clear that non-adjacent elements are also ordered properly. This is a property we'll need in Binary Search. To get that “transitive” property about non-adjacent elements, we would need to do an inductive proof, which is just extra effort. So, don't express sortedness in this adjacent-elements form.

Exercise 13.3.

Write a quantified expression that says the elements in a given integer array are strictly increasing.

Exercise 13.4.

Prove the following lemma, which shows that you can get the transitive property from the adjacent-elements property.

```
lemma SortedTransitive(a: array<int>, i: int, j: int)  
  requires forall k :: 0 <= k < a.Length - 1 ==> a[k] <= a[k+1]  
  requires 0 <= i <= j < a.Length  
  ensures a[i] <= a[j]
```

13.2.1. Binary Search postcondition

We could write a postcondition for Binary Search that is like the one for Linear Search. For example, we could take the postcondition of `LinearSearch1` in Section 13.1.1. But it's easy for the method to provide some additional information back to the caller. We'll return the “earliest insertion point”, which means the index of the leftmost occurrence of the key, if it exists in the array, or the index where to insert the key, if it's not in the array. More precisely, we'll return the length of the longest prefix in which everything is less than the key we're looking for. Stated differently, we'll return a number n such that the first n elements of the array are strictly smaller than the key, and the rest of the elements are at least as large as the key.

So, here's the specification of the Binary Search we're going to write:

```
method BinarySearch(a: array<int>, key: int) returns (n: int)  
  requires forall i, j :: 0 <= i < j < a.Length ==> a[i] <= a[j]  
  ensures 0 <= n <= a.Length  
  ensures forall i :: 0 <= i < n ==> a[i] < key  
  ensures forall i :: n <= i < a.Length ==> key <= a[i]
```

Before we launch into writing the implementation of this method, we can try out the “earliest insertion point” specification by writing a client method. This gives us some assurance that the specification we wrote does what we want and expect. Here is a method that determines whether or not a given key is indeed in the sorted array:

```
method Contains(a: array<int>, key: int) returns (present:
bool)
  requires forall i, j :: 0 <= i < j < a.Length ==> a[i] <=
a[j]
  ensures present == exists i :: 0 <= i < a.Length && key ==
a[i]
{
  var n := BinarySearch(a, key);
  present := n < a.Length && a[n] == key;
}
```

13.2.2. Implementation

The main invariant of Binary Search is that we have a subsection of the array, a “window”, from `lo` to `hi`, where we're still looking for the key. Anything to the left of the window is too small to be the key; I'll call this the “too-small section” of the array. Anything to the right of the window is at least as large as the key; I'll call this the “too-big section” of the array. We'll keep shrinking the window until it has size 0, at which time the too-small section and too-big section meet. We obtain a logarithmic number of iterations by halving the size of the window each time.

Here is our loop specification along with the initial and final assignments. Note that `lo` and `hi` take the place of `n` from the postcondition; that is, we're applying the *replace constant by variable* Loop Design Technique 12.0 twice. After the loop, `lo` and `hi` are equal, so the rest of the invariant and the assignment to `n` establish the method's postcondition.

```
{
  var lo, hi := 0, a.Length;
```

```

while lo < hi
  invariant 0 <= lo <= hi <= a.Length
  invariant forall i :: 0 <= i < lo ==> a[i] < key
  invariant forall i :: hi <= i < a.Length ==> key <= a[i]
  n := lo;
}

```

To implement the loop, we take the midpoint between `lo` and `hi` and compare the element at that midpoint with the key:

```

var mid := (lo + hi) / 2;
if a[mid] < key {
  ?
} else {
  ?
}

```

We can use annotations, as we've done before, to give us a precise guide to what the assignments at the two question marks should be. But let's see if we can work it out from our understanding of invariants. Here is a diagram to make the formulas more visual:



In the then branch of the `if`, we're looking at an element (`a[mid]`) that's smaller than the key. Since the array is sorted, all elements to the left of `a[mid]` are then also less than the key. This suggests that we should grow the too-small section, enough to include the element `a[mid]`. This is done by the assignment

```
lo := mid + 1;
```

In the else branch of the `if`, we've found `a[mid]` to be at least as large as the key, which means the elements to its right are, too. This suggests that we should grow the too-

big section, enough to include the element `a[mid]`. This is done by the assignment

```
hi := mid;
```

Note the asymmetry in the two assignments. This stems from the fact that `lo` is a limit of the too-small section; that is, an index `i` is in the too-small section if `i < lo`.

We're using the default **decreases** clause, `hi - lo`, which is the window size. (Remember, if you want to find out what **decreases** clause is used by default for a particular loop, hover your mouse over the **while** keyword in the Dafny IDE and a tool tip will give you this information.) An iteration moves `a[mid]` from the window into either the too-small section or the too-big section, which makes the window size (that is, `hi - lo`) decrease.

Er, not so fast! Is `a[mid]` really in the window? That is, is it really the case that `lo <= mid < hi`? The diagram above certainly makes it seem that way, but it's hard to capture every aspect of a problem into a diagram. We're computing `mid` to be the integer average of `lo` and `hi`. For any integers `lo` and `hi` such that `lo < hi`, the following proof calculation in Dafny confirms that `lo <= mid < hi`:

```
calc {
  lo;
==
  (lo + hi) / 2;
<= { assert lo <= hi; }
  (lo + hi) / 2; // this is mid
< { assert lo < hi; }
  (lo + hi) / 2;
==
  hi;
}
```

So, we're good, after all!

Exercise 13.5.

Change the assignment `lo := mid + 1`; to `lo := mid`;. What error does the verifier report? Construct an input to

BinarySearch that (if you would ignore the verification error and nevertheless compile and run the program) would cause the error that the verifier is complaining about.

Exercise 13.6.

Write a specification and implementation for a Binary Search that returns the “latest insertion point”, that is, an array position just *after* all occurrences of the key.

Exercise 13.7.

Specify and write a version of Binary Search that returns as soon as it encounters any occurrence of the key. Return -1 if the key is not in the array.

Exercise 13.8.

Akin to what we did in Sections 13.1.3 and 13.1.4, add a precondition to BinarySearch that says key is in the given array. Then, simplify the rest of the specification and method body. It's okay to return the index of any array element that contains the key.

13.3. Minimum

Let's develop a method that computes the minimum of an array. We'll build up to it, piece by piece, which illustrates some benefits of specification and verification during program design and development. The example also shows two more loop design techniques.

13.3.0. There's nothing smaller

Our method takes as input an integer array a and returns an integer m that is as small as any element in a . This postcondition is easily expressed with a universal quantifier.

```
method Min(a: array<int>) returns (m: int)
  ensures forall i :: 0 <= i < a.Length ==> m <= a[i]
```

We'll replace the constant `a.Length` by a loop index `n` (Loop Design Technique 12.0), which gives us a loop specification:

```
{
  var n := 0;
  while n != a.Length
    invariant 0 <= n <= a.Length
    invariant forall i :: 0 <= i < n ==> m <= a[i]
}
```

The loop body will increment `n`. Before doing so, if we find that `a[n]` is smaller than `m`, then we update `m` accordingly.

```
{
  if a[n] < m {
    m := a[n];
  }
  n := n + 1;
}
```

We did that pretty quick, huh?

13.3.1. There is something that small

The specification we wrote is only part of the story. Already in Section 1.4.1, we argued that a `Min` method should return one of its inputs. Now that we have a whole array of input integers, we'll use an existential quantifier to express this additional postcondition:

```
ensures exists i :: 0 <= i < a.Length && m == a[i]
```

To incorporate this method goal into the loop specification, we could apply the technique of replacing a constant by a variable (`a.Length` by `n`, again), but here we can do something even simpler:

|| **Loop design technique 13.1.** (Always)
 || To establish a postcondition `Q`, make `Q` a loop invariant.

Clearly, if this were the only postcondition we're interested in, it would be senseless to iterate the loop at all. This technique is useful when there are other postconditions as well, which is the case for our `Min` method.

We'll use the new postcondition itself as an invariant:

```
invariant exists i :: 0 <= i < a.Length && m == a[i]
```

It says that `m` is, at all times, one of the elements of `a`. This condition is maintained by the loop body, but the verifier complains that it doesn't hold on entry to the loop. If you hadn't noticed before, you will now notice that our program never initializes `m`. Evidently, in Section 13.3.0, the initial value of `m` was immaterial.

Exercise 13.9.

Explain why the initial value of `m` did not matter in Section 13.3.0. On entry to a method, the value of an out-parameter is arbitrary. Describe the execution of the method, given `a` as the 3-element array 2, 10, 1, if `m` happens to start with the initial value -62081.

Okay, so we have to initialize `m`, and we have to make sure it's to an element of `a`. Like a kid in a candy store or a guitarist in a music store, we're overcome with a wealth of enticing ideas. We could pick the first element of the array:

```
m := a[0];
```

or we could pick the last element:

```
m := a[a.Length - 1];
```

or why not pick an element two-thirds into `a`:

```
m := a[2 * a.Length / 3];
```

Any of these would do, but only if the array contains any elements at all! This is a precondition we neglected to include. Let's add it

```
requires a.Length != 0
```

and pick any of the initializations of m shown above.

Exercise 13.10.

We can initialize m to an element two-thirds into a . What about one element after that?

```
 $m := a[2 * a.Length / 3 + 1];$ 
```

If you use the conventional choice of $m := a[0]$, then there's no reason to iterate the loop for $n == 0$. You can skip that loop iteration by simply initializing n to 1.

Exercise 13.11.

If you start n from 1 as I just suggested, which of the three occurrences of the number 0 in the loop invariants can you then also change to 1? Explain.

Exercise 13.12.

Specify and write a method to compute the maximum of an array of type `array<nat>`. Allow the array to be empty, and in that case return 0 as the maximum.

13.3.2. Summary of the development journey

We started this example with one postcondition and wrote the code. Then, we added a second postcondition and adjusted the loop invariant. To establish that loop invariant on entry to the loop, we realized we had to add a method precondition.

This sort of process is typical when writing code. We may not know from the start of the development process what the specification needs to be. As we go along, we realize that we need to change or add various postconditions and preconditions. Any change to the specification affects clients. This is a good reason to have specifications be explicit—and having a verifier check clients automatically is nice, too. 😊

13.4. Coincidence Count

The specification and proof of this next program uses multisets. As we have seen before (in the two advice paragraphs in Section 10.2.4, which briefly mentioned the name *extensionality*), working with multisets is not fully automatic. This means we often have to debug proofs ourselves, giving the program verifier more and more detail until it's able to confirm our proof.

The program is a fun one: given two sorted arrays, compute how many elements they have in common. In the face of duplicate elements, what I said is a bit ambiguous, so let me be more precise. How many elements of one array can you uniquely pair with an equal element of the other array? Stated differently, if you collect the elements of each array into a multiset, what is the size of the intersection of those two multisets? Here is that specification in Dafny:

```
method CoincidenceCount(a: array<int>, b: array<int>) returns
(c: nat)
  requires forall i, j :: 0 <= i < j < a.Length ==> a[i] <=
a[j]
  requires forall i, j :: 0 <= i < j < b.Length ==> b[i] <=
b[j]
  ensures c == |multiset(a[..]) * multiset(b[..])|
```

The two preconditions say that a and b are sorted (see Section 13.2.0). The postcondition uses some new notation, so let me explain it.

13.4.0. Notation

We'd like to view the two arrays as multisets. Dafny has no built-in conversion from arrays to multisets, so we first turn the array's elements into a sequence (see Section 13.0.4) and then turn the sequence into a multiset.

Function **multiset** turns a sequence into a multiset, * denotes multiset intersection, and the vertical-bar brackets

give you the size (aka cardinality) of a multiset. Note the round parentheses after function `multiset`, which is what you'd expect. If you put curly braces after the keyword `multiset`, you're forming a multiset from some given elements. For example, `multiset{}` is the empty multiset and `multiset{x}` is the singleton multiset that contains `x`.

You should now be able to read the notation in the postcondition of method `CoincidenceCount`. It says that `c` is the number of elements in the intersection of the multisets of elements of `a` and `b`.

13.4.1. Loop specification

In the implementation of `CoincidenceCount`, we'll use two loop-index variables, `m` and `n`, to keep track of how many elements of `a` and `b`, respectively, we have processed. The corresponding loop invariant is

invariant `0 <= m <= a.Length && 0 <= n <= b.Length`

The idea is that the loop will advance `m` or `n` or both, depending on how the next elements in `a` and `b` compare. It will be easiest to use a what's-yet-to-be-done invariant (but I encourage you to try a what-has-been-done invariant, too, for comparison):

invariant `c + |multiset(a[m..]) * multiset(b[n..])|
== |multiset(a[..]) * multiset(b[..])|`

Termination will follow from the increments of `m` and `n`, but since a loop iteration may increment only one of them, we have to include both `a.Length - m` and `b.Length - n` in our **decreases** clause. I'll combine them using `+` (but you can equally well combine them into a lexicographic pair):

decreases `a.Length - m + b.Length - n`

Finally, we need a loop guard. Should we loop while *both* `m < a.Length` and `n < b.Length`, or should we continue looping while *at least one* of these hold? Once we have processed

all the elements of one array, we have accounted for all coincidences, so we might as well stop then.

All in all, this gives us the following loop specification:

```
{
  c := 0;
  var m, n := 0, 0;
  while m < a.Length && n < b.Length
    invariant 0 <= m <= a.Length && 0 <= n <= b.Length
    invariant c + |multiset(a[m..]) * multiset(b[n..])|
              == |multiset(a[..]) * multiset(b[..])|
    decreases a.Length - m + b.Length - n
}
```

The verifier is able to prove the method postcondition from this loop specification, so all we have left to do is implement the loop.

13.4.2. Loop body

By guessing that the assignments $m := m + 1$ and $n := n + 1$ and $c := c + 1$ will play a role in the loop body, we can write some annotations like we have done for several previous loop programs, and then get guidance toward the right solution. But we can also try to think of how these assignments are to be used and then work with the program verifier to see if we're right.

Surely, our loop body is going to inspect the next elements of a and b . These can compare in three ways, so we'll write a conditional statement with three cases:

```
if
  case a[m] == b[n] =>
    ?
  case a[m] < b[n] =>
    ?
  case b[n] < a[m] =>
    ?
```

If $a[m]$ and $b[n]$ are equal, we have detected another coincidence, so we should increment c , and then we are

done processing those two elements, so we also increment m and n :

```
c, m, n := c + 1, m + 1, n + 1;
```

If $a[m]$ is less than $b[n]$, then it is less than all of the remaining elements of b (since b is sorted). From this, we conclude that $a[m]$ will never contribute to the coincidence count, so we might as well move past it. We do that with the assignment

```
m := m + 1;
```

and symmetrically in the case where $b[n]$ is less than $a[m]$:

```
n := n + 1;
```

It seems we constructed this program by relying on our intuition and informal reasoning. That is true. But note how the loop invariant helped us get here. We started by designing a plausible loop specification from the method specification. Then, all we have to do is to write the loop body. To write the loop body, we need to think about how to change variables in such a way that we both decrease the termination metric and maintain the invariant. This is a simpler task than thinking about all possible loop-iteration sequences. Being a simpler task, we are more likely to be able to rely on our intuition and informal reasoning to write the non-loopy code of the loop body. So, the loop specification was a large part in making this possible.

It's also true that we're not done yet.

13.4.3. Proof

We constructed the `CoincidenceCount` loop body by thinking about the three cases. But you'll see that the verifier complains that it cannot prove the maintenance of the invariant about c . Either we made a mistake or the verifier needs some help from us. Either way, our response is the same: we start filling in the proof ourselves.

Let's start by figuring out which one of the three cases in the loop body the verifier has problems with. Copy the failing invariant into an **assert** statement at the end of each **case**:

```
assert c + |multiset(a[m..]) * multiset(b[n..])|  
      == |multiset(a[..]) * multiset(b[..])|;
```

This stops the complaint about the loop invariant. This is because the program's control gets through an **assert** statement only if the asserted condition holds. By putting the assertion at the end of each path to the end of the loop body, any control that reaches that point satisfies the loop invariant, so there is no complaint about it.

Instead, we get a complaint about each assertion that cannot be proved. In our program, all three of them! We've got some work ahead of us. Let's do one at a time.

13.4.4. The coincidence case

In the $a[m] == b[n]$ case, we had argued, informally, that we detected a coincidence and therefore should be able to increment c and move past these array elements. Stated differently, we expect that the intersection of the elements in $a[m..]$ and $b[n..]$ should equal the element $a[m]$ together with the intersection of the remaining elements, $a[m+1..]$ and $b[n+1..]$. Let's state this in an assertion in this **case**:

```
assert multiset(a[m..]) * multiset(b[n..])  
      == multiset{a[m]} + (multiset(a[m+1..]) *  
multiset(b[n+1..]));
```

The verifier complains about this assertion, but no longer complains about the assertion about c in this **case**. This means that our reasoning seems good, and all we now need to do is convince the verifier that this assertion holds.

We could do the proof of the asserted condition right here in the body of `CoincidenceCount`, but I'm going to write it as a separate lemma. The lemma will have this assertion as

its postcondition. As the precondition of the lemma, we need to collect from this **case** all the context we need for the lemma:

```
lemma MultisetIntersectionPrefix(a: array<int>, b: array<int>,
                                m: nat, n: nat)
  requires m < a.Length && n < b.Length
  requires a[m] == b[n]
  ensures multiset(a[m..]) * multiset(b[n..])
          == multiset{a[m]} + (multiset(a[m+1..]) *
multiset(b[n+1..]))
```

To make sure we wrote this correctly, let's insert a call to this lemma from the **case** we're working on. We might as well erase the assertions we added for debugging purposes in that **case**. The first **case** of `CoincidenceCount` then looks like:

```
case a[m] == b[n] =>
  MultisetIntersectionPrefix(a, b, m, n);
  c, m, n := c + 1, m + 1, n + 1;
```

The verifier can now prove this path through the loop body, so let's shift our attention to the proof of the new lemma.

We'll write a proof calculation, as so often is the case in a lemma. We'll start from the left-hand side of the lemma's postcondition and try to get to the right-hand side. Before the calculation, I will define `E` to denote the singleton multiset `a[m]`, which will be convenient in a moment.

```
{
  var E := multiset{a[m]};
  calc {
    multiset(a[m..]) * multiset(b[n..]);
```

Next, we want to lop off `E` from each of the multisets:

```
==
  (E + multiset(a[m+1..])) * (E + multiset(b[n+1..]));
```

but the verifier complains it can't prove this. Don't lose face. We'll just need to supply more details to help out the verifier.

We actually did two mini-steps in this step. One mini-step is that we split the sequence `a[m..]` into the concatenation

of the singleton sequence $[a[m]]$ and $a[m+1..]$ (and the analogous split for $b[n..]$). The other mini-step is that we changed the multiset construction of this concatenation into the union of two multisets:

```
multiset(A + A') == multiset(A) + multiset(A')
```

where A and A' are $[a[m]]$ and $a[m+1..]$.

The first mini-step replaces, in the argument to a function (here, function **multiset**), one expression for a sequence by another expression for the same sequence. This is a situation where we usually need to help the verifier out. I mentioned this same thing, but about multisets instead of sequences, as the “First piece of advice” in Section 10.2.4. Let's try phrasing this equality in the hint of the calculation step we just tried:

```
multiset(a[m..]) * multiset(b[n..]);
== { assert a[m..] == [a[m]] + a[m+1..]
    && b[n..] == [b[n]] + b[n+1..]; }
    (E + multiset(a[m+1..])) * (E + multiset(b[n+1..]));
```

Yes, that works! (Evidently, the verifier figured out the second mini-step by itself.)

From here, we distribute the intersection over union:

```
== // distribute * over +
    E + (multiset(a[m+1..]) * multiset(b[n+1..]));
}
```

This completes the proof of the lemma—the lemma we used to prove that the first **case** in the loop of `CoincidenceCount` maintains the invariant.

13.4.5. The first non-coincidence case

In the case for $a[m] < b[n]$, we had argued that we can remove element $a[m]$ from consideration, since it is not part of $b[n..]$. We can write down the condition we need as an assertion in that **case**:

```
assert multiset(a[m..]) * multiset(b[n..])
```

```
== multiset(a[m+1..]) * multiset(b[n..]);
```

The verifier complains about this assertion, but it no longer complains about the assertion about *c* in this case. This tells us that all we need to do is prove this assertion.

Let's write a lemma that captures the property we need. Rather than passing *b* and *n* as separate parameters, we'll think of `multiset(b[n..])` as a multiset *B*:

```
lemma MultisetIntersectionAdvance(a: array<int>, m: nat,
                                B: multiset<int>)
  requires m < a.Length && a[m] !in B
  ensures multiset(a[m..]) * B == multiset(a[m+1..]) * B
```

To make sure we stated the lemma we need, let's clean up the second **case** and call this lemma:

```
case a[m] < b[n] =>
  MultisetIntersectionAdvance(a, m, multiset(b[n..]));
  m := m + 1;
```

This case now verifies, so we'll switch our attention to proving the lemma. Let's start this proof in a way similar to how we started the proof of the previous lemma:

```
{
  var E := multiset{a[m]};
  calc {
    multiset(a[m..]) * B;
  } == { assert a[m..] == [a[m]] + a[m+1..]; }
    (E + multiset(a[m+1..])) * B;
```

Next, we distribute intersection over union:

```
== // distribute * over +
   (E * B) + (multiset(a[m+1..]) * B);
```

The verifier can now fill in the remaining proof glue by itself, but I think it looks nice to include one more step in the calculation:

```
== { assert E * B == multiset{}; }
   multiset(a[m+1..]) * B;
}
```

Two down, one to go.

13.4.6. The second non-coincidence case

The case $b[n] < a[m]$ is symmetric to the case $a[m] < b[n]$ we just did. Rather than writing another lemma like the previous, let's use the previous one. We'll invoke the lemma as:

```
MultisetIntersectionAdvance(b, n, multiset(a[m..]));
```

Unfortunately, the verifier is still not able to prove the assertion about c in this third **case**, so we'll need to supply more of the proof ourselves.

The difference between this case and the previous is that a and b (and m and n) have swapped places. By calling the same lemma that we developed before, we end up with a property where b is in the left argument of the intersection and a is on the right. You and I know that intersection is commutative, so this order ought not to matter. But this is once again an equality about multisets that is used in a function (here, the function $|\_|$ for multiset cardinality), so we'll need to mention this equality explicitly in order for the verifier to get it.

We replace the final **case** of the loop body with this proof calculation (and assignment of n):

```
case b[n] < a[m] =>
  calc {
    multiset(a[m..]) * multiset(b[n..]);
    == // multiset union is commutative
    multiset(b[n..]) * multiset(a[m..]);
    == { MultisetIntersectionAdvance(b, n,
                                     multiset(a[m..])); }
    multiset(b[n+1..]) * multiset(a[m..]);
    == // multiset union is commutative
    multiset(a[m..]) * multiset(b[n+1..]);
  }
  n := n + 1;
```

This completes the proof of our `CoincidenceCount` method.

Exercise 13.13.

Write a more symmetric lemma that can be used in both of the non-coincidence cases.

13.4.7. Summary of the development

To write the method that computes the coincidence count of two sorted arrays, we wrote the method specification (in terms of multisets), then developed a loop specification, and then, with a much smaller problem left to solve, reasoned informally about how to implement the loop body (that is, how to implement one arbitrary iteration of the loop).

The proof of the loop body required much more effort. We had to fill in various proof steps ourselves, mostly to obtain *extensionality* of both sequences and multisets. I showed by example how to debug the verifier's failed attempt to automatically produce the whole proof. This is typical—to either find the problem in our own reasoning or figure out what it is the proof needs that we haven't yet supplied, we start writing the proof ourselves, like we practiced in Chapter 5 and many subsequent chapters.

13.5. Slope Search

Slope Search looks for a key in a sloping landscape. More precisely, the landscape is a rectangular area where any step North or East takes you to higher (or unchanged) elevations. A typical way to indicate elevations in a 2-dimensional map is to draw *contour lines* that connect the points with equal elevation. For illustration, I've drawn such contour lines on the grid in Figure 13.0.

The lower left-hand corner has the lowest elevation and the upper right-hand corner has the highest elevation. The grid is suggestive of the 2-dimensional array (that is, matrix) given as input to the slope search. The elements of

the array are integers that represent the elevation, and the “key” we are to find is a point that has a particular elevation.

In terms of the the matrix *a*, the slope of the landscape is expressed by the following two quantifiers:

```
(forall i, j, j' ::
  0 <= i < a.Length0 && 0 <= j < j' < a.Length1 ==>
  a[i,j] <= a[i,j']) &&
(forall i, i', j ::
  0 <= i < i' < a.Length0 && 0 <= j < a.Length1 ==>
  a[i,j] <= a[i',j])
```

We're given one more thing: there exists a point with elevation *key*. From these pieces of information, we are ready to write the specification of the method to implement:

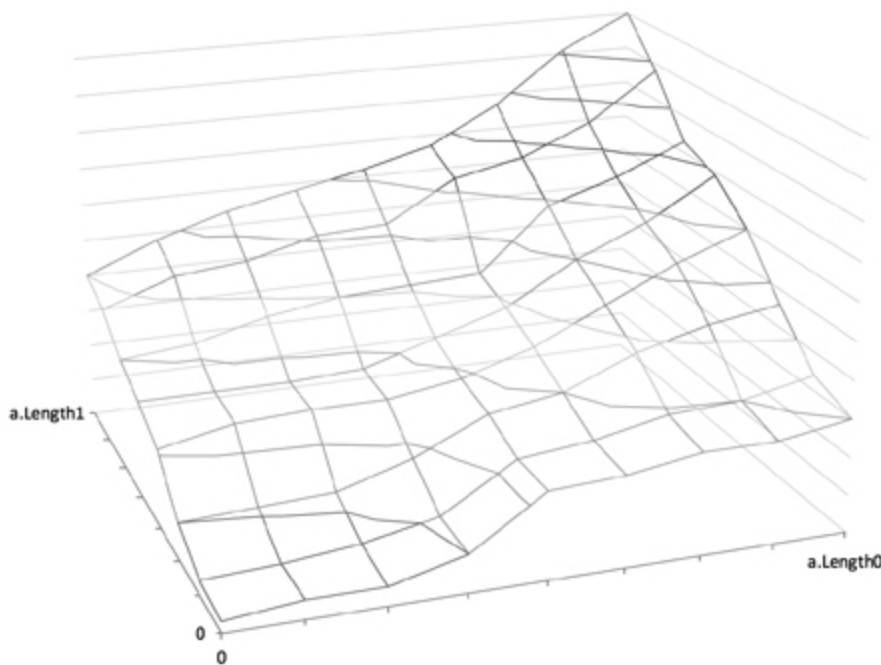


Figure 13.0 Example input to Slope Search. The diagram shows contour lines that indicate where the matrix values cross certain “elevation” thresholds.

```

method SlopeSearch(a: array2<int>, key: int) returns (m: int,
n: int)
  requires forall i, j, j' ::
    0 <= i < a.Length0 && 0 <= j < j' < a.Length1 ==>
      a[i,j] <= a[i,j']
  requires forall i, i', j ::
    0 <= i < i' < a.Length0 && 0 <= j < a.Length1 ==>
      a[i,j] <= a[i',j]
  requires exists i, j ::
    0 <= i < a.Length0 && 0 <= j < a.Length1 && a[i,j] == key
  ensures 0 <= m < a.Length0 && 0 <= n < a.Length1
  ensures a[m,n] == key

```

13.5.0. Starting position of the search

The loop index for our program won't be a single integer, but a pair of integers that will index into the matrix. The first thought that comes to mind is to start the loop index at the origin and iteratively increase the position of one or both of the dimensions. After a while, the search would reach the contour line for key, that is, a place where values on one side are less than key and values on the other side are key or higher. But then what? Should the search now continue clockwise along the contour line or counterclockwise?

The simplest way to follow a contour line is to start at one of its ends. We can start where the key contour line hits the x-axis or we can start where it hits the y-axis. It doesn't matter which, so let's say we want to start searching this contour line from the y-axis.

Alright, so how do we find where the key contour line hits the y-axis? We can do this with a linear search from the origin. Or, faster would be to do a binary search along the y-axis. For our example, I will do a linear search, but starting from the opposite side from the origin (that is, from the upper left-hand side of the grid). This will lead to a simple algorithm with one loop, where every step will either go right or down.

As we move the loop index right and down, our invariant is that the value `key` exists in the rectangle whose upper left-hand corner is our loop index.

13.5.1. The implementation

I just described the salient parts of the slope-search algorithm: the starting position and the invariant, and the idea that we're going to follow one contour line. Since the contour line weaves between the discrete points in the matrix, it still seems quite tricky to make sure our program doesn't accidentally miss the key. Let's write out the program and see if we're able to prove it.

We'll make the out-parameters `m,n` be the loop index, and we'll maintain as an invariant that `m,n` be a proper array index. Then, the loop guard is especially easy to write. Here's the initialization and loop specification:

```
{
  m, n := 0, a.Length1 - 1;
  while a[m,n] != key
    invariant 0 <= m < a.Length0 && 0 <= n < a.Length1
    invariant exists i, j ::
      m <= i < a.Length0 && 0 <= j <= n && a[i,j] == key
    decreases a.Length0 - m + n
}
```

The postcondition follows from the first invariant and the negation of the loop guard. In fact, this is an instance of the *omit a conjunct* Loop Design Technique 11.0. The second loop invariant is an instance of the *replace constant by variable in precondition* Loop Design Technique 13.0. Since we decided to let `m,n` be a proper index into the array rather than being limits for our search, note that the constant that `n` is replacing is `a.Length1 - 1`.

The **decreases** clause speaks to our decision to increase `m` and decrement `n`.

In the body of the loop, we need to compare `a[m,n]` with `key`. If it is less than `key`, we move to a higher elevation by

increasing m . If it exceeds key , then we move to a lower elevation by decreasing n .

```
{  
  if a[m,n] < key {  
    m := m + 1;  
  } else {  
    n := n - 1;  
  }  
}
```

This maintains the invariant and it decreases the termination metric, so we are done!

13.6. Canyon Search

A problem related to Slope Search is Canyon Search, where instead of searching for a key in a sloping landscape, we're searching for the lowest point in a canyon. Specifically, we're given two sorted arrays, a and b , and are asked to find the minimum absolute difference between an element of a and an element of b .

Let's start by writing the specification, then give the problem some thought, and finally implement the method using a loop.

13.6.0. Method specification

The problem mentions the absolute difference, or call it *distance*, between two values. We'll define a function to express distance:

```
function Dist(x: int, y: int): nat {  
  if x < y then y - x else x - y  
}
```

Our method starts with two nonempty, sorted arrays and returns a measure of distance:

```
method CanyonSearch(a: array<int>, b: array<int>) returns (d:  
nat)
```

```

requires a.Length != 0 && b.Length != 0
requires forall i, j :: 0 <= i < j < a.Length ==> a[i] <=
a[j]
requires forall i, j :: 0 <= i < j < b.Length ==> b[i] <=
b[j]

```

As the postcondition, we'll use the representation of minimum that we've used before, most recently in Section 13.3. It says that the minimum is among the candidate values and that the minimum is not greater than any candidate value:

```

ensures exists i, j ::
  0 <= i < a.Length && 0 <= j < b.Length && d == Dist(a[i],
b[j])
ensures
  forall i, j :: 0 <= i < a.Length && 0 <= j < b.Length ==>
  d <= Dist(a[i], b[j])

```

13.6.1. About the canyon

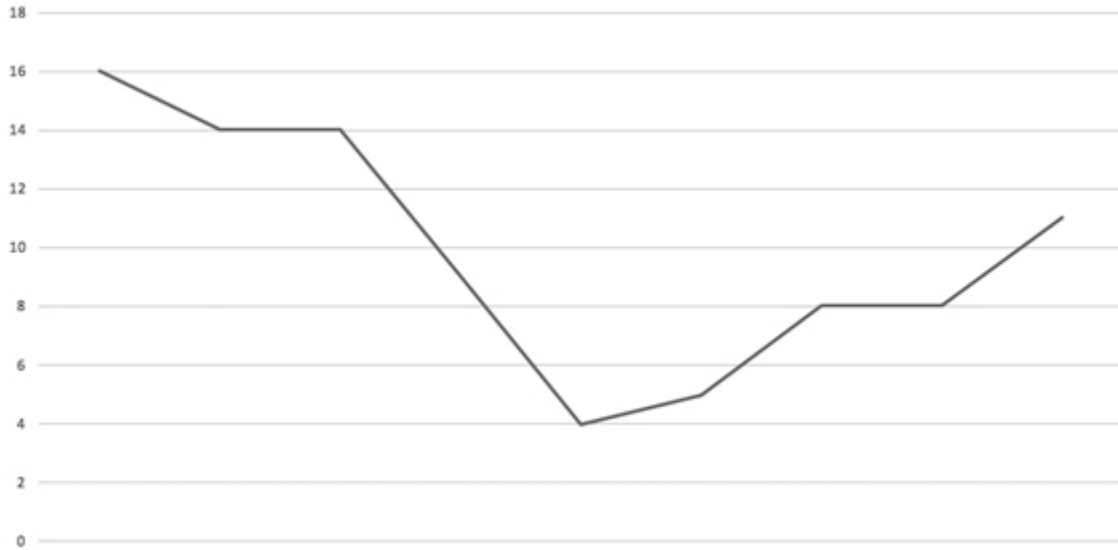
Given the two arrays, we can think of the `a.Length * b.Length` landscape where the element at index `i,j` is `Dist(a[i], b[j])`. Consider a cross section of this matrix, say along the `a`-axis where `j` is fixed. As an example, suppose the elements of `a` are

3, 5, 5, 10, 23, 24, 27, 27, 30

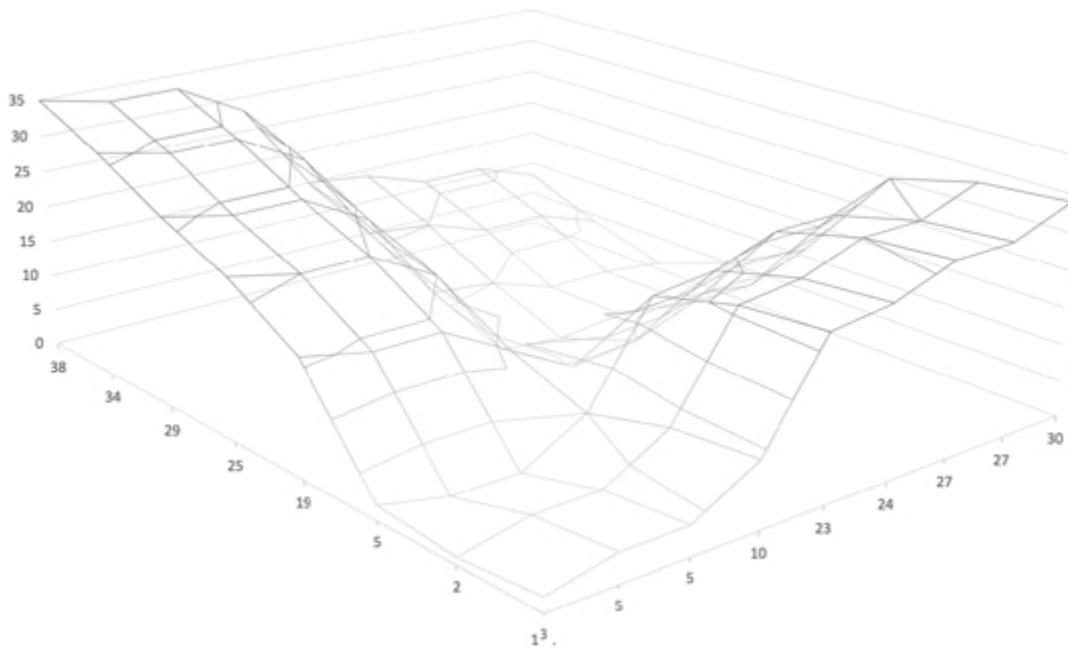
and that `b[j]` is 19. The distances between the `a` values and `b[j]` are

16, 14, 14, 9, 4, 5, 8, 8, 11

which we can plot as follows:



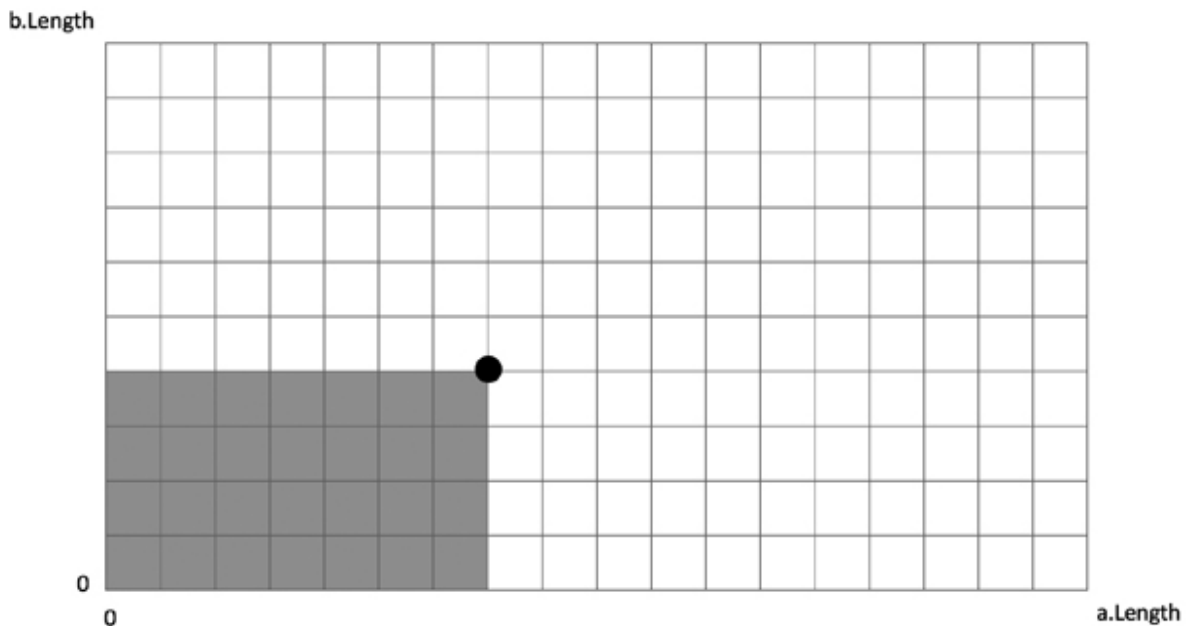
This diagram has the shape of a “V”, whose lowest point is where the a value is closest to $b[j]$. This will be true for every cross section in both dimensions. The whole 2-dimensional landscape thus looks like a canyon, which explains the name of the algorithm. The following diagram may help you visualize the situation:



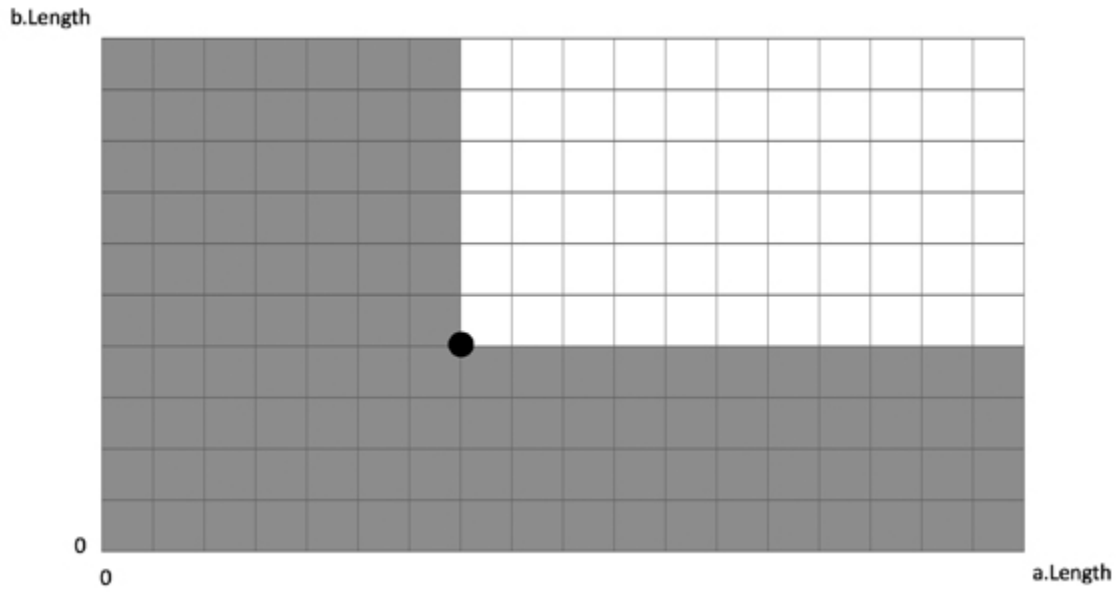
The canyon shape gives us hope of writing an algorithm that searches along the canyon floor, that is, where the

loop index tries to stay close to the lowest point of each V.

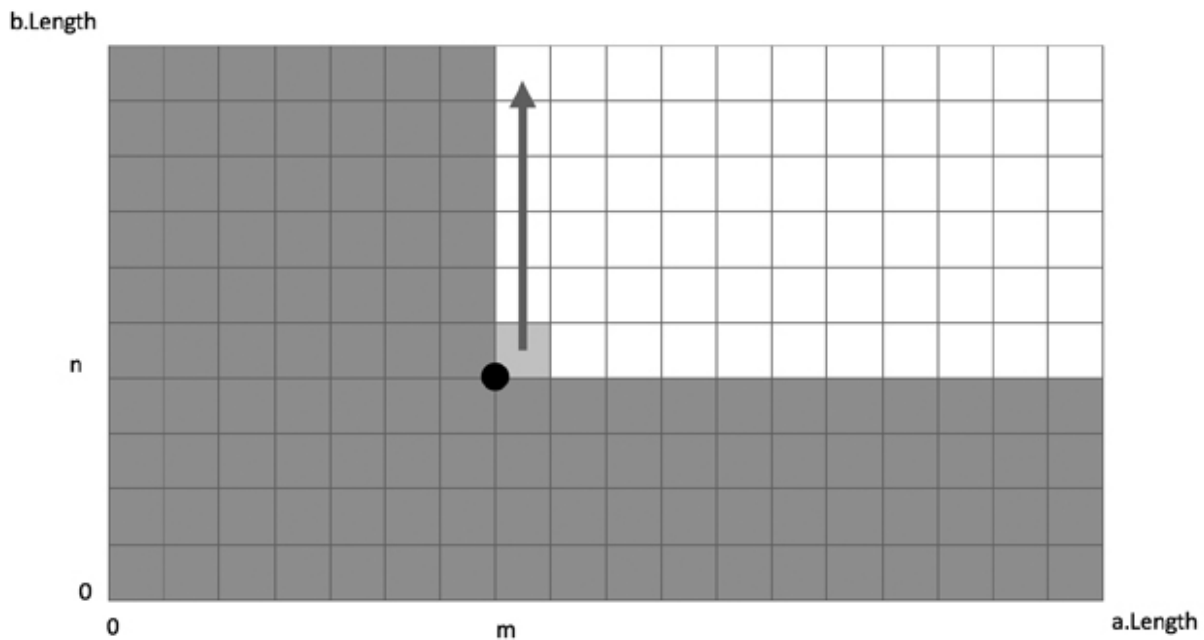
For the invariant, we need to include something that resembles the universal quantifier in the postcondition. In the postcondition, it says that d is below the distance of all pairs of points. We could maintain an invariant that says d is below all points where both coordinates are below the loop index. Pictorially, this says that d is below those points in the shaded region in the following diagram:



However, this is not so easy when the loop index reaches an edge of the landscape. Better is to aim for an invariant where d is below all points where *either* coordinate is below the loop index, as illustrated by the shaded region in the following diagram:



How can we maintain such an invariant? Let m, n be the loop index. If $a[m]$ is below $b[n]$, then $a[m]$ is below all values in b from index n upward. Pictorially, the lightly shaded square in



represents $\text{Dist}(a[m], b[n])$. If $a[m] \leq b[n]$, then the distances recorded in landscape values grow along the arrow. So, we only need to update d if $\text{Dist}(a[m], b[n])$ is a

new minimum, and then we can maintain the invariant by incrementing m . Symmetrically, if $b[n] \leq a[m]$, we can increment n .

13.6.2. Method implementation

We now have all the ideas we need to implement the method with a loop. We use loop indices m and n :

```
var m, n := 0, 0;
while m < a.Length && n < b.Length
  invariant 0 <= m <= a.Length && 0 <= n <= b.Length
  decreases a.Length - m + b.Length - n
```

We apply the *always* Loop Design Technique 13.1 for the existential postcondition:

```
invariant exists i, j ::
  0 <= i < a.Length && 0 <= j < b.Length &&
  d == Dist(a[i], b[j])
```

which means we'll need to initialize d to the distance between any two array values. The distance $\text{Dist}(a[0], b[0])$ will do, so we write the following at the beginning of the method body, before the loop:

```
{
  d := Dist(a[0], b[0]);
```

The universal quantifier in the method's postcondition remains. We determined above that we want it to cover the L-shaped region. We do that by excluding the points i, j such that

```
m <= i < a.Length && n <= j < b.Length
```

Our last invariant is therefore

```
invariant forall i, j ::
  0 <= i < a.Length && 0 <= j < b.Length ==>
  d <= Dist(a[i], b[j]) || (m <= i && n <= j)
}
```

What we did for this invariant is a good general technique:

||

Loop design technique 13.2. (Weaken)

To establish a postcondition Q , use a loop invariant that weakens Q by a predicate R .

For the loop body, we first update d , in case the current position is a new minimum so far:

```
{  
  d := if Dist(b[n], a[m]) < d then Dist(b[n], a[m]) else d;
```

The rest of the loop body increments m or n according to our observations in Section 13.6.1:

```
  if  
  case a[m] <= b[n] =>  
    m := m + 1;  
  case b[n] <= a[m] =>  
    n := n + 1;  
}
```

The verifier is happy with this program, so we are done!

Exercise 13.14.

The way I wrote the **if-case** statement, either alternative can be taken if $a[m]$ and $b[n]$ are equal. However, if they are equal, then we have $\text{Dist}(a[m], b[n]) == 0$, so there is no reason to continue the search. Adjust the **if** statement to reflect this observation.

Exercise 13.15.

Given three sorted arrays, a , b , and c , specify and implement a program that finds the smallest measure of $\text{Dist}(x, y) + \text{Dist}(y, z) + \text{Dist}(z, x)$ where x, y, z range over all the elements of a, b, c , respectively.

13.7. Majority Vote

As the last example in this chapter, I will present a sweet algorithm by Boyer and Moore called Majority Vote. For the

problem, we are given a collection of votes. Each vote is for one candidate and we're told that one candidate has a strict majority of the votes. That is, strictly more than half of the votes are for one specific candidate. The problem is to find that candidate. And to do so in linear time, that is, by examining each vote only once.

Instead of using arrays for this program, I will use sequences (see Section 13.0.4). As part of the example, I will show an advanced specification technique that involves ghost parameters.

13.7.0. Counting occurrences

Here is a function for counting occurrences in a sequence.

```
function Count<T(==)>(a: seq<T>, lo: int, hi: int, x: T): nat
  requires 0 <= lo <= hi <= |a|
{
  if lo == hi then
    0
  else
    Count(a, lo, hi - 1, x) + if a[hi - 1] == x then 1 else 0
}
```

It returns the number of times x occurs in the sequence a from index lo to index hi . Because this compilable function needs to compare elements of type τ , the type parameter is restricted to types that can be compared, as indicated by the suffix $(==)$ (see also Section 6.5). The form of the body of `Count` is like that of function `SumDown` in Section 12.3.0, which takes us in the direction of a what-has-been-done invariant (see Section 12.3).

While we have our eyes on this function, let's prove a couple of properties of it. One property is that the number of occurrences in the subsequence $a[lo..hi]$ can be split into those in $a[lo..mid]$ and those in $a[mid..hi]$:

```
lemma SplitCount<T>(a: seq<T>, lo: int, mid: int, hi: int, x:
T)
  requires 0 <= lo <= mid <= hi <= |a|
```

```

ensures Count(a, lo, mid, x) + Count(a, mid, hi, x)
    == Count(a, lo, hi, x);
{
}

```

The proof is completed by Dafny's automatic induction.

Sidebar 13.0

Two half-open intervals are *consecutive* if the upper bound of one is the lower bound of the other.

Consecutive intervals are easily combined by dropping the two common bounds. For example,

`a[3..9] + a[9..15] == a[3..15]`

which is to say that concatenating the 9 - 3 elements from 3 on with the 15 - 9 elements from 9 on gives you the 15 - 3 elements from 3 on.

As another example, if `lo <= mid <= hi`, then the two quantifiers in the conjunction

`(forall i :: lo <= i < mid ==> P(i)) &&`
`(forall i :: mid <= i < hi ==> P(i))`

can be combined into one:

`forall i :: lo <= i < hi ==> P(i)`

In words, if *P* holds from *lo* to *mid* and from *mid* to *hi* (remember that I'm using the word "to" for half-open intervals, see Sidebar 1.0), then *P* holds from *lo* to *hi*. For summations, it looks like this:

$$\sum_{i:=lo}^{<mid} F(i) + \sum_{i:=mid}^{<hi} F(i) = \sum_{i:=lo}^{<hi} F(i)$$

The other property is that the number of occurrences of two distinct elements in the subsequence $a[lo..hi]$ does not exceed $hi - lo$, and its proof is also completed by Dafny's automatic induction:

```
lemma DistinctCounts<T>(a: seq<T>, lo: int, hi: int, x: T, y:
T)
  requires 0 <= lo <= hi <= |a|
  ensures x != y ==>
    Count(a, lo, hi, x) + Count(a, lo, hi, y) <= hi - lo
{
}
}
```

I've chosen to write $x \neq y$ as the antecedent of an implication in the lemma's post-condition rather than writing it in the lemma's precondition. This means the lemma is callable regardless of if x and y are distinct, and thus the caller easily obtains the *contrapositive* of this implication. (For some tradeoffs between these two specification styles in lemmas, see the discussion of `LessTransitive` in Section 7.1.)

Exercise 13.16.

Mark both of the lemmas above with the attribute `{:induction false}` to turn off automatic induction. Then, write a proof for each lemma.

13.7.1. Method specification

To specify the problem of majority vote, we define a predicate that says what it means to have a strict majority. For x to have a strict majority among the votes in a from index lo to hi , we can write

```
(hi - lo) / 2 < Count(a, lo, hi, x)
```

However, I find it easier to work with $2 *$ on the right than with $/ 2$ on the left, so I will equivalently define the predicate as follows:

```
predicate HasMajority<T(==)>(a: seq<T>, lo: int, hi: int, x: T)
```

```

requires 0 <= lo <= hi <= |a|
{
  hi - lo < 2 * Count(a, lo, hi, x)
}

```

To specify a method for computing the winner, we could write

```

method FindWinner'<Candidate(==)>(a: seq<Candidate>)
  returns (k: Candidate)
  requires exists K :: HasMajority(a, 0, |a|, K)
  ensures HasMajority(a, 0, |a|, k)

```

The precondition says that a majority winner exists. As it turns out, it will be convenient to be able to name this winner in the implementation of the method, but the scope of the existentially bound variable k is confined to the precondition. Here is where the technique of using a ghost parameter, which I alluded to above, comes in. Here's how I will write the method specification:

```

method FindWinner<Candidate(==)>(a: seq<Candidate>, ghost K:
Candidate)
  returns (k: Candidate)
  requires HasMajority(a, 0, |a|, K)
  ensures k == K

```

You may find this specification peculiar at first, because it promises to find the winner, provided the winner is already passed in as a parameter. So, then, the method can easily be implemented by one assignment:

```

k := K;

```

No, not quite. k is ghost, and you're not allowed to assign a ghost variable to a compiled variable. So, the implementation of the method must compute k some other way. It is only the specification that uses k . Let's now focus on writing the method implementation, and I'll later return to the question—or, mystery, you may think—of how a caller will be able to pass in a value for k .

13.7.2. Design of implementation

In our standard fashion, we'll use a loop index—say, n —that ranges from 0 through $|a|$. As we go along, we'll use k to keep track of the “leading candidate”, that is, the candidate that has received the largest number of votes among the first n votes. We'll also use a variable c to keep track of how many votes for k we have seen so far. This works great, as long as k continues to be the leading candidate. However, it may be that some candidate has most of the votes early in the sequence, but is later overtaken by another candidate. When our loop detects that a leading candidate is no longer in the lead, we need to change k , but how are we to update c without rescanning the early part of the sequence?

The beauty of this algorithm comes from the realization that we can ignore any prefix of the sequence where there is no majority winner. Consider the following diagram, which illustrates the first n votes and the remaining $|a| - n$ votes:



We are given that k has most of the $|a|$ votes. In other words, the density of votes for k in a is greater than $1/2$. If a portion of the sequence has a k -density that's less than $1/2$, then the rest of the sequence must have a k -density that's even higher than $1/2$. So, if among the first n elements of a , there is no candidate with a majority vote, we can just ignore those elements, since the winner we're looking for has a majority among the remaining elements.

Instead of computing k and c in $a[..n]$, we'll compute them for $a[lo..hi]$. That is, in addition to variables k and c , we'll use two variables, lo and hi . hi is the length of the prefix of a that we have examined so far, and lo is the length of a prefix of a that has no majority winner.

13.7.3. Loop specification

From the design sketched above, we write the following loop specification:

```
{
  k := a[0];
  var lo, hi, c := 0, 1, 1;
  while hi < |a|
    invariant 0 <= lo <= hi <= |a|
    invariant c == Count(a, lo, hi, k)
    invariant HasMajority(a, lo, hi, k)
    invariant HasMajority(a, lo, |a|, K)
```

Evidently, this is not sufficient to prove the postcondition, for the verifier complains (which you'll see if you fill in the final } for the method body). The invariant and the negation of the guard give us

$\text{HasMajority}(a, lo, |a|, k) \ \&\& \ \text{HasMajority}(a, lo, |a|, K)$

This says that both k and K have a majority of votes in a part of the sequence. Intuitively, only one candidate can have a strict majority, so the conditions we have ought to imply $k == K$. The verifier needs help to see this. Luckily, (the contrapositive of) one of the lemmas we wrote in Section 13.7.0, along with the definition of `HasMajority`, expresses exactly this. Therefore, we'll end the method body with

```
  DistinctCounts(a, lo, |a|, k, K);
}
```

and now, only the loop body remains to be written.

13.7.4. Loop implementation

If the next vote to be examined is for the leading candidate, all we need to do is increase (the loop index and) the current count for that candidate:

```
{
  if a[hi] == k {
    hi, c := hi + 1, c + 1;
```

Even with $a[hi]$ being a vote for another candidate, it may be that k remains in the lead in $a[lo..hi+1]$:

```
    } else if hi + 1 - lo < 2 * c {  
        hi := hi + 1;
```

In the remaining case, we have detected that there is no majority winner through this vote, so we'll ignore the portion of a up to $hi + 1$:

```
    } else {  
        hi := hi + 1;
```

Next, we have to reset the variables to start the search anew. The most difficult part of the loop invariants to reestablish is the last one. The reasoning goes like this:

At this time, we know that k has exactly half of the votes in $a[lo..hi]$.

```
    calc {  
        true;  
    ==>  
        2 * Count(a, lo, hi, k) == hi - lo;
```

This means no candidate, k included, can have more than half the votes—if k and k are the same, then k has exactly half the vote; otherwise, lemma `DistinctCounts` from Section 13.7.0 says that k has no more votes than the other half.

```
    ==> { DistinctCounts(a, lo, hi, k, K); }  
        2 * Count(a, lo, hi, K) <= hi - lo;
```

The invariant on entry to the loop tells us that k has a majority of the votes in $a[lo..|a|]$, so by splitting the range into $a[lo..hi]$ and $a[hi..|a|]$, we conclude that k has a majority of the votes in the latter.

```
    ==> { SplitCount(a, lo, hi, |a|, K); }  
        |a| - hi < 2 * Count(a, hi, |a|, K);  
    == // def. HasMajority  
        HasMajority(a, hi, |a|, K);  
    }
```

If you don't care to include the thread of reasoning in this proof calculation, you can replace it with just the calls to

the two lemmas:

```
DistinctCounts(a, lo, hi, k, K);  
SplitCount(a, lo, hi, |a|, K);
```

We are now ready for the final assignment in this branch of the `if` statement in the loop body:

```
    k, lo, hi, c := a[hi], hi, hi + 1, 1;  
  }  
}
```

And that concludes our implementation of method `FindWinner`.

13.7.5. Determining a winner, if there is one

In Section 13.7.1, I said I would come back to the mystery of how a caller of `FindWinner` can pass in `K`, which is the value that `FindWinner` is going to compute. I will reveal the solution to that mystery now.

Assign-such-that statement

To do that, I first need to introduce the assign-such-that statement in Dafny. It has the form

```
x :| P;
```

and it assigns to `x` a value such that `P` holds. A proof obligation of this statement is that there exists a value that `x` can take on. For example, the statement

```
x :| 0 <= x < N;
```

sets `x` to an arbitrary natural number less than `N`. This is possible only if `N` is strictly positive, so the proof obligation of this statement is exactly that. To be technical about it, the proof obligation has the form

```
exists x :: 0 <= x < N
```

but, in this case, that just amounts to `N` being strictly above 0.

Implementing FindWinner'

Using the assign-such-that statement, we can implement the method FindWinner', which I considered briefly in Section 13.7.1 and which has a more conventional-looking precondition—more to the point, FindWinner' does not have a ghost in-parameter.

```
method FindWinner'<Candidate(==)>(a: seq<Candidate>)
  returns (k: Candidate)
  requires exists K :: HasMajority(a, 0, |a|, K)
  ensures HasMajority(a, 0, |a|, k)
{
  ghost var K :| HasMajority(a, 0, |a|, K);
  k := FindWinner(a, K);
}
```

This method introduces a ghost variable K and sets it to a value satisfying $\text{HasMajority}(a, 0, |a|, K)$. This comes with a proof obligation that such a K exists, which follows directly from the precondition of FindWinner'. Once K has been assigned, the code above uses it when calling FindWinner.

By the way, there is no reason to worry about how the assign-such-that statement finds such a K , since it's but a ghost assignment. We prove there exists a value for K and that's all we need to know.

Searching for a winner

Method FindWinner' shows a useful basic idea, but as a method, it isn't really any more useful than FindWinner. I mean, how is the caller of FindWinner' going to discharge the proof obligation that such a K exists? Since we've come this far, let's take two more steps and turn the algorithm into something useful we can call.

We can make a small adjustment to FindWinner to make it callable even if there isn't a majority winner. If there isn't, we can't say anything about the value that's returned. Our first of two steps toward a useful method is to turn FindWinner into a new method, SearchForWinner, which takes

yet another ghost parameter, `hasWinner`. If `hasWinner` is **true**, then `SearchForWinner` will act like `FindWinner` and will return the majority winner. If `hasWinner` is **false**, we have no guarantees about what is returned.

```

method SearchForWinner<Candidate(==)>(a: seq<Candidate>,
                                     ghost hasWinner: bool,
                                     ghost K: Candidate)
    returns (k: Candidate)
    requires |a| != 0
    requires hasWinner ==> HasMajority(a, 0, |a|, K)
    ensures hasWinner ==> k == K
{
    k := a[0];
    var lo, hi, c := 0, 1, 1;
    while hi < |a|
        invariant 0 <= lo <= hi <= |a|
        invariant c == Count(a, lo, hi, k)
        invariant HasMajority(a, lo, hi, k)
        invariant hasWinner ==> HasMajority(a, lo, |a|, K)
    {
        if a[hi] == k {
            hi, c := hi + 1, c + 1;
        } else if hi + 1 - lo < 2 * c {
            hi := hi + 1;
        } else {
            hi := hi + 1;
            DistinctCounts(a, lo, hi, k, K);
            SplitCount(a, lo, hi, |a|, K);
            if hi == |a| {
                return;
            }
            k, lo, hi, c := a[hi], hi, hi + 1, 1;
        }
    }
    DistinctCounts(a, lo, |a|, k, K);
}

```

The differences between this method and `FindWinner` are:

- The precondition `HasMajority(a, 0, |a|, K)` needs to hold only if `hasWinner` is **true**.
- Method `FindWinner`'s `HasMajority` precondition implies `a` is nonempty, but since we have weakened that

precondition, we'll add $|a| \neq 0$ as a separate precondition.

- The postcondition $k == K$ is similarly weakened to $\text{hasWinner} \implies k == K$
- The fourth loop invariant also gets the antecedent hasWinner , but the other invariants are the same as in `FindWinner`.
- When we restart the variables to start the search anew, we must first check if we have explored all of the sequence elements. This was unnecessary in `FindWinner`, since it assumes there is a majority winner. The new method terminates the loop abruptly with a **return** statement in the event that all sequence elements have been explored.

Method `SearchForWinner` can be called with an arbitrary value for K by just setting `hasWinner` to **false**. But we mustn't always do that, for then we don't learn anything from what `SearchForWinner` does.

This is where we use a second step.

A non-mysterious, but rather fantastical, use of `SearchForWinner`

We are now ready to write a method with no assumptions about the sequence it gets, and no ghost parameters either. So that it can signal to callers whether or not there is a majority winner, I declare the following type:

```
datatype Result<Candidate> = NoWinner | Winner(Candidate)
```

The useful method is now specified as follows:

```
method DetermineElection<Candidate>==(a: seq<Candidate>)  
  returns (result: Result<Candidate>)  
  ensures match result  
    case Winner(c) => HasMajority(a, 0, |a|, c)  
    case NoWinner => !exists c :: HasMajority(a, 0, |a|, c)
```

This method tells us exactly whether or not there is a majority winner; and if there is, it tells us what that winner

is.

The implementation starts off by checking for an empty sequence:

```
{  
  if |a| == 0 {  
    return NoWinner;  
  }  
}
```

This is because we can call `SearchForWinner` only if `a` is nonempty.

Next, the implementation introduces a ghost variable `hasWinner` and sets it to the boolean value that indicates whether or not there is a winner:

```
ghost var hasWinner := exists c :: HasMajority(a, 0, |a|, c);
```

Cool assignment, you gotta admit!

After that, the implementation introduces one more ghost variable, `w`, assigned such that *if* there is a winner, then `w` is that winner, and if there is no winner, then `w` is set to an arbitrary value of its type:

```
ghost var w;  
if hasWinner {  
  w := HasMajority(a, 0, |a|, w);  
} else {  
  w := a[0];  
}
```

The `assign-such-that` statement comes with a proof obligation that a value for `w` exists, and it follows from the guard `hasWinner` and the assignment to `hasWinner`.

With these two ghost variables (and knowing `|a| != 0`), we can call `SearchForWinner`:

```
var c := SearchForWinner(a, hasWinner, w);
```

The value of `c` is the majority winner, if there is one. It would therefore be nice if we could return

```
if hasWinner then Winner(c) else NoWinner
```

but we cannot, because `hasWinner` is a ghost variable. Instead, we scan the sequence once more to check whether

or not c is a majority winner. If it is, then we know that `hasWinner` is **true**. And since `SearchForWinner` is guaranteed to return the majority winner if there is one, then: if we find that c is not a majority winner, then apparently `hasWinner` must be **false**. Folks (drum roll), our useful method endeth thus:

```
    return if HasMajority(a, 0, |a|, c) then
        Winner(c)
    else
        NoWinner;
}
```

13.8. Summary

In this chapter, I have shown many example programs that use arrays or sequences. The algorithms we considered all do some kind of search into the given data, and their invariants often have some form like “we know the key is not here: . . . ” or “we know the key is somewhere here: . . . ”. Another common aspect of these programs is that their specifications use quantifications over the indices of the arrays.

The final implementations of the programs are deceptively simple. For many of them, it would be difficult to convince yourself of their correctness without some form of proof. The verifier checks our proof work to make sure we're not missing any cases.

For some of these programs, including the ostensibly trivial Linear Search, we spent a noticeable amount of time thinking about the program's specification. By being precise about the specification, we are ushered into making decisions about corner cases, like how and when to signal failure and whether or not to say which occurrence of the key to find. Once we write the implementation, the verifier checks that it satisfies the specification.

In the advanced Section 13.7, I put ghost parameters and ghost variables to good use. The ghost parameter k in the specification of `MajorityVote` gave us a name for the unknown winning candidate. Since a majority winner is unique (as follows from our lemma `DistinctCounts`), we were able to formulate the postcondition as the tastefully simple condition $k == K$. And in method `DetermineElection`, we used ghost constructs to determine values for the ghost parameters of `SearchForWinner`. While the ghosts helped the development of the program, the final program compiles and runs without them.

Exercise 13.17.

A consecutive stretch of equal elements in a sequence is called a *plateau*. (a) Define a ghost predicate `Plateau(s, lo, hi)` to say that $s[lo..hi]$ is a plateau. (b) Specify a method that, given an integer sequence s , returns the length of the longest plateau in s . (c) Implement the method.

Exercise 13.18.

Given three non-decreasing integer sequences with a common element, return a common element.

Notes

An implementation bug in the Java library's `Binary Search` method made headlines when, after years in deployment, it was finally discovered. The problem was the arithmetic overflow that the statement

```
mid := (lo + hi) / 2;
```

is susceptible to when the type of these variables is a machine integer, which is bounded. This is not an issue in Section 13.2.2, because Dafny's type `int` is unbounded. If your language uses bounded integers and the verifier

checks for arithmetic overflows, then you would get an error about the right-hand side of the assignment to `mid`. If your language uses signed integers and modular arithmetic (that is, a *wraparound* semantics), then the verification would report an error about the possible negative indexing in the expression `a[mid]`. I don't use them in this book, but the Dafny language includes declarations that allow you to define bounded integers. If Section 13.2.2 had used those, then the verifier would indeed complain about the possible arithmetic overflow.

Exercise 13.19.

How can you change the right-hand side of the assignment to `mid` so that none of its subexpressions causes an overflow.

The implementation of Slope Search in Section 13.5 is linear in the sides of the 2-dimensional map. It is possible to do better, by considering a generalization of Binary Search, as shown by Richard Bird [17]. Following Gries [55], Bird calls the algorithm Saddleback Search.

Some of the algorithms in the sections and exercises of this chapter are used in several classic textbooks on program proofs. Among them is *The Science of Programming* by David Gries, which also traces the origins of such algorithms [55].

The assign-such-that statement was introduced by Ralph Back under the name *nondeterministic assignment* [10, 6]. It, in combination with a precondition, has been used extensively by Carroll Morgan as a *specification statement* [93].

Chapter 14

Modifying Arrays



The many array programs we studied in the previous chapter have in common that they don't change the contents of the arrays. That's about to, ahem, change.

Because heap-allocated storage is accessed via references, the storage itself is not passed as parameters to functions and methods. Specifications nevertheless need to be able to identify the relevant pieces of the heap. For this purpose, we use a *frame*.

For now, our frames are going to be very simple. So simple that I'll postpone a general presentation of frames until Chapter 16. In Section 14.0, I'll tell you what you need to know about frames in this chapter and the next. In

Section 14.1, we'll practice writing simple programs that mutate arrays.

14.0. Simple Frames

This section gives a quick guide to working with frames for simple array programs.

14.0.0. Modifies clauses

Consider the `Min` method in Section 13.3.0. Our intention was that `Min` would return the minimum of the *given* array elements. But what if the method implementation were to *change* the value of every array element to, say, 1330 and then return the value 1330? Upon return from such an implementation, the expression given in `Min`'s **ensures** clause would hold. That is, when the method returns, the minimum of the array really *is* 1330! 🤖 Does our specification really allow such an implementation?

Luckily, no. Part of what a method specification does is describe what the method is allowed to modify and what it must leave unchanged. This is expressed by the method's *write frame*, which you specify using a **modifies** clause. Because the `Min` method has no **modifies** clause (stated differently, it has an *empty modifies* clause), it is not allowed to make modifications to the given array.

The use of frames is governed by a set of rules that I will refer to as *frame bylaws*. Here is the first frame bylaw:

If a method changes the elements of an array *a* given as a parameter, its specification must include **modifies** *a*.

Here is an example:

```
method SetEndpoints(a: array<int>, left: int, right: int)
```

```

    requires a.Length != 0
    modifies a
{
    a[0] := left;
    a[a.Length - 1] := right;
}

```

If no **modifies** clause were given, then the body of this method would not live up to its contract. The verifier would complain that the method modifies the contents of an array that is not disclosed in the method's **modifies** clause.

It's the array referenced by `a` that is allowed to be changed, according to the specification **modifies** `a`. For example,

```

method Aliases(a: array<int>, b: array<int>)
    requires 100 <= a.Length
    modifies a
{
    a[0] := 10;
    var c := a;
    if b == a {
        b[10] := b[0] + 1;
    }
    c[20] := a[14] + 2;
}

```

is correct, because at the time `b[10]` and `c[20]` are updated, it is known that these refer to elements of array `a`. Without the **if** `b == a` test, however, the assignment to `b[10]` would give rise to a verification error.

Exercise 14.0.

Try adding the postcondition

```
ensures a[0] == left && a[a.Length - 1] == right
```

to method `SetEndpoints`. Adjust the method's precondition (as little as possible) to make the method verify.

Exercise 14.1.

Add **modifies** `a` to the specification of method `Min` from Section 13.3.0 and then implement the method as the

| “make it 1330!” solution I sketched above.

14.0.1. Old values

Suppose you want to write the postcondition for a method that increments the elements of a given array. You must then express that the values of the elements on return from the method are larger than the values of the elements on entry to the method. In other words, the postcondition needs to refer to both the pre-state and post-state of the method. For this reason, a method postcondition is a *two-state predicate*.

An expression *E* in a method's **ensures** clause ordinarily refers to the value of *E* in the method's post-state. To refer to the value of *E* in the method's pre-state, you write **old**(*E*). For example, the **modifies** clause of the following method says that the array's elements may be modified, but the postcondition restricts those modifications to ones that increment *a*[4], do not increment *a*[6], and leave *a*[8] unchanged:

```
method UpdateElements(a: array<int>)
  requires a.Length == 10
  modifies a
  ensures old(a[4]) < a[4]
  ensures a[6] <= old(a[6])
  ensures a[8] == old(a[8])
{
  a[4], a[8] := a[4] + 3, a[8] + 1;
  a[7], a[8] := 516, a[8] - 1;
}
```

To be more precise than I was above, **old** affects only the heap dereferences in its argument. For example, in

```
method OldVsParameters(a: array<int>, i: int) returns (y: int)
  requires 0 <= i < a.Length
  modifies a
  ensures old(a[i] + y) == 25
```

the variables `a`, `i`, and `y` are unaffected by the enclosing `old(_)`. That is, in-parameters `a` and `i` denote their values on entry to the method and out-parameter `y` denotes its value on exit from the method, just as if `old` were not used. Only the heap dereference denoted by the square brackets is interpreted in the pre-state of the method.

A common mistake is to use `old` with an unintended argument. The postcondition of the following method illustrates:

```
method Increment(a: array<int>, i: int)
  requires 0 <= i < a.Length
  modifies a
  ensures a[i] == old(a)[i] + 1 // common mistake
{
  a[i] := a[i] + 1; // error: postcondition violation
}
```

Since there are no heap dereferences in the argument to `old`, the attempted expression `old(a)` would mean the same thing as just `a`.

Exercise 14.2.

Move the parentheses of `old` in the specification of `Increment`, so that its body will establish the postcondition.

Exercise 14.3.

Write an implementation for `OldVsParameters` that satisfies the specification.

By the way, only the method's **ensures** clause is a two-state predicate. The **requires**, **modifies**, and **decreases** clauses are always interpreted in the method's pre-state.

14.0.2. New arrays

The purpose of a **modifies** clause for a method `M` is to make it clear which pieces of the caller's state may be changed by

M. This does not apply to arrays that are allocated by M, since a caller has no access to such arrays before calling M.

A method is allowed to allocate a new array and change the elements of that array without mentioning this array in the **modifies** clause.

For example,

```
method NewArray() returns (a: array<int>)
  ensures a.Length == 20
{
  a := new int[20];
  var b := new int[30];
  a[6] := 216;
  b[7] := 343;
}
```

is correct, even without a **modifies** clause.

14.0.3. Fresh arrays

Consider a caller of NewArray() above:

```
method Caller() {
  var a := NewArray();
  a[8] := 512; // error: modification of a's elements not
allowed
}
```

In order for caller to modify the elements of a, it must prove that a is in caller's **modifies** clause or prove that a has been allocated since the time of entry to caller. We could only hope to do the latter. To do that, we must strengthen the specification of NewArray to promise that the array returned by NewArray is allocated inside NewArray. Knowing that, it is then clear that the array is allocated on behalf of caller.

Such a specification is written using the **fresh** predicate, as follows:

```
method NewArray() returns (a: array<int>)
  ensures fresh(a) && a.Length == 20
```

With this specification of `NewArray`, method `Caller` verifies.

14.0.4. Reads clauses

A function cannot modify anything, so it does not have a write frame. However, a function has a *read frame*, which announces the function's dependencies on the heap and is specified using a **reads** clause. This dependency information is used to determine if various mutations of the heap affect the value of the function (which is especially important if the body of the function is not available or if the function is recursive).

If a function accesses the elements of an array `a`, its specification must include **reads a**.

The following function depends on the values of the given array `a`, so it must include the **reads a** specification:

```
predicate IsZeroArray(a: array<int>, lo: int, hi: int)  
  requires 0 <= lo <= hi <= a.Length  
  reads a  
  decreases hi - lo  
{  
  lo == hi || (a[lo] == 0 && IsZeroArray(a, lo + 1, hi))  
}
```

Note that **reads** clauses express which parts of the mutable heap a function depends on. Thus, the elements of a sequence and the properties of a datatype value can be inspected by a function without the need for a **reads** clause (which also explains why we never saw any **reads** clauses in Parts 0 and 1 of the book). For example, the sequence version of `IsZeroArray` does not need a **reads** clause:

```
predicate IsZeroSeq(a: seq<int>, lo: int, hi: int)  
  requires 0 <= lo <= hi <= |a|  
  decreases hi - lo  
{  
  lo == hi || (a[lo] == 0 && IsZeroSeq(a, lo + 1, hi))  
}
```

```
}
```

14.1. Basic Array Modification

Let's consider some illustrative examples that modify arrays in simple ways.

14.1.0. Initializing an array

We'll write a method that sets all array elements to a given value. Here is the method specification:

```
method InitArray<T>(a: array<T>, d: T)
  modifies a
  ensures forall i :: 0 <= i < a.Length ==> a[i] == d
```

To implement this method, we specify a loop, obtaining the invariant by replacing the constant `a.Length` in the postcondition by the loop index for our loop:

```
{
  var n := 0;
  while n != a.Length
    invariant 0 <= n <= a.Length
    invariant forall i :: 0 <= i < n ==> a[i] == d
}
```

It's been a while since we practiced working backward from the loop invariant over the loop-index update `n := n + 1` (see Chapters 2, 11, and 12). Let's do it here for this simple program.

```
{{ (forall i :: 0 <= i < n ==> a[i] == d) && a[n] == d }}
{{ forall i :: 0 <= i < n + 1 ==> a[i] == d }}
n := n + 1
{{ forall i :: 0 <= i < n ==> a[i] == d }}
```

This calculation immediately tells us that we need to set `a[n]` to `d` before the update to `n`. So, here is the loop body that completes our method implementation:

```
{
  a[n] := d;
  n := n + 1;
```

```
}
```

14.1.1. Initializing a matrix

Let's repeat the `InitArray` method, but this time for a matrix:

```
method InitMatrix<T>(a: array2<T>, d: T)
  modifies a
  ensures forall i, j ::
    0 <= i < a.Length0 && 0 <= j < a.Length1 ==> a[i,j] == d
```

The postcondition quantifies over both dimensions of indices of `a`. This suggests that we need two loop indices. We've seen many programs like that in the previous chapter, but in those examples, we were lucky enough to get away with a single loop that, in some way, traversed through a path of index values. Here, there are no shortcuts—we must go through each of the quadratically many index pairs. That suggests using two loops, one nested inside the other.

Our outer loop will have the effect of initializing a whole row with each iteration. In other words, after `m` iterations, `m` rows will be initialized. This is reflected in the loop invariant, which we obtain by replacing the constant `a.Length0` in the postcondition with the loop index `m` (Loop Design Technique 12.0):

```
{
  var m := 0;
  while m != a.Length0
    invariant 0 <= m <= a.Length0
    invariant forall i, j ::
      0 <= i < m && 0 <= j < a.Length1 ==> a[i,j] == d
}
```

For this program, too, let's work backward from the loop invariant over the update of the loop index:

```
{{ (forall i, j :: 0 <= i < m && 0 <= j < a.Length1 ==> a[i,j]
== d) &&
  (forall j :: 0 <= j < a.Length1 ==> a[m,j] == d) }}
```

// apply One-Point Rule on second quantifier

```

{{ (forall i, j :: 0 <= i < m && 0 <= j < a.Length1 ==> a[i,j]
== d) &&
  (forall i, j :: i == m && 0 <= j < a.Length1 ==> a[i,j] == d)
}}
// apply Range Split
{{ forall i, j :: 0 <= i < m + 1 && 0 <= j < a.Length1 ==>
a[i,j] == d }}
m := m + 1
{{ forall i, j :: 0 <= i < m && 0 <= j < a.Length1 ==> a[i,j]
== d }}

```

The top annotation shows two quantifiers, which thus become the postcondition of our inner loop. It's easy to forget that the inner loop needs to establish *both* of these quantifiers, not just the `forall j` quantifier. If we only mention row `m` in the invariant of the inner loop, then that loop specification would allow any modification whatsoever of the other rows.

Heap-related loop frames

The root cause of the issue I just mentioned is loop frames, that is, the (usually implicit) part of the loop specification that says what the loop is allowed to modify. I mentioned loop frames before in the context of local variables (Sections 11.0.5 and 11.1.3), but here we also need to consider what a loop specification says about heap-allocated storage, like the elements of arrays.

Loops have the notion of a **modifies** clause for heap-allocated storage, just like methods do. However, it is rare that the **modifies** clause of a loop is any different than the **modifies** clause of the enclosing method (or enclosing loop, in case of nested loops). Therefore, unless a loop mentions an explicit **modifies** clause, the default **modifies** clause of the loop is that of the enclosing method (or enclosing loop).

The inner loop

Back to our matrix initialization method. Our working-backward calculation above concluded that the inner loop

must establish the two quantifiers shown in the top annotation. The first quantification (**forall** *i*, *j*) already holds on entry to the inner loop, so all the inner loop needs to do is maintain this condition. We'll apply the *always* Loop Design Technique 13.1 to it. For the second quantification (**forall** *j*), we'll replace the constant `a.Length1` with a loop index *n*. What we get is:

```
{
  var n := 0;
  while n != a.Length1
    invariant 0 <= n <= a.Length1
    invariant forall i, j ::
      0 <= i < m && 0 <= j < a.Length1 ==> a[i,j] == d
    invariant forall j :: 0 <= j < n ==> a[m,j] == d
```

It's easy to fill in the rest of the method implementation:

```
{
  a[m,n] := d;
  n := n + 1;
}
m := m + 1;
}
```

14.1.2. Incrementing the values in an array

I'll show two more examples at this level of difficulty, to make sure you master this basic technique. In this next example, the method increments every element of a given array by 1. Here is the method specification:

```
method IncrementArray(a: array<int>)
  modifies a
  ensures forall i :: 0 <= i < a.Length ==> a[i] == old(a[i]) + 1
```

Note that the **old** expression encloses the expression `a[i]`, which means it denotes element *i* of array *a* on entry to the method. Remember from Section 14.0.1, an expression like `old(a)[old(i)]` denotes a different value—since **old** has no effect on parameters and bound variables, it is the same as just `a[i]`. It is the square brackets that dereference the

heap, so it is the square brackets that we want to apply **old** to, and we do that by writing **old(a[i])**.

If we think of replacing the constant `a.Length` with a loop index `n`, we would write the loop as follows:

```
{
  var n := 0;
  while n != a.Length
    invariant 0 <= n <= a.Length
    invariant forall i :: 0 <= i < n ==> a[i] == old(a[i]) + 1
  {
    a[n] := a[n] + 1;
    n := n + 1;
  } // error: second loop invariant not maintained by loop body
}
```

However, in this form, we cannot maintain the invariant. The loop body sets `a[n]` to `a[n] + 1`, and in order to maintain the invariant, we need to know

`a[n] + 1 == old(a[n]) + 1`

The loop frame is the same as for the enclosing method, so as far as the loop specification is concerned, the elements of `a` can have any values allowed by the invariant. The invariant above only talks about the array elements `a[..n]`, so there is no information about the relation between the current value of `a[n]` and the value of `a[n]` on entry to the loop.

Debugging the verification

To realize this is the problem, you must remember to look only at the loop specification and not be tempted to draw any other conclusions about what states iterations of the loop body may reach (Chapter 11). If you still don't see what the problem is, I suggest you debug the problem in the following way.

Start by adding an **assert** statement with the violated invariant at the place where it is supposed to hold. Your loop body will now look like

```

    a[n] := a[n] + 1;
    n := n + 1;
    assert forall i :: 0 <= i < n ==> a[i] == old(a[i]) + 1; //
error

```

This should also have had the effect of trading the error message on the invariant for an error message on the assertion. That's what we'd expect, because if we could prove this condition here, then the invariant would be fine, too.

Next, move—or copy, if you wish—this assertion to before the assignment to n , and in doing so, replace n by $n + 1$, as when you're computing the weakest precondition. You'll then have

```

    a[n] := a[n] + 1;
    assert forall i :: 0 <= i < n + 1 ==> a[i] == old(a[i]) + 1;
// error
    n := n + 1;

```

This assertion before the assignment to n is the same as the assertion we had after the assignment to n a moment ago. Therefore, the verifier should now be complaining about this assertion and nothing else. (If you find otherwise, then you should focus your debugging on why that is.)

Next, let's manually split this quantifier. The range is $0 \leq i < n + 1$, so we split it into the disjunction of $0 \leq i < n$ and $i == n$. We apply the One-Point Rule to the latter. (See Section 13.1.5 or Appendix B if you need a reminder about these quantifier rules.) Showing the result as two additional assertions, we now have

```

    a[n] := a[n] + 1;
    assert forall i :: 0 <= i < n ==> a[i] == old(a[i]) + 1;
    assert a[n] == old(a[n]) + 1; // error
    assert forall i :: 0 <= i < n + 1 ==> a[i] == old(a[i]) + 1;
    n := n + 1;

```

The verifier now shows the middle assertion as the one it cannot prove. Since the preceding assignment statement just gave a new value to $a[n]$, we expect $a[n]$ in the violated assertion to have the value that the right-hand side of the

assignment (that is, $a[n] + 1$) had just before the assignment. Let's add an assertion to that effect:

```
assert a[n] + 1 == old(a[n]) + 1; // error
a[n] := a[n] + 1;
assert forall i :: 0 <= i < n ==> a[i] == old(a[i]) + 1;
assert a[n] == old(a[n]) + 1;
assert forall i :: 0 <= i < n + 1 ==> a[i] == old(a[i]) + 1;
n := n + 1;
```

The verifier now complains about the first of the assertions, but about nothing else. This tells us that if we can only establish that condition at the point of the assertion, then the verification goes through.

I think the failing assertion we ended up with clearly points out the missing information, namely the need to know $a[n] == \text{old}(a[n])$ at the beginning of the loop. Now, we think about the whole loop for a minute and realize that array elements at index n and higher have not been changed by the loop. To add this fact to the verification process, we incorporate it into the invariant:

```
invariant forall i :: n <= i < a.Length ==> a[i] ==
old(a[i])
```

This causes all the verifier's complaints to go away. In other words, the new invariant is checked to hold on entry to the loop, all of the assertions are checked to hold, and all the invariants can now be proved to hold at the end of the loop body. Since the **assert** statements were added just to help us diagnose the situation, they have served their purpose, so we can remove them.

Exercise 14.4.

Specify and implement a method that increases every element of a matrix by 1.

14.1.3. Copying an array

Let's copy the elements of one array into another.

```

method CopyArray(src: array, dst: array)
  requires src.Length == dst.Length
  modifies dst
  ensures forall i :: 0 <= i < src.Length ==> dst[i] ==
old(src[i])

```

This method takes two arrays, one of which is listed in the **modifies** clause. Clearly, this gives the method license to modify the elements of `dst`. More precisely, the elements of the array referenced by `dst` can be modified. If `src` happens to reference the same array as `dst`—that is, if `src == dst`—then it is also true that the method can modify the elements of the array reference by `src`. As I remarked before (in Section 13.0.2), a **modifies** clause says which arrays are allowed to be modified, but does not care what expression you use to reference those arrays.

In the event that `src` and `dst` reference the same array, a postcondition like

```

ensures forall i :: 0 <= i < src.Length ==> dst[i] == src[i]

```

would trivially hold even if the method completely altered the contents of the array, because this postcondition would simply say that the post-value of each element (of the one array we're dealing with) is equal to itself. That's why we write the specification to say **old**(src[i]). (Alternatively, we could have added the precondition `src != dst`.)

Having settled on that method specification, the loop specification and loop body follow patterns we've seen before.

Note that, akin to the situation we debugged in Section 14.1.2, we need to be sure to include the loop invariant that says the elements of `src` are unchanged in order to handle the situation where `src == dst`.

```

{
  var n := 0;
  while n != src.Length
    invariant 0 <= n <= src.Length
    invariant forall i :: 0 <= i < n ==> dst[i] == old(src[i])

```

```

    invariant forall i :: 0 <= i < src.Length ==> src[i] ==
old(src[i])
{
    dst[n] := src[n];
    n := n + 1;
}
}

```

Practice writing array modifications using loops in the following exercises:

Exercise 14.5.

Add `src != dst` as a precondition. Then, simplify the postcondition and invariants as much as possible (without changing the meaning of the postcondition).

Exercise 14.6.

Change `old(src[i])` to `src[i]` in the postcondition of `CopyArray`, as hypothetically considered above. Then, change the implementation of `CopyArray` so that it makes all elements of `src` the same if `src == dst`.

Exercise 14.7.

Specify and implement a method that copies the elements of one matrix to another of the same dimensions. Allow the source and destination matrices to be the same and make sure your specification says the right thing in that case. (Write a little test harness if you're uncertain.)

Exercise 14.8.

Specify and implement a method

```
method DoubleArray(src: array<int>, dst: array<int>)
```

that sets `dst[i]` to `2 * src[i]` for each `i`. Assume the given arrays have the same lengths, and allow the possibility that they reference the same array.

Exercise 14.9.

Specify and implement a method that reverses the elements of a given array.

Exercise 14.10.

Specify and implement a method that takes a square matrix and transposes its elements.

Exercise 14.11.

Specify and implement a method that in a given array rotates the elements “left”. That is, what used to be stored in `a[(i + 1) % a.Length]` gets stored in `a[i]`.

Exercise 14.12.

Specify and implement a method that in a given array rotates the elements “right”. That is, what used to be stored in `a[(i - 1) % a.Length]` gets stored in `a[i]`.

14.1.4. Loop-less array operations

Loops like the ones I showed in this section illustrate how you work with quantified loop invariants, and they provide useful practice for doing more complicated loops. Beyond that, loops like these are quite tedious—they are conceptually simple and yet require loop invariants that come across as far more complicated than the loop bodies that they are describing.

If you reflect on it, you realize that loops are the wrong programming construct for doing these simple, uniform operations. Some programming languages don't have anything better to offer, but others do. I'll show you two such constructs that are found in Dafny.

Array constructor

When allocating an array, you can give a function that specifies the initial elements, like

```
a := new int[25](F);
```

where F is a function from the indices of the new array (from 0 to 25 in this example) to the desired element values of the new array (of type `int` in this example).

Most often, the function F is given as an *inline function*, often known as a *lambda expression*. It has the form $x \Rightarrow E$ where x is the formal parameter and expression E is the body of the function. For example,

```
a := new int[25](i => i * i);
```

allocates an array of length 25 whose elements are initialized to the first 25 squares. Another example is the commonly occurring

```
a := new int[n](_ => 0);
```

which allocates an array of length n where each element is initialized to 0. As the example shows, if the formal parameter of the inline function is not used in the body of the inline function, you can omit a specific name and instead write `_`.

For multi-dimensional arrays, you use a function with more than one parameter. Using an inline function for that purpose, here is an example that initializes all elements of a new 50 by 50 matrix with d along the diagonal and 0 everywhere else:

```
m := new int[50, 50]((i, j) => if i == j then d else 0);
```

Note that this inline function puts the list of parameters in parentheses—in the special case of one parameter, the parentheses can be omitted, as I showed in the examples above.

Inline functions with specifications To allocate an array as a copy of another, you would write something like:

```
b := new T[a.Length](i => a[i]); // error in accessing a
```

However, the verifier complains about this, because it checks the well-formedness of the inline function without regard to where the function is being used. In particular,

for $i \Rightarrow a[i]$ to be a well-formed function, we need to both make sure i is in range and that a is allowed by the frame of the function. If we declared a named function, this would be clear to us (we've seen function preconditions throughout the book and I introduced read frames in Section 14.0.4). Inline functions can also have **requires** and **reads** clauses, and we need to use both to do the copying example. Here is the correct form, which the verifier accepts:

```
b := new T[a.Length](i requires 0 <= i < a.Length reads a =>
                        a[i]);
```

This is long, but it still saves us the hassle of writing a loop and inventing a loop invariant.

Sequence constructors There is a similar construct for constructing a sequence. The expression

```
seq(500, i => i % 2 == 0)
```

denotes a sequence of 500 booleans, where the elements behind even indices are **true** and the elements behind odd indices are **false**.

Remember that sequences are values whereas arrays are references to mutable elements. Accordingly, the sequence constructor is an expression, not an allocation statement.

Aggregate statements

Dafny has an *aggregate statement* that can be used to update array elements. You can think of the aggregate statement as a generalization of simultaneous assignment. So, whereas

```
a[5], a[7] := a[5] + 25, a[7] + 49;
```

updates two array elements, the aggregate statement

```
forall i | 0 <= i < a.Length {
    a[i] := a[i] + i * i;
}
```

updates all array elements in a similar way.

Despite the syntactic similarity with the **for** and **foreach** statements in languages like Java or C#, the crucial point about the **forall** statement is that *it is not a loop*. This has two consequences.

One consequence is that the body of the **forall** statement is performed simultaneously for all values of the bound variable. Certain computations are easy to formulate in this way. For example,

```
forall i | 0 <= i < a.Length && i % 2 == 0 {  
    a[i] := a[i] + 1;  
}
```

increments even-indexed elements by 1, the `IncrementArray` method in Section 14.1.2 can be implemented as just

```
forall i | 0 <= i < a.Length {  
    a[i] := a[i] + 1;  
}
```

and

```
forall i | 0 <= i < a.Length {  
    a[i] := a[(i + 1) % a.Length];  
}
```

rotates all the elements of `a` to the left (which Exercise 14.11 asked you to do with a loop). However, it is not possible to do something like sum the elements of an array, since this cannot be done as one simultaneous operation over all indices.

The other consequence of the **forall** statement not being a loop is that reasoning about it does not require a loop invariant! For example, you can say goodbye to the long and boring loop invariants of the nested loops in Section 14.1.1 and instead simply write

```
forall i, j | 0 <= i < a.Length0 && 0 <= j < a.Length1 {  
    a[i,j] := d;  
}
```

To make compilation work out without too many complications, there are some restrictions on the **forall** statement. The most noticeable of these is that the body must contain just one assignment statement.

In the future, I will use a **forall** statement when possible, but we'll see plenty of programs where iteration is really needed. For those, you'll be glad to have mastered the technique of writing loop invariants on the simpler programs first.

Exercise 14.13.

Do Exercise 14.7 using a **forall** statement.

Exercise 14.14.

Do Exercise 14.8 using a **forall** statement.

Exercise 14.15.

Do Exercise 14.9 using a **forall** statement.

Exercise 14.16.

Do Exercise 14.10 using a **forall** statement.

14.2. Summary

In this chapter, I introduced *read and write frames*, which specify which parts of the heap a function may depend on and which parts of the heap a method may modify, respectively. Read frames are declared with **reads** clauses and write frames with **modifies** clauses, and both of them denote a set of references into the heap.

I also introduced some features that are useful in specifications when array elements or other object fields are modified in the heap. **old**(E) denotes the value of E in the prestate of the method, **fresh**(E) says that the reference (or set of references) E has been allocated since entry to the enclosing method.

It is important to understand how to specify and write loops that set array elements in various ways. I gave a number of examples that initialize arrays or perform simple updates on arrays. These highlight the importance of specifying what has changed and what hasn't changed in the heap.

Finally, I showed some language features that can eliminate the need to write loops in the first place.

In the next chapter, we'll consider some programs that manipulate arrays in more interesting ways.

Notes

The design of the pioneering programming language Euclid [74] targeted the construction of verifiable programs. It featured declarations of pre- and postcondition, inline assertions, and module invariants. In order for a procedure to access a variable declared outside the procedure, the procedure had to explicitly *import* the variable. Such an import clause also indicated if the procedure might modify the variable or only read it.

Chapter 15

In-situ Sorting



In Chapter 8, I covered specifications and programs for sorting inductive lists. Since lists are immutable, those programs necessarily returned sorted versions of their inputs, without modifying the input. In this chapter, we'll consider programs that reorder the elements of a given array to make it sorted. That is, the sorting is done *in situ*, Latin for “in place”.

15.0. Dutch National Flag

I'll start us with a problem of sorting an array with three kinds of values. The values are the three colors red, white, and blue, and the order in which we'll sort them is what has given this problem its name, the Dutch National Flag (first red, then white, then blue).

We'll start by introducing the colors and a way to compare them.

```
datatype Color = Red | White | Blue
```

```
ghost predicate Below(c: Color, d: Color) {  
  c == Red || c == d || d == Blue  
}
```

Since we are going to sort the array in place, the only parameter to our method is a reference to the array. The write frame says the method can modify the contents of the array. As we saw in Section 8.0, the postcondition has two parts. One part says the elements are sorted on exit, and the other part says that the multiset of elements in the array is the same before and after the method. Using the notation we saw in Section 13.4, we can write the latter using a conversion from the array elements to a sequence to a multiset:

```
method DutchFlag(a: array<Color>)  
  modifies a  
  ensures forall i, j :: 0 <= i < j < a.Length ==> Below(a[i],  
a[j])  
  ensures multiset(a[..]) == old(multiset(a[..]))
```

The idea behind the algorithm is to structure the given array into four adjacent segments. The first segment will contain only red elements, the next segment only white elements, and the last segment only blue elements. The remaining segment (which I'll place between the white and blue segments, though another choice would be to place it between the red and white segments) will contain the elements we haven't yet sorted into its proper segment.

To keep track of the four segments, we'll need three markers. These markers will play the role of our loop indices, and I will name them r , w , and b . The following diagram summarizes the ideas so far. In fact, the diagram very much expresses the invariant of the loop we'll write.



Here's the loop specification:

```
{
  var r, w, b := 0, 0, a.Length;
  while w < b
    invariant 0 <= r <= w <= b <= a.Length
    invariant forall i :: 0 <= i < r ==> a[i] == Red
    invariant forall i :: r <= i < w ==> a[i] == White
    invariant forall i :: b <= i < a.Length ==> a[i] == Blue
    invariant multiset(a[..]) == old(multiset(a[..]))
}
```

The initial assignments of r, w, b make the three colored segments empty, leaving all array elements in the unsorted segment. The loop guard says to continue iterating as long as there are elements in that unsorted segment. When the unsorted segment is empty, the invariant and the negation of the guard imply $w == b$, which says the array consists solely of the three colored segments in the correct order. The final invariant uses the *always* Loop Design Technique 13.1.

By the way, note the consistent use of half-open intervals in the ranges of the quantifiers. This is a consequence of thinking of r, w, b as delineating the segments, and it is also the way we usually represent ranges in computer science. The ranges compose nicely, because two segments are consecutive when the upper limit of one segment is the

lower bound of the next (see also Sidebar 13.0). Indeed, when $w == b$, you can directly see that the three quantifiers account for all the array elements.

One task left: implement the loop body. Each iteration will sort one more element, so we need to inspect one of the elements in the unsorted segment. The obvious candidates are $a[w]$ and $a[b-1]$, since those elements are adjacent to a sorted region. We'll use $a[w]$, which gives us the following structure of the loop body:

```
{
  match a[w]
  case Red => ?
  case White => ?
  case Blue => ?
}
```

To fill in the three question marks, you may find the diagram representation of the invariant from above useful.

The simplest case is when $a[w]$ is white. Then, all we need to do is include it in the white region, which we do by incrementing w :

```
case White =>
  w := w + 1;
```

If $a[w]$ is red, we want to move it into the red region. But the white region starts where the red region ends, so we'll need to move the first white element to the end of the white region:

```
case Red =>
  a[r], a[w] := a[w], a[r];
  r, w := r + 1, w + 1;
```

In the event that the white region is empty—that is, if $r == w$ —the swap of $a[r]$ and $a[w]$ will amount not moving any elements, but that is still fine.

Finally, if $a[w]$ is blue, we move it to $a[b-1]$ and decrement b . This will put the blue element into the blue region and make progress toward termination. However, if we only did

this much, then we would break the last invariant, because we would duplicate `a[w]` and clobber the element previously stored in `a[b-1]`. So, it is important to swap `a[w]` and `a[b-1]`:

```
case Blue =>
  a[w], a[b-1] := a[b-1], a[w];
  b := b - 1;
```

This completes the program.

Exercise 15.0.

Change the invariant to place the unsorted segment between the red and white segments. Then, re-implement the initialization and loop body accordingly.

Exercise 15.1.

Specify a method that sorts an array of booleans. Consider **false** to be ordered before **true**. Implement the method using a loop that swaps array elements (akin to what we did for the Dutch National Flag).

Exercise 15.2.

Like many representations of sorting algorithms, the `DutchFlag` method above is a simplification of what you would actually do. I represented each element as a color. More typically, the array elements are more complex pieces of data and the color is just the key that we're sorting the array by. Define a predicate `CBelow` that compares two values based on their color:

```
ghost predicate CBelow<T>(color: T -> Color, x: T, y: T)
```

and implement a method that sorts an array based the color of its elements.

```
method DutchFlagKey<T>(a: array<T>, color: T -> Color)
  modifies a
  ensures forall i, j :: 0 <= i < j < a.Length ==>
    CBelow(color, a[i], a[j])
  ensures multiset(a[..]) == old(multiset(a[..]))
```

Exercise 15.3.

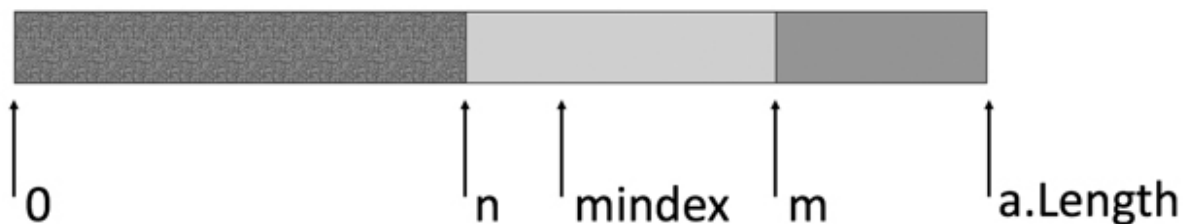
(Advanced.) Sorting algorithms typically use swapping to rearrange the order of the given elements. But sorting an array of three colors (without any other payloads) can be done without swapping: First, scan the elements of the given array and count the number of occurrences of each color. Second, assign the output according to those three counts, writing the appropriate number of reds, whites, and blues, respectively. Implement and verify the Dutch National Flag method in that way.

15.1. Selection Sort

Selection Sort is a sorting algorithm that repeatedly finds the minimum among the unsorted elements and then appends it to the segment of sorted elements. Its specification is like that of `DutchFlag` in the previous section, but here I'm using integers instead of colors:

```
method SelectionSort(a: array<int>)  
  modifies a  
  ensures forall i, j :: 0 <= i < j < a.Length ==> a[i] <= a[j]  
  ensures multiset(a[..]) == old(multiset(a[..]))
```

We'll have two nested loops. With n as its loop index, an invariant of the outer loop is that the first n elements of the array are sorted. The following diagram illustrates:



The inner loop scans the remainder of the array to find the minimum. As we've seen before, we can specify the minimum as being below all the elements examined and as being one of the elements to be examined. We'll need to

keep track of where in the array the minimum is, so we'll use a variable `minindex` for this purpose. By making sure `minindex` is an index into the unsorted part of the array, it follows that `a[minindex]` is one of the elements to be examined. The diagram above sketches this situation, where `m` is the loop index of the inner loop.

That's the main idea. Let's implement the method.

Our starting point is straightforward. We'll use a loop index `n`, we'll turn the first postcondition into an invariant by replacing the constant `a.Length` with variable `n`, and we'll turn the other postcondition into an “always” invariant:

```
{
  var n := 0;
  while n != a.Length
    invariant 0 <= n <= a.Length
    invariant forall i, j :: 0 <= i < j < n ==> a[i] <= a[j]
    invariant multiset(a[..]) == old(multiset(a[..]))
}
```

Next, we'll work on the inner loop. We've written many loops like it before (including the minimum program in Section 13.3), so we may immediately come up with:

```
var minindex, m := n, n;
while m != a.Length
  invariant n <= m <= a.Length && n <= minindex < a.Length
  invariant forall i :: n <= i < m ==> a[minindex] <= a[i]
  {
    if a[m] < a[minindex] {
      minindex := m;
    }
    m := m + 1;
  }
```

We can do a little better. If we make sure `minindex` is less than `m`, then we can write the first invariant as

```
invariant n <= minindex < m <= a.Length
```

This gives us an error, complaining that `minindex < m` does not hold on entry to the loop. But we can fix that by starting `m` off at `n + 1`, which is known not to exceed `a.Length`.


```
var mindex, m := n, n + 1;
```

Indeed, there is no reason to iterate the inner loop for $m == n$, since we initialize `mindex` to n , which clearly is the index of the minimum element in $a[n..n+1]$.

Great. After the inner loop, `mindex` tells us where the next smallest element is. So, all we need to do is swap $a[mindex]$ into place:

```
a[n], a[mindex] := a[mindex], a[n];  
n := n + 1;
```

Something is wrong! The verifier complains that the universally quantified loop invariant is not maintained by the loop. On entry to the loop body, that invariant told us the first n elements are sorted and now we are saying the first $n + 1$ elements are sorted. What's wrong with that?

One possible culprit is the inner loop. We've seen before (when initializing a matrix in Section 14.1.1) that an inner loop sometimes needs to include some loop invariants of the outer loop. But not here, because our inner loop does not modify the array or anything in the heap; the verifier detects this syntactically, and therefore it is automatically known that the sorted segment of the array remains sorted.

Exercise 15.4.

An alternative way to write the inner loop is to swap any new minimum into $a[n]$ as soon as it's detected. This means the inner loop will modify the array, so then the inner loop does need some invariants from the outer loop. After we fix the situation with the current algorithm, change the inner loop as I've just described and verify it. (Arguably, this alternative program is worse, but as an exercise, it will let you get more good practice with invariants.)

Alright, then what? We can check if the verifier still thinks the first n elements are sorted, by adding the

following assertion immediately after the swap and *before* the increment of n :

```
assert forall i, j :: 0 <= i < j < n ==> a[i] <= a[j];
```

The verifier confirms this assertion. This can only mean one thing: the verifier is not convinced that our new $a[n]$ is ordered after $a[..n]$.

When you work with program proofs, situations like this arise all the time. We try to write down our design decisions as invariants and these guide our program design. Inevitably, some part of the design remains in our head. Either we thought about the situation correctly, but didn't think to write down the conditions as invariants, or we missed some part of the design that may or may not be correct. Whichever the case may be, the way to make a convincing correctness argument is to formulate what is true as an invariant.

What's missing in our case is the property that the elements in the sorted segment of the array are not just sorted among themselves, but they are also in their final position. Stated differently, the sorted elements are all below the unsorted elements. This is a crucial property of Selection Sort. Let's write it down as an invariant of the outer loop:

```
invariant forall i, j :: 0 <= i < n <= j < a.Length ==>  
a[i] <= a[j]
```

This property comes in handy in other sorting algorithms, too, so we might as well define a predicate for it:

```
ghost predicate SplitPoint(a: array<int>, n: int)  
  requires 0 <= n <= a.Length  
  reads a  
{  
  forall i, j :: 0 <= i < n <= j < a.Length ==> a[i] <= a[j]  
}
```

If you choose to introduce this predicate, you can formulate the last invariant as:

invariant SplitPoint(a, n)

With this invariant, stated directly as a quantifier or via the SplitPoint predicate, we have verified Selection Sort.

15.2. Quicksort

The final sorting routine in this book is Quicksort. It performs well in practice, but is noticeably more complicated to verify than the other sorting routines we have considered. The difficulty comes down to specifying what parts of the array are modified and what relationships continue to hold in the face of those modifications.

15.2.0. Method specification

Things do start easy, for the specification of Quicksort is the same as that of SelectionSort:

```
method Quicksort(a: array<int>)  
  modifies a  
  ensures forall i, j :: 0 <= i < j < a.Length ==> a[i] <= a[j]  
  ensures multiset(a[..]) == old(multiset(a[..]))
```

Quicksort divides up its input into smaller segments and sorts those segments recursively. To specify bounds of the segment to be sorted, we'll use an auxiliary method, and the main Quicksort method calls the auxiliary method with 0 and a.Length as the bounds:

```
{  
  QuicksortAux(a, 0, a.Length);  
}
```

We'll have to work at the specification of QuicksortAux for a while. Here are parts of the specification that it's clear we'll need:

```
method QuicksortAux(a: array<int>, lo: int, hi: int)  
  requires 0 <= lo <= hi <= a.Length  
  modifies a  
  ensures forall i, j :: lo <= i < j < hi ==> a[i] <= a[j]
```

decreases hi - lo

We will also need the postcondition that says the multiset of array elements is unchanged, but we'll also need more precise information about which array elements are changed at all. To express `a[i] == old(a[i])` for the indices `i` outside the range from `lo` to `hi`, we'll use a quantifier. We'll have to write this quantifier, and also the multiset property, several times, so let's define a predicate that we can reuse where needed.

15.2.1. Two-state predicates

This predicate is different than any other we've seen so far, because we want it to relate the current state with the pre-state of a method. For this purpose, Dafny provides *two-state predicates*. These are able to read two heaps, the current heap and some heap from the past. The prior heap is accessed with `old` expressions, just as in method bodies and postconditions. Calls to a two-state predicate in effect pass in the prior heap. That's all we need to know about two-state predicates to use them in Quicksort.

Here is our two-state predicate:

```
twostate predicate SwapFrame(a: array<int>, lo: int, hi: int)
  requires 0 <= lo <= hi <= a.Length
  reads a
{
  (forall i :: 0 <= i < lo || hi <= i < a.Length ==>
    a[i] == old(a[i])) &&
  multiset(a[..]) == old(multiset(a[..]))
}
```

Our first use of this predicate is as a postcondition in the specification of `QuicksortAux`:

```
ensures SwapFrame(a, lo, hi)
```

15.2.2. The core algorithm

The idea of Quicksort is simple. Pick an element—known as the *pivot*—among those to be sorted. In terms of the Dutch National Flag, think of the pivot as white, the elements smaller than the pivot as red, and those larger than the pivot as blue. Then, sort according to color, like in the Dutch National Flag algorithm. In Quicksort, this subroutine is usually named `Partition`. Method `Partition` returns the index to the pivot. Then, Quicksort recursively sorts the elements to the left of the pivot and the elements to the right of the pivot.

Here is the body of `QuicksortAux`, which performs the steps I just described:

```
{
  if 2 <= hi - lo {
    var p := Partition(a, lo, hi);
    QuicksortAux(a, lo, p);
    QuicksortAux(a, p + 1, hi);
  }
}
```

Alas, even after specifying `Partition` (which we'll do below), `QuickSortAux` does not verify as given. The problem is that our `SwapFrame` predicate does not directly give us the information we need about the relation of the elements. We could try to prove some lemmas about `SwapFrame`. Such lemmas would have to be *two-state lemmas*, which I will not cover in this book. Instead, we can use the `SplitPoint` predicate I introduced for Selection Sort at the end of Section 15.1.

15.2.3. Split points

As a final enhancement of the specification of `QuicksortAux`, we will require and ensure that `lo` and `hi` are split points. As a precondition, this says that elements `a[..lo]` are below `a[lo..hi]` and those, in turn, are below `a[hi..]`. As a

postcondition, this comes down to saying that the method does not swap any elements across split-point boundaries.

Here's what we add to the specification of `QuicksortAux`:

```
requires SplitPoint(a, lo) && SplitPoint(a, hi)
ensures SplitPoint(a, lo) && SplitPoint(a, hi)
```

15.2.4. Partition

It's time to specify and implement `Partition`. In spirit, it is essentially the Dutch National Flag algorithm, but the formulation is different in several ways from what we did in Section 15.0. In the specification, one difference is that we are now sorting only a segment of an array, another is that we don't actually have colors (but see Exercise 15.2), and a third is that `Partition` returns the index of the pivot. In the implementation, one difference is that we need to select the pivot, and another is that it will be easier to keep the pivot separate from the other elements while sorting the “red” and “blue” elements into place.

The specification is big:

```
method Partition(a: array<int>, lo: int, hi: int) returns (p: int)
  requires 0 <= lo < hi <= a.Length
  requires SplitPoint(a, lo) && SplitPoint(a, hi)
  modifies a
  ensures lo <= p < hi
  ensures forall i :: lo <= i < p ==> a[i] < a[p]
  ensures forall i :: p <= i < hi ==> a[p] <= a[i]
  ensures SplitPoint(a, lo) && SplitPoint(a, hi)
  ensures SwapFrame(a, lo, hi)
```

In addition to the `SplitPoint` and `SwapFrame` predicates like in `QuickSortAux`, this specification says that `p` returns as an index into `a[lo..hi]`. This is important in order for `QuicksortAux` to make progress with each of its recursive calls. The postcondition also says how the elements in the segment `a[lo..hi]` relate to the pivot element `a[p]`.

15.2.5. Implementing Partition

We have only one thing left to do for Quicksort, namely to implement method `Partition`. We'll do it in three steps.

As the first step, we select the pivot from among the elements to be partitioned. This selection affects performance and there are heuristics for selecting it well. For the sake of correctness, it does not matter which element we pick, so let's just pick `a[lo]`:

```
{  
  var pivot := a[lo];
```

As the second step, we use a loop to order the smaller-than-pivot elements before the pivot-or-larger elements in `a[lo+1..hi]`. Along the lines of the Dutch National Flag, we use two markers, `m` and `n`, so that `a[lo+1..m]` are smaller than pivot, `a[m..n]` have not yet been examined, and `a[n..hi]` are pivot or larger. Just like the enclosing method specification is big, so is the loop specification:

```
  var m, n := lo + 1, hi;  
  while m < n  
    invariant lo + 1 <= m <= n <= hi  
    invariant a[lo] == pivot  
    invariant forall i :: lo + 1 <= i < m ==> a[i] < pivot  
    invariant forall i :: n <= i < hi ==> pivot <= a[i]  
    invariant SplitPoint(a, lo) && SplitPoint(a, hi)  
    invariant SwapFrame(a, lo, hi)
```

The implementation of the loop examines the next element and either extends the lower segment or swaps it into the upper segment:

```
{  
  if a[m] < pivot {  
    m := m + 1;  
  } else {  
    a[m], a[n-1] := a[n-1], a[m];  
    n := n - 1;  
  }  
}
```

As the third step, we need to insert the pivot into the elements we partitioned. We swap it with the element that's adjacent to the upper region, and then we return the resulting index:

```
    a[lo], a[m - 1] := a[m - 1], a[lo];  
    return m - 1;  
}
```

I went through this algorithm with its big specifications quickly. Now that we've reached the end, I suggest that, as a way to continue to familiarize yourself with what just passed in front of your eyes, you go back and temporarily comment out parts of the specifications to see where verification fails.

Exercise 15.5.

One heuristic for selecting a pivot is to look at the leftmost, middle, and rightmost values of $a[lo..hi]$ and pick the median of these three values. Add code to do that in `Partition`.

Exercise 15.6.

Write a method that sorts an integer array by inserting all elements into the priority queue we developed in Chapter 10 and then successively removing the smallest remaining element.

15.3. Summary

In this chapter, I showed three algorithms that sort a given array by rearranging its elements. Along the way, we encountered read and write frames, and even two-state predicates. Throughout, we expressed properties of interest using universal quantifiers.

The main point of this and the previous few chapters has been to show you how to reason about algorithms using invariants. When you design a loop, it is crucial that you

pay attention to what can be said about the starting and ending state of each iteration. The loop invariant is where you record this design. The loop invariant lets you narrow your focus from the infinitely many iteration behaviors of a loop into the behaviors of a single, but arbitrary, iteration. If you make it clear what you expect to hold at the top of every iteration, you'll have a strong guide to the code you need to write in the loop body.

Invariants are not unique to loops. For example, in Chapter 10, we saw invariants for data structures, and in the chapters to come, we will see invariants for objects.

Here are some exercises for writing your own proofs of sorting-related algorithms. Each of these involves many details, so expect to spend a good amount of time debugging your programs and proofs.

Exercise 15.7.

The central operation in Merge Sort (which we saw in a functional setting in Section 8.2) is the merge operation. Using arrays, specify and implement a method

```
method Merge(a: array<int>, b: array<int>) returns (c:  
array<int>)
```

that takes two sorted arrays, *a* and *b*, and returns a new, sorted array *c* whose elements are those from *a* and *b*.

Exercise 15.8.

In Section 8.1, we implemented Insertion Sort for a list in the functional setting. For this exercise, specify and implement a method that performs Insertion Sort on a given integer array. Insertion Sort inserts each element, in turn, into a sorted prefix of the array.

You'll need two nested loops. For this exercise, do a swap in each iteration of the inner loop (which means you can use the *always* Loop Design Technique 13.1 for the “same elements” condition).

Exercise 15.9.

Implement Insertion Sort for a given integer array, but (unlike in Exercise 15.8) don't do a swap in the inner loop. Instead, just before the inner loop, save the value of the element to be inserted:

```
var x := a[i];
```

Then, in each iteration of the inner loop, just copy the next element into place:

```
while // ...  
  a[j] := a[j - 1];
```

After the inner loop, write the saved value into place:

```
a[j] := x;
```

These steps will have the effect of rotating the element into place, and it saves time because you only do “half of a swap” with each iteration.

Hint: To prove the “same elements” property, your inner loop will need an invariant like

```
invariant multiset(a[..][j := x]) ==  
multiset(old(a[..]))
```

where the sequence-update expression $s[j := x]$ denotes the sequence that is like s except with element j replaced by the value x .

Notes

Quicksort was invented by C. A. R. Hoare, who also introduced the triple notation we've been using in program reasoning. Years after its invention, it was one of the first sorting algorithms to be formally verified [51]. Now, more than half a century later, it is interesting to read that paper's concluding remarks on the prospect of mechanizing such a proof.

An even earlier formal proof was developed for Treesort 3 [88], a version of Heapsort due to Robert W. Floyd.

There are many formal proofs of sorting algorithms. For example, more modern proofs of Insertion Sort, Quicksort, and Heapsort have been formalized in the Coq system [\[49\]](#), and Counting Sort (see also Exercise 15.3) and Radix Sort have been verified in the KeY system [\[37\]](#).

In the Notes section of Chapter 8, I remarked about some functional sorting algorithms. The Why3 gallery of verified programs I mentioned there also contains several imperative sorting algorithms [\[20\]](#).

An attempt to formally verify TimSort in Java's standard library revealed a bug in the implementation [\[38\]](#). Informed by the failed proof, the bug was fixed and the corrected algorithm was verified in the KeY system.

In the development of Quicksort in Section 15.2, it was convenient to declare and use the `twostate` predicate `SwapFrame`. While some verifiers support such predicates (e.g., Frama-C [\[14\]](#) supports user-defined predicates that relate two or more memory states), not all do. Fortunately, our Quicksort program can easily be written without a name for this predicate, as the next exercise asks you to do:

Exercise 15.10.

In the Quicksort program in Section 15.2, replace all calls to the two-state predicate `SwapFrame` with the body of `SwapFrame` (suitably parameterized), and then remove the declaration of `SwapFrame` altogether. Make sure your new program still verifies.

Chapter 16

Objects



An *object* is a stateful abstraction with identity. In class-based languages (which includes Dafny), each object is created as an instance of a *class*. Access to an object always goes via a *reference* (which is also true of arrays). The class defines state components of its objects, which are called *fields*, as well as operations on the objects, which are functions and methods. Collectively, fields, functions, and methods of a class are called *members* of the class. The

identity of an object is determined by its reference, not by the value of its fields.

In this chapter, I'll give three examples that use objects. The examples gradually introduce how we write specifications for and reason about objects. The two essential ingredients are *invariants* of the objects structures and *frames*.

16.0. Checksums

Let's write a class `ChecksumMachine`, whose objects compute checksums for some data. Clients can provide the data in pieces, rather than having to supply all the data to be checksummed at once. For this reason, we need something stateful, which is why using an object is appropriate.

The data will be a string, that is, a sequence of characters (Dafny defines **string** as a synonym for **seq<char>**). The checksum will be a simple hash code: add up the integer values of the characters in the string and then take the remainder after dividing by 137. The expression `ch as int` is used to convert the character value `ch` to an integer value. Here is the definition of our hash function, which is ghost, since we'll call it only in specifications:

```
ghost function Hash(s: string): int {  
    SumChars(s) % 137  
}  
  
ghost function SumChars(s: string): int {  
    if |s| == 0 then 0 else  
        var last := |s| - 1;  
        SumChars(s[..last]) + s[last] as int  
}
```

16.0.0. Specification

The view of our class that we want to present to clients is

```

class ChecksumMachine {
  var data: string
  constructor ()
    ensures data == ""
  method Append(d: string)
    modifies this
    ensures data == old(data) + d
  function Checksum(): int
    reads this
    ensures Checksum() == Hash(data)
}

```

This declaration introduces `ChecksumMachine` as a class with one field, `data`, which is used to accumulate the pieces of data that clients provide via the `Append` method. Any access of a member of the class requires an object reference. For example, if `m` is a reference to a `ChecksumMachine` object, then `m.data` denotes the `data` field of that object.

The class provides a *constructor*, which is used to initialize an object upon creation. A class can declare several constructors, and these can be named, just like methods. Commonly, a class has an *anonymous* constructor, as this example shows. Constructors can have parameters, but no parameters are needed in our example. The postcondition of a constructor tells callers of the state of the object after construction.

The form of method `Append` and function `Checksum` should look familiar, but there is one difference from what we have seen previously. Since this method and function are declared as members of the type `ChecksumMachine`, they implicitly take an additional parameter known as the *receiver* and named **this**. You see it used in the **modifies** clause of `Append`, which specifies that `Append` may change the fields of **this**, and in the **reads** clause of `Checksum`, which specifies that `Checksum` may depend on the fields of **this**.

The implicit nature of **this** provides a natural way to refer to fields and methods in the class. The specifications in the

example show several uses of the expression `data`, which is a shorthand for `this.data`.

Method `Append` can modify the state of its receiver parameter, and its postcondition relates the pre-state and post-state values of `this.data`. The pre-state value of an expression is denoted using `old`, which I introduced in Section 14.0.1. As I mentioned in that section, `old` only affects heap dereferences in its argument expression. For objects, this means the (implicit or explicit) `."` that dereferences fields. For example, `old(this.data)` and `old(data)` refer to the pre-state value of the `data` field of `this`. In contrast, `old(this).data` uses an `old` whose argument expression does not dereference the heap, so it is the same as the expression `this.data`.

I used a postcondition with function `Checksum()` rather than giving its body. This will allow us to give an alternative, but equivalent, expression in the body, once we implement the function. In the postcondition, the mention of `Checksum()` itself refers to the value returned by the function (see Chapter 6).

Exercise 16.0.

Define bodies with simple implementations for the constructor, method, and function in `ChecksumMachine`.

16.0.1. Test harness

Before we discuss a good implementation of `ChecksumMachine`, let's write a test harness for it (cf. Section 9.3.3 and Exercise 10.0). The test harness shows how a client uses the class and checks that our specifications are reasonable.

```
method ChecksumMachineTestHarness() {  
  var m := new ChecksumMachine();  
  m.Append("green");  
  m.Append("grass");  
  var c := m.Checksum();  
  print "Checksum is ", c, "\n";  
}
```

}

This method allocates a `ChecksumMachine` object and calls its constructor. Local variable `m` is set to reference the object. The harness then calls `Append` on the object referenced by `m`. Notice how the receiver parameter, `m`, is written before the name of the method being called. Finally, the test harness obtains the checksum of "greengrass" and prints the result.

It's worth thinking about how framing works in this example. The test-harness method does not declare any **modifies** clause of its own. This means that it's not allowed to modify the state of any previously existing object. Extending the frame bylaw I stated about arrays in Section 14.0.2, we have

A method is allowed to allocate new arrays and objects and change their state (that is, the elements of the arrays and the fields of the objects) without mentioning these arrays and objects in the **modifies** clause.

The test-harness method above allocates a `ChecksumMachine` object, and therefore it is allowed to modify the state of that object. The frame specification of `Append` says `Append` can modify the state of its receiver parameter, which the test harness passed in as `m`. We conclude that `ChecksumMachineTestHarness` satisfies its frame specification.

16.0.2. Invariant

To write a good implementation for `ChecksumMachine`, we would like to compute the checksum incrementally. That is, we want to keep track of a checksum-so-far, and we'll update it with each call to `Append`. Then, we don't need to store the concatenation of the appended strings at run time, and the `Checksum` function will evaluate to the checksum-so-far.

To accomplish this, we'll introduce a field for the checksum-so-far. Inside the class, we add

```
var cs: int
```

We want to continue using data in specifications, but not in the compiled program, so we change it from a compiled field to a ghost field:

```
ghost var data: string
```

The next step takes more work. We want to record our design decision to maintain the relation

```
cs == Hash(data)
```

This is an *invariant* of each object. We have seen similar data-structure invariants before, namely in Chapter 10, where our data structures were immutable and defined by **datatype** declarations. Analogously to what we did there, we will here declare a predicate `Valid()` that is **true** for an object whenever the object satisfies its invariant. What makes this predicate an object invariant comes down to how we use it: as a postcondition of the constructor, and as a pre- and postcondition of every method. In other words, the constructor establishes `Valid()` initially, and all methods maintain it.

Here is our definition of the validity predicate:

```
ghost predicate Valid()  
  reads this  
{  
  cs == Hash(data)  
}
```

Because the function `Valid()` reads the fields of **this**, it must declare a **reads** clause. I described such read frames for arrays in Section 14.0.4. Here is that frame bylaw, stated for objects:

If a function accesses the fields of an object `o`, its specification must include **reads** `o`.

Sidebar 16.0

In this chapter, I have more or less duplicated the frame bylaws for arrays into analogous ones for objects. Had I presented material in the opposite order, I might just have pointed out that arrays *are* objects, but with some special syntax. Array elements are “fields” of an array object, and the “field names” for the elements are not named as literals but rather are computed by integer expressions.

Here is an updated version of the portion of our class that we want clients to see:

```
class ChecksumMachine {  
  ghost var data: string  
  ghost predicate Valid()  
    reads this  
  constructor ()  
    ensures Valid() && data == ""  
  method Append(d: string)  
    requires Valid()  
    modifies this  
    ensures Valid() && data == old(data) + d  
  function Checksum(): int  
    requires Valid()  
    reads this  
    ensures Checksum() == Hash(data)  
}
```

Note that `cs` is a private implementation decision in the class, so I omitted it from the portions of the class I showed above. More to the point, `cs` does not appear in specifications. This is good information-hiding practice—it frees clients from having to know this detail and it gives the implementation the freedom to change the detail in the future (I talked a lot about information hiding in Chapter 9,

and I will declare a module for the current example in Section 16.0.4).

When we make a change in specifications, it's a good idea to update test harnesses, too. In our case, the addition of `Valid()` does not affect our test harness, which still verifies without any changes.

16.0.3. Implementation

We've designed our class. Next, we implement the constructor, method, and function.

The constructor and the function `Checksum` are easy:

```
constructor ()  
  ensures Valid() && data == ""  
{  
  data, cs := "", 0;  
}  
  
function Checksum(): int  
  requires Valid()  
  reads this  
  ensures Checksum() == Hash(data)  
{  
  cs  
}
```

The constructor modifies the fields of **this**, but **this** is considered newly allocated inside the constructor, so no **modifies** clause is needed.

A constructor is allowed to assign to the fields of the object being constructed, **this**, without mentioning **this** in the **modifies** clause.

Method `Append` can be implemented by a loop that processes one character of parameter `d` at a time:

```
method Append(d: string)  
  requires Valid()  
  modifies this
```

```

ensures Valid() && data == old(data) + d
{
  var i := 0;
  while i != |d|
    invariant 0 <= i <= |d|
    invariant Valid()
    invariant data == old(data) + d[..i]
  {
    cs := (cs + d[i] as int) % 137;
    data := data + [d[i]];
    i := i + 1;
  }
}

```

Exercise 16.1.

(Advanced) At the beginning of this section, I defined `SumChars` to recurse on a prefix of the given string (akin to the `SumDown` function in Section 12.3.0). This works well with the loop above in `Append`. Change the definition of `SumChars` to instead recurse on a suffix of the given string (akin to `SumUp` in Section 12.3.0). Then, adjust the body of `Append` to make it verify. (Caution: This exercise takes some patience, because you'll need to assert some properties about sequences.)

16.0.4. Module

I've been rather informal above about the “client's view” of the class. Let's wrap up the example by wrapping a module and export set around it. To jog your memory about modules and export sets, see Section 9.2.

```

module Checksums {
  export
    reveals Hash, SumChars, ChecksumMachine
    provides ChecksumMachine.data, ChecksumMachine.Valid
    provides ChecksumMachine.Append, ChecksumMachine.Checksum

  // the definitions from earlier in this section...
}

```

Here, I decided to reveal `Hash` and `SumChars`. If instead you want to keep the body of `Hash` private to the module, then you can change the export set to provide `Hash` and to omit `SumChars` altogether.

The export set reveals `ChecksumMachine`, which tells importers that this type is a class. By revealing a class, the anonymous constructor of the class (if any) is automatically provided in the export set as well. Any other members that are to be exported need to be explicitly mentioned in the export set. For example, the export set above makes the field `data` available to importers, whereas the field `cs` is not exported and thus cannot be seen or used outside the module.

16.0.5. Summary

In this example, we started with a simple class where `data` was a compiled field used in both specifications and (see Exercise 16.0) implementations.

In some order, we then added a compiled field `cs` and used a `Valid()` predicate to state the invariant relationship between `data` and `cs`. We changed specifications to say that the constructor establishes `Valid()`, that the function requires `Valid()`, and that the method maintains `Valid()`. We changed the implementations to use just `cs`, not `data`, and this allowed us to mark `data` as ghost. Finally, we wrote a module with an export set for our class.

Exercise 16.2.

Write a variation of the `Checksums` module where the class keeps track of `SumChars( data)` instead of `Hash(data)`. That is, whereas the program above uses a field `cs` with the invariant `cs == Hash(data)`, use a field `sum` with the invariant `sum == SumChars( data)`. The client's view of the module should be the same as in the example above.

Exercise 16.3.

The program in Exercise 16.2 performs one modulus operation for each invocation of function `Checksum()`. So, if the function is called twice without any change to data, the value of `sum % 137` will nevertheless be re-computed. If an operation (here, the modulus operation) is expensive, one can use the technique of *memoization* to store and reuse the computed value. Modify your answer to Exercise 16.2 to use memoization.

Specifically, change `Checksum()` from a function to a method, so that it can modify the state of the `ChecksumMachine` object:

```
method Checksum() returns (checksum: int)  
  requires Valid()  
  modifies this  
  ensures Valid() && data == old(data)  
  ensures checksum == Hash(data)
```

Then, introduce a field `var ocs: Option<int>`, where `Option` is the type

```
datatype Option<T> = None | Some(T)
```

Your invariant will be that `ocs` is either `None` (if no value is being memoized) or `Some(sum % 137)`. In method `Checksum()`, return the memoized checksum if there is one; otherwise, compute the checksum, memoize it, and return it.

Exercise 16.4.

Write an imperative version of the `Blockchain` program in Exercise 10.5. Specifically, declare `Ledger` to be a class

```
class Ledger {  
  var transactions: List<int>  
  var balance: nat  
  
  ghost predicate Valid()  
    reads this  
  {  
    balance == Sum(transactions)  
  }  
}
```

```
// ...  
}
```

where `List` is the datatype from Section 6.0 and `Sum` is a recursive ghost function you'll write to add up the numbers in a given `List<int>` (hint: `Sum` will look similar to the `Length` function in Section 6.1). Your class will also have a constructor and two methods, `Deposit` and `Withdraw`, that mutate the object. Write the specifications of these in terms of `Valid()` and `balance`. Write and verify a test harness to make sure your specifications are adequate. Implement and verify the class.

16.1. Tokenizer

A common task in software is to convert groups of small items into larger items. One such example is a *tokenizer*, which parses a stream of characters into a stream of strings in various syntactic categories. In this section, I present a class that implements a simple tokenizer. It reads characters from a fixed string and chops these into tokens, which it returns one at a time. The example uses simple invariants, which we'll encode as a validity predicate, just like in the previous section.

16.1.0. Syntactic categories

We start with the syntactic categories:

```
datatype Category = Identifier | Number | Operator  
                  | Whitespace | Error | End
```

Any input that doesn't fit the first four categories is considered an error. The category `End` denotes end-of-input.

Next, we need a way to check if a character is in a given category. In this simple tokenizer, categories do not share characters (but see Exercise 16.6). We don't intend to ever

call this predicate on `End`, so we'll exclude it using a precondition.

```
predicate Is(ch: char, cat: Category)
  requires cat != End
  decreases cat == Error
{
  match cat
  case Whitespace => ch in " \t\r\n"
  case Identifier => 'A' <= ch <= 'Z' || 'a' <= ch <= 'z'
  case Number => '0' <= ch <= '9'
  case Operator => ch in "+-*/%!=><~^&|"
  case Error =>
    !Is(ch, Identifier) && !Is(ch, Number) &&
    !Is(ch, Operator) && !Is(ch, Whitespace)
}
```

Two of the alternatives use `in` to check membership in a sequence of characters, that is, a **string**. The **decreases** clause is needed to justify the termination of the recursive calls. These will decrease the boolean `cat == Error` from **true** to **false** (see Section 3.2).

Exercise 16.5.

For predicate `Is`, what happens if you (a) remove the precondition or (b) remove the **decreases** clause?

16.1.1. The class and invariant

An instance of class `Tokenizer` has a source string, which we'll declare as a field. Because this field stays the same throughout the life of the object, we can declare it to be an *immutable field*, which is done with the keyword **const**. An immutable field is defined by the constructor and is never changed thereafter.

```
class Tokenizer {
  const source: string
}
```

The remaining parts of this example are declared inside this class as well.

The only other piece of the state of the tokenizer is a field `n` that keeps track of how many characters have been processed so far. This means `n` will always be in the range from 0 through `|source|`, so we'll write that down in the validity predicate that captures the object invariant:

```
var n: nat
ghost predicate Valid()
  reads this
{
  n <= |source|
}
```

Here, I chose to declare `n` as a `nat`. An alternative is to declare it as an `int`, in which case the invariant also needs to state the lower bound `0 <= n`.

To construct a `Tokenizer` object, we need the source string to be tokenized:

```
constructor (s: string)
  ensures Valid() && source == s && n == 0
{
  source, n := s, 0;
}
```

16.1.2. Read specification

The most interesting part of the tokenizer remains: method `Read()`, which returns the next token. Our design will be for the method to discard whitespace tokens. The outparameters of the method are the category of the next non-whitespace token as well as a string representing the token itself. To make the specification precise, it will be convenient to be able to talk about the source-string position where the returned token begins. Unless there's a need to return that position at run time, we can declare this out-parameter as `ghost`.

```
method Read() returns (cat: Category, ghost p: nat, token: string)
```

To maintain or not maintain validity

The specification of `Read` starts in the same way as the mutating method `Append` in the previous section:

```
requires Valid()  
modifies this  
ensures Valid()
```

This is typical for mutating methods, but far from universal. For `Read`, we need to think about what should happen once the end of input has been reached or if an error has been found. As I wrote it above, `Read` always maintains validity, and validity is all that is required for a call. Consequently, `Read` can be called even after it detects end-of-input or an error.

If we wanted to do this differently, then we would need to arrange for the precondition not to hold when a previous call to `Read()` has returned `End` or `Error`. But how would we refer to a previous return value of `Read()`? One way is to introduce a ghost field `ReadIsCallable`, set it to **false** when `Read` returns `End` or `Error`, and to require `ReadIsCallable` as a precondition to `Read`.

A simpler and equally effective alternative is for `Read` not to promise to maintain validity after detecting end-of-input or an error. To do this, we would simply change the postcondition above to

```
ensures cat == End || cat == Error || Valid()
```

This renders the tokenizer unusable once end-of-input or an error has been detected, because the only way a caller can conclude that `Valid()` holds after a call is if the call does not return a category of `End` or `Error`.

It is worth stressing that this is an underspecification. That is, it *may* be that `Valid()` actually holds even after end-of-input or an error. A specification like

```
ensures cat == End || cat == Error ==> !Valid()
```

would say that validity definitely does not hold in this case. Such a specification serves no one, so it is better to use underspecification of validity.

Well, I'm going to proceed with the postcondition I first wrote above, which is to say that validity is always maintained by Read.

Postcondition

Postconditions are designed to benefit callers, and every postcondition creates an obligation for the implementation. It is not appropriate to include in the postcondition everything that is true after the call, because some of those things are private implementation decisions. For the Read method of the tokenizer, it will feel like we're trying to include everything we can possibly think of, because I will write many conditions. I'd like to think of these conditions as helping us think through the design of the method. But if you feel differently, you may want to write fewer postconditions than I'm about to.

I already mentioned that the tokenizer will discard whitespace tokens. We'll write that as a postcondition:

```
ensures cat != Whitespace
```

I alluded to that we're going to return out-parameter *p* as the position in the source string after skipping whitespace. Thus, we have

```
ensures old(n) <= p <= n <= |source|
```

Category End refers to the end of source string. It is returned when *p* equals *|source|*. In fact, that is the only circumstance under which End is returned:

```
ensures cat == End <==> p == |source|
```

If skipping whitespace takes us to end-of-input or an error, then we do no more parsing. That is, we'll have *p* == *n*. For all other categories, we do more parsing.

```
ensures cat == End || cat == Error <==> p == n
```

Let's now connect the characters of the input to categories. First up, `source[old(n) .. p]` will be whitespace:

```
ensures forall i :: old(n) <= i < p ==>
  Is(source[i], Whitespace)
```

The next stretch of the input, `source[p..n]` will be of the returned category. Note that this is also the case when the category is `End` or `Error`—trivially so, because then `p == n`:

```
ensures forall i :: p <= i < n ==> Is(source[i], cat)
```

The idea is to parse the input into the longest possible tokens. This is documented by the following postcondition, which says that what follows `n` is a different category:

```
ensures p < n ==> n == |source| || !Is(source[n], cat)
```

The antecedent `p < n` in this postcondition is needed, since the rest of the condition does not hold when the category is `Error`.

Finally, we say what `token` returns as:

```
ensures token == source[p..n]
```

This is a long postcondition with many details. Good thing we have the help of the verifier to make sure we get all the details right in the implementation and the proof thereof.

16.1.3. Read implementation

The implementation of `Read` consists of three parts: skip whitespace, determine the syntactic category, and scan the token.

The first part is done by a loop. We have many postconditions to establish, so we'll include loop invariants for those that pertain to `p` and `source[old(n)..p]` and whitespace:

```
{
  // skip whitespace
  while n != |source| && Is(source[n], Whitespace)
    invariant old(n) <= n <= |source|
    invariant forall i :: old(n) <= i < n ==>
```

```

        Is(source[i], Whitespace)
    {
        n := n + 1;
    }
    p := n;

```

If you look through the method postconditions, you'll see that the invariants of this loop account for as many of them as is possible in this part of the method body.

Next up, we determine the value of cat:

```

// determine syntactic category
if n == |source| {
    return End, p, "";
} else if Is(source[n], Identifier) {
    cat := Identifier;
} else if Is(source[n], Number) {
    cat := Number;
} else if Is(source[n], Operator) {
    cat := Operator;
} else {
    return Error, p, "";
}

```

In two of these cases, we will return with $p == n$. The postcondition tells us token should be the empty string in those cases.

Finally, we increment n while we're still seeing characters of category cat:

```

// read token
var start := n;
n := n + 1;
while n != |source| && Is(source[n], cat)
    invariant p <= n <= |source|
    invariant forall i :: p <= i < n ==> Is(source[i], cat)
    {
        n := n + 1;
    }
token := source[start..n];
}

```

Both of these loop invariants use the *always* Loop Design Technique 13.1. The final two postconditions follow from

the negation of the loop guard and the assignment to `token`, respectively.

Exercise 16.6.

Extend the `Tokenizer` example to treat the first character of a token differently. Specifically, replace predicate `Is` with two predicates: `IsStart(ch, cat)` says whether or not character `ch` is a legal initial character of category `cat`, and `IsFollow(ch, cat)` says whether or not `ch` is a legal non-initial character of `cat`. Use these to allow `Identifier` to start with a letter and to continue with any letters and digits. Then, update the specification and implementation of method `Read()`.

Exercise 16.7.

As written, `Tokenizer` holds on to the given string forever. Change the design to make it possible to prune the storage through a method that throws away the prefix of the source string that has already been processed. Specifically, make `source` a **ghost** and declare a mutable field `suffix` with the invariant `suffix == source[m..]`, where `m` is a new field. For example, you can introduce fields `m` and `j` such that `m + j == n`, in which case you can declare `n` as ghost as well. Modify the implementation of `Read` accordingly.

16.2. Simple Aggregate Objects

Objects are often implemented in terms of other objects. Such objects are called *aggregate objects* and are the subject of the rest of this book. I'll introduce them in two sections. In this section, we'll build an aggregate object from the simple kinds of objects we've seen in the previous two sections. Then, in section 16.3, we'll build an aggregate object from other aggregate objects.

In both this section and the next, the example I use is a coffee maker built from a grinder and a water tank. That is, a coffee maker is an aggregate object whose *constituent objects* are a grinder and a water tank. This example will let us focus on the structure of the aggregate object. The resulting class is more of a sketch than a useful program, but after learning the specification technique here, we'll write some useful programs in the next chapter.

16.2.0. Constituent objects with simple frames

The coffee-maker class we're about to write makes use of two other classes. Here is the coffee maker's view of those classes, which have the form that we have seen in the previous two sections.

```
class Grinder {
  var HasBeans: bool
  ghost predicate Valid()
    reads this
  constructor ()
    ensures Valid()
  method AddBeans()
    requires Valid()
    modifies this
    ensures Valid() && HasBeans
  method Grind()
    requires Valid() && HasBeans
    modifies this
    ensures Valid()
}

class WaterTank {
  var Level: nat
  ghost predicate Valid()
    reads this
  constructor ()
    ensures Valid()
  method Fill()
    requires Valid()
    modifies this
    ensures Valid() && Level == 10
}
```

```

method Use()
  requires Valid() && Level != 0
  modifies this
  ensures Valid() && Level == old(Level) - 1
}

```

As you can see, each class declares a validity predicate that it uses as an object invariant. The details of validity are not important to the coffee maker, other than the fact that `Valid()` depends only on the state of **this**.

We'll use one more class, `Cup`, which will be used in the output of the coffee maker. The use of the `Cup` class is rather artificial, but is meant to be suggestive of some useful output. Indeed, all I'm saying about `Cup` is that it has a constructor:

```

class Cup {
  constructor ()
}

```

16.2.1. CoffeeMaker version 0

In addition to a validity predicate and a constructor, our coffee maker supports a function `Ready()` that says if the coffee maker currently has all the supplies it needs to make a cup of coffee, a method `Restock()` that replenishes the coffee maker's supplies, and a method `Dispense()` that produces a cup of coffee, provided the coffee maker has the necessary supplies.

```

class CoffeeMaker {
  ghost predicate Valid()
    reads this
  constructor ()
    ensures Valid()
  predicate Ready()
    requires Valid()
    reads this
  method Restock()
    requires Valid()
    modifies this
    ensures Valid() && Ready()
}

```



```

method Dispense(double: bool) returns (c: Cup)
  requires Valid() && Ready()
  modifies this
  ensures Valid()
}

```

As you see in this class, predicate `Valid()` is used as an object invariant, following the style we saw in the previous sections of this chapter. Function `Ready()` can be queried before calling `Dispense()`, and a call to `Restock()` guarantees that one call to `Dispense()` can be made.

To start an implementation of this class, we'll add fields to hold a grinder and a water tank:

```

var g: Grinder
var w: WaterTank

```

The coffee maker is valid when both of these objects are. We record this in the validity predicate of class `CoffeeMaker`:

```

ghost predicate Valid()
  reads this
{
  g.Valid() && w.Valid() // error: insufficient reads clause
}

```

Alas, this does not work. Predicate `Valid()` is declared to read only the fields of `this`, but it actually also reads the fields of `this.g` and `this.w`, since `g.Valid()` and `w.Valid()` reads those. To accommodate these other dependencies, we can adjust the `reads` clause of `Valid()` like this:

```

ghost predicate Valid()
  reads this, g, w

```

We'll make the same change to the `reads` clause of `Ready()` and change the `modifies` clauses of `Restock()` and `Dispense()` to

```

modifies this, g, w

```

With these expanded frames, we have made it possible for the coffee maker to read and write its two constituent objects. However, we have violated good principles of information hiding, because the fact that the `CoffeeMaker`

uses one grinder and one water tank should be a private implementation decision of the class. For example, we don't want `CoffeeMaker` clients to be affected if we later change the implementation of `CoffeeMaker` to use a different set of components for making coffee.

So, the frames need to account for the constituent objects referenced by the fields `g` and `w`, but we don't want to mention these fields in the **reads** and **modifies** clauses. The solution to this frame problem is abstraction. We'll abstract the constituent objects into a *representation set*.

16.2.2. Representation sets

The *representation set* of an object `o` is the set of objects used to implement the abstraction provided by `o`. This always includes `o` itself and also includes `o`'s constituent objects. For a `CoffeeMaker` object `o`, the representation set is $\{o, o.g, o.w\}$. A client that uses `o` through `o`'s methods and functions should not need to know how `o` is implemented. But it is reasonable for the client to be aware that `o` may be implemented in terms of other objects.

Let `Repr` denote the representation set of a `CoffeeMaker` object. We want to list this whole set of objects in **modifies** and **reads** clauses. So far, we have listed individual objects in such clauses, but Dafny also allows **modifies** and **reads** clauses to contain expressions that denote sets of objects. So, if `Repr` is such a set, we just list `Repr` in these frame-specification clauses.

To evolve `CoffeeMaker` into version 1, we declare `Repr` as a field in the class:

```
ghost var Repr: set<object>
```

We need `Repr` only for specification purposes, so we declare it as **ghost**. The type of `Repr` is a set of objects, so `Repr` can contain references to objects of any type (including arrays).

This is good, because the representation set of our coffee maker contains a `CoffeeMaker`, a `Grinder`, and a `WaterTank`.

Next, we change the methods `Restock` and `Dispense` to specify their write frames as `Repr`:

```
modifies Repr
```

Similarly, we change the read-frame specification of `Ready` to

```
reads Repr
```

Somewhere, we need to specify which objects are in `Repr`. This becomes part of the object invariant. Thus, we change the body of `Valid()` to

```
this in Repr &&  
g in Repr && g.Valid() &&  
w in Repr && w.Valid()
```

This says that **this**, `g`, and `w` are part of `Repr`. It would be possible to write

```
Repr == {this, g, w} &&  
g.Valid() && w.Valid()
```

However, it is typical to specify only a lower bound on the objects that are in `Repr`, so I'll stick with what I first wrote above.

That was the body of `Valid()`. According to it, `Repr` is the representation set of a `CoffeeMaker`. In more detail, the body of `Valid()` says that the `Repr` field (whose type is a set of objects) of a valid `CoffeeMaker` object contains the objects that are referenced by fields `g` and `w`.

What about the frame specification of `Valid`? The read frame of `Valid` must include those objects whose fields `Valid` depends on. The body of `Valid` reads the fields of **this** (in particular, the fields `Repr`, `g`, and `w` of **this**) and the fields of `g` and `w` (by calling `g.Valid()` and `w.Valid()`). These three objects are in the representation set, which points in the direction of using the following read frame for `Valid`:

```
reads Repr // error: insufficient reads clause
```

This isn't quite right. This *would* work when `Valid()` holds, because `Valid()` implies that **this**, `g`, and `w` are in `Repr`. But `Valid()` is just a function that can return **false** or **true**. The **reads** clause of `Valid()` must be correct even when `Valid()` returns **false**. The way to solve this problem is simple: also include **this** in the **reads** clause. Here is our correctly defined predicate `Valid()`:

```
ghost predicate Valid()
  reads this, Repr
{
  this in Repr &&
  g in Repr && g.Valid() &&
  w in Repr && w.Valid()
}
```

Because the **reads** clause of `Valid` is special in this way, it is worthwhile to go through the read dependencies of it in great detail:

- The **reads** clause itself mentions **this.Repr**. This is legal, because we also included **this** in the **reads** clause.
- The body of the predicate mentions **this.Repr**. This is legal, again because **this** is included in the **reads** clause.
- The body mentions **this.g** and **this.w**. These are legal, because **this** is included in the **reads** clause.
- The body depends on `g.Valid()` *if* the two preceding conjuncts evaluate to **true**. That is, because `&&` is a short-circuit operator, the third conjunct is evaluated only if the first and second conjuncts evaluate to **true**. The second conjunct is `g in Repr`, so if it evaluates to **true**, then we are in a state where `Repr` contains the object `g`. Since `Repr` is listed in the **reads** clause of `Valid`, the body of `Valid` is allowed to read the fields of any object in `Repr`. So, it is legal for the third conjunct to read the fields of `g`.
- The body depends on `w.Valid()` *if* the four preceding conjuncts evaluate to **true**. Since those four conjuncts

imply that w is in Repr , it is legal for the fifth conjunct to read the fields of w .

Note that if you reverse the order of the second and third conjuncts, then the body of `Valid` may read the fields of g even if g isn't in Repr , so the verifier will report an “insufficient reads clause” error.

16.2.3. Class implementation

Having sorted out the issues with read and write frames, we can move on to the implementation of the methods and other members of the class.

Here is the constructor:

```
constructor ()  
  ensures Valid()  
{  
   $g$  := new Grinder();  
   $w$  := new WaterTank();  
   $\text{Repr}$  := {this,  $g$ ,  $w$ };  
}
```

Note that Repr is just a field like any other. (Well, it is ghost, but the verifier is blind to that distinction.) This means we need to set Repr . To establish the postcondition of the constructor, that is, `Valid()`, we set Repr to {**this**, g , w }. In other words, I'm pointing out that while **reads** and **modifies** clauses are part of specifications in Dafny, the language does not treat Repr specially. Nor does it treat `Valid()` specially. We reason about programs in terms of object invariants and representation sets, and we define these using a predicate we call `Valid()` and a ghost field we call Repr .

Next up, here is the implementation of predicate `Ready`, which will allow the machine to dispense single and double shots of coffee:

```
predicate Ready()  
  requires Valid()
```

```

    reads Repr
  {
    g.HasBeans && 2 <= w.Level
  }

```

Finally, we implement the two methods of `CoffeeMaker` as follows:

```

method Restock()
  requires Valid()
  modifies Repr
  ensures Valid() && Ready()
{
  g.AddBeans();
  w.Fill();
}
method Dispense(double: bool) returns (c: Cup)
  requires Valid() && Ready()
  modifies Repr
  ensures Valid()
{
  g.Grind();
  if double {
    w.Use(); w.Use();
  } else {
    w.Use();
  }
  c := new Cup();
}

```

16.2.4. Test harness

I've emphasized the importance of writing a test harness, so that we check the usability of our specifications. Let's do that for our coffee maker. Here is just about the simplest way we can think of to obtain a cup of coffee:

```

method CoffeeTestHarness() {
  var cm := new CoffeeMaker();
  cm.Restock(); // modifies clause violated
  var c := cm.Dispense(true); // modifies clause violated
}

```

Ay ay ay! Our coffee maker fails to be useful even in this basic use case. The problem reported by the verifier is that

the call to `Restock` may modify objects that `CoffeeTestHarness` is not specified to modify. The test-harness method has an empty **modifies** clause, so that means it is only allowed to modify fields of fresh objects, that is, objects that have been allocated since entry to the method (these frame bylaws were discussed in Section 16.0). The **modifies** clause of `Restock` tells us that the call to `Restock` may modify the fields of any object in `cm.Repr`. What is `cm.Repr` at the time `Restock` is called? Specifically, are the objects in `cm.Repr` fresh? With our current specifications, there is no information about the freshness of `Repr`. Since `Repr` is a field that coffeemaker clients will reason in terms of, we need to include some information about `Repr` in our `CoffeeMaker` specifications.

16.2.5. Specifications about `Repr`

Let's consider how to write specifications about the representation set.

Constructor

First up is the constructor. What can we say about `Repr` in the postcondition of it? From our test harness, we have learnt that callers care about when the objects in `Repr` have been allocated. To that end, we'll add the following postcondition to the constructor in version 2 of `CoffeeMaker`:

ensures **fresh**(`Repr`)

We briefly saw **fresh** before (Section 14.0.3), applied to one array. Here, we're using a set of objects as an argument to **fresh**, which says that every object in that set is newly allocated. By adding this postcondition to the constructor, we eliminate the error reported about the call to `Restock` in our test harness. Progress!

Maintaining freshness

The verifier still complains about the second method call, saying that this call to `Dispense` may modify more objects than are allowed by the specification of `CoffeeTestHarness`. Again, the problem is that we don't have enough information about `Repr`. Let me describe three ways out.

No change One way to specify the effect of the `CoffeeMaker` methods on `Repr` is to realize that `Repr` is not changed by these methods. Hence, we can add the following postcondition to both `Restock` and `Dispense`:

ensures `Repr == old(Repr)`

With these postcondition specifications, our little test harness finally verifies.

Immutable fields An alternative way to specify the methods' effect on `Repr` is to realize that `Repr` is never changed after the constructor. This means we can declare the `Repr` field to be *immutable*:

ghost const `Repr: set<object>`

When (ghost or compiled) fields are not intended to be changed, using such **const** declarations is nice. That keeps us from accidentally assigning to the fields and it also saves us from having to write postconditions that say the fields don't change.

Newly added objects are fresh The two previous ways of specifying the methods' effects on `Repr` do work for our `CoffeeMaker` class. However, saying that `Repr` never changes limits the implementation of the class. For example, what if the class sometimes has a need to replace the grinder by a new `Grinder` object (for example, after every 1000 cups of coffee)? Even if there is no need to do that in our current `CoffeeMaker` class, this kind of extensibility gives future changes of the class a way to incorporate new constituent objects during the lifetime of the aggregate object.

The third, and preferred, way to specify a mutating method's effect on `Repr` is to admit in the postcondition that the method may add objects to `Repr`. But the method promises only to add newly allocated objects to `Repr`. In other words, the post-state value of `Repr` may contain any or all of the objects contained in `Repr` in the pre-state, and all additional objects added to `Repr` are fresh. Here's how we write that specification:

```
ensures fresh(Repr - old(Repr))
```

With this specification, our test harness verifies.

Abandoning, capturing, or releasing constituent objects

Most commonly, constituent objects are allocated and retained by the aggregate object that uses them. Indeed, the aggregate objects we have encountered so far do that. But what happens if the aggregate object stops using one of its constituent objects, what happens if an aggregate object can be customized by a client-provided constituent object, and what happens if the aggregate object releases a constituent object to its client? Let's consider a couple of examples to see how the specifications and code can be adjusted for such situations.

As a first example, consider a method that changes the coffee maker's grinder. That is, instead of always using the same grinder, the coffee maker may abandon one grinder and start using a new one. Here's how I would write the specification and code:

```
method ChangeGrinder()  
  requires Valid()  
  modifies Repr  
  ensures Valid() && fresh(Repr - old(Repr))  
{  
  g := new Grinder();  
  Repr := Repr + {g};  
}
```

This specification is *the* standard specification for a mutating method. The body of the method allocates a new Grinder and then—remember!—adds that new object to Repr.

But what happened to the previous value of *g*? Typically, that previous Grinder object was accessible only from this field of this CoffeeMaker. When we assign a new value to *g*, the previous Grinder object becomes unreachable at run time, and so the run-time system will eventually reclaim the storage used by that object in a garbage-collected language like Dafny.

The body of `ChangeGrinder` above does not remove the previous Grinder object from Repr. This is typical, especially if you expect that object to become unreachable in the executing program. We could remove the old *g* from Repr, but there's usually no point in doing so. This is one reason the validity predicate only gives a lower bound on the elements in Repr.

As the second example, consider a method that accepts a Grinder from a client:

```
method InstallCustomGrinder(grinder: Grinder)
  requires Valid() && grinder.Valid()
  modifies Repr
  ensures Valid() && fresh(Repr - old(Repr) - {grinder})
{
  g := grinder;
  Repr := Repr + {g};
}
```

To establish the first postcondition, `Valid()`, we need for grinder to be valid. This follows from the precondition. As for freshness of the objects added to Repr, it is not possible to establish the typical postcondition `fresh(Repr - old(Repr))`, because the given Grinder was allocated before the call to `InstallCustomGrinder`. Therefore, the specification subtracts off {grinder} from the set of fresh additions to Repr. This postcondition says that grinder is *captured* by the CoffeeMaker object.

To be more precise, the specification *allows* grinder to be captured. We could add grinder **in** Repr as a postcondition, but there's usually no benefit to doing so, and therefore such a postcondition is better omitted.

Exercise 16.8.

Specify and implement a method that *releases* (that is, removes and returns) the grinder of a coffee maker, replacing it with a new one that the method allocates. Be certain your specification allows the following test harness to be verified:

```
method RemoveGrinderHarness() {  
    var cm := new CoffeeMaker();  
    var grinder := cm.RemoveGrinder();  
    cm.Restock();  
    grinder.AddBeans();  
}
```

16.2.6. Summary of specification idiom

Programming with aggregate objects is not easy. In this section, I have shown how to write specifications for an aggregate object whose constituent objects are of the simple kind we saw in Sections 16.0 and 16.1. There are many details to keep track of, so specifying and verifying programs with aggregate objects takes patience.

The two basic building blocks are object invariants and representation sets.

In the idiomatic specifications I'm using, an object invariant is encoded as a validity predicate, by convention named `valid()`. Such a validity predicate is typically a postcondition of every constructor, a precondition of every function, and a pre- and postcondition of every method.

In the idiomatic specifications I'm using, representation sets are declared as a ghost field `Repr`, and relevant objects of `Repr` are specified and maintained as object invariants (so, in the validity predicate). Typically, methods have the

frame specification **modifies** `Repr`, and functions have the frame specification **reads** `Repr`, except the function `Valid()`, whose frame specification is given as **reads** **this**, `Repr`. Typically, constructors have a postcondition **fresh**(`Repr`) and methods have a postcondition **fresh**(`Repr` - **old**(`Repr`)).

Following this specification idiom, the frame for an object is `Repr`. Because `Repr` is a variable that dynamically can change over time (for example, expanding into newly allocated objects), the specification idiom I'm using is called *dynamic frames* [69]. (The name of the programming language Dafny comes from a permutation of some letters in $\text{Dyn}^{\text{amic}} \text{f}^{\text{ra}}\text{mes}$.)

Exercise 16.9.

In Section 16.0, we wrote class `ChecksumMachine` with simple frames (in particular, **reads this** and **modifies this**). Add a ghost variable `Repr` in that class to stand for the representation set. Then, change the specifications of that class to follow the idiom introduced in this section.

16.3. Full Aggregate Objects

By the class definitions in the previous section, a `CoffeeMaker` object is an aggregate object that contains a constituent `Grinder` object and a constituent `WaterTank` object. The representation set of a `CoffeeMaker` object contains not just the `CoffeeMaker` object, but also the two constituent objects. We encoded the representation set as a ghost field `Repr` in class `CoffeeMaker`. The `Repr` field abstracts over the constituent objects, so that clients of `CoffeeMaker` don't need to, or get to, see the details of what constituent objects are used to implement the `CoffeeMaker`. As such, `Repr` is the dynamic frame of a `CoffeeMaker`.

In the previous section, I introduced `Grinder` and `WaterTank` as objects with simple frames. Commonly, such objects are

themselves aggregate objects. In this section, I'll show the CoffeeMaker example again, but this time with Grinder and WaterTank having dynamic frames.

16.3.0. Constituent objects with dynamic frames

Here is the CoffeeMaker's view of the embellished Grinder and WaterTank classes. The difference from Section 16.2.0 is the introduction of a Repr field in each class, frame specifications that mention Repr, and postconditions about the freshness of Repr:

```
class Grinder {
  var HasBeans: bool
  ghost var Repr: set<object>
  ghost predicate Valid()
    reads this, Repr
  constructor ()
    ensures Valid() && fresh(Repr)
  method AddBeans()
    requires Valid()
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr)) && HasBeans
  method Grind()
    requires Valid() && HasBeans
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr))
}

class WaterTank {
  var Level: nat
  ghost var Repr: set<object>
  ghost predicate Valid()
    reads this, Repr
  constructor ()
    ensures Valid() && fresh(Repr)
  method Fill()
    requires Valid()
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr)) && Level == 10
  method Use()
```

```

    requires Valid() && Level != 0
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr)) &&
        Level == old(Level) - 1
}

```

16.3.1. CoffeeMaker invariant: subsets

The representation set of a CoffeeMaker now includes the whole representation sets of its constituent objects. We change the body of Valid() to be

```

this in Repr &&
g in Repr && g.Repr <= Repr && g.Valid() &&
w in Repr && w.Repr <= Repr && w.Valid()

```

16.3.2. Two-phase constructors

For the new CoffeeMaker constructor, we want to write something like

```

constructor ()
    ensures Valid() && fresh(Repr)
{
    g := new Grinder();
    w := new WaterTank();
    Repr := {this, g, w} + g.Repr + w.Repr; // illegal first-
phase
                                           // use of fields
}

```

However, this is not legal Dafny code. In Dafny classes, constructors are divided into two *phases*. The purpose of the first phase is to set the object's fields to values of their respective types and to define the values of immutable fields. If you don't care what the initial value of a field is and the field is of a type that the compiler knows how to initialize (a so-called *auto-init type*), then you don't have to assign to the field in the constructor. In this first phase, the object is still being constructed, so there are restrictions on what you can do with the fields.

The assignment to `g` and the particular use of `g` in the set display in the righthand side of the assignment to `Repr` are allowed, but the expression `g.Repr` (that is, `this.g.Repr`) is not allowed in the first phase of the constructor. Here is one way to legally write the code above in the first phase of the constructor:

```
var gg := new Grinder();
var ww := new WaterTank();
g, w := gg, ww;
Repr := {this, g, w} + gg.Repr + ww.Repr;
```

Expression `gg.Repr` avoids the use of possibly uninitialized fields of `this`, so it is accepted in the first phase.

Another way to write a legal constructor is to place the assignment of `Repr` in the second phase. Since the type of `Repr` is a set (which is an auto-init type), the compiler is able to automatically assign an initial value to `Repr` in the first phase. Our assignment in the second phase then replaces the compiler's arbitrary initial value. The second phase of a constructor is usually left empty, but if you want to write code there, then you use the special statement `new`; to mark the end of the first phase and the beginning of the second phase. In the second phase of the constructor, all fields have values of their respective types, so there are no restrictions on how you may use the fields. Thus, here is another way to legally write the body of the `CoffeeMaker` constructor:

```
g := new Grinder();
w := new WaterTank();
new;
Repr := {this, g, w} + g.Repr + w.Repr;
```

If for some reason you want to, it's also possible to assign *some* value to `Repr` in the first phase, and then replace or update the field in the second phase. For example, the following is another legal way to write the constructor of `CoffeeMaker`:

```
g := new Grinder();
```

```

w := new WaterTank();
Repr := {this, g, w};
new;
Repr := Repr + g.Repr + w.Repr;

```

16.3.3. Separation among representation sets

Predicate `Ready()` is the same as in Section 16.2, but method `Restock()` poses some new problems. Let's take a look.

```

method Restock()
  requires Valid()
  modifies Repr
  ensures Valid() && fresh(Repr - old(Repr)) && Ready()
{
  g.AddBeans();
  w.Fill(); // precondition violation; modifies violation
} // postcondition violation

```

The verifier reports several errors, the first one of which is a precondition violation at the call to `Fill`. When you encounter this situation for the first time (or second time, or third time, . . .), it can be rather perplexing. Therefore, let me guide you through a systematic process of figuring out what's going on.

We'll start by focusing on the failing precondition of `Fill`. To debug what's going on, we add an assertion of that failing condition immediately before the call to `Fill`:

```

g.AddBeans();
assert w.Valid(); // assertion violation
w.Fill(); // modifies violation

```

The verifier immediately gives us the feedback that assertion may be violated, but for any execution that gets past the assertion, the precondition of `Fill` holds (that is, the complaint about violating the precondition has been replaced by a complaint about violating the assertion). This confirms that we have written the condition correctly in the assertion. By now having the condition `w.Valid()` in front of our eyes, we can more easily think about how it may be failing.

The validity of the water tank `w` should follow from the validity of the coffee maker. That is, we expect `w.Valid()` to hold on entry to the method. Let's run that by the verifier:

```
assert w.Valid();  
g.AddBeans();  
assert w.Valid(); // assertion violation  
w.Fill(); // modifies violation
```

The verifier proves `w.Valid()` at the beginning of the method, but it's not able to prove it after the call to `AddBeans`. From this, it appears that the call to `AddBeans` has an effect on `w.Valid()`. That's surprising, because we expect the grinder to operate independently of the water tank.

At this point, read and write frames come into play. The call to `AddBeans` can modify the state of the objects in its **modifies** clause, namely the objects in `g.Repr`. The `WaterTank` function `Valid` depends on the state of the objects in its **reads** clause, namely the objects in `w.Repr`. So, the only way that `g.AddBeans()` can affect the value of `w.Valid()` is if there is overlap between `g.Repr` and `w.Repr`. Let's ask the verifier about this:

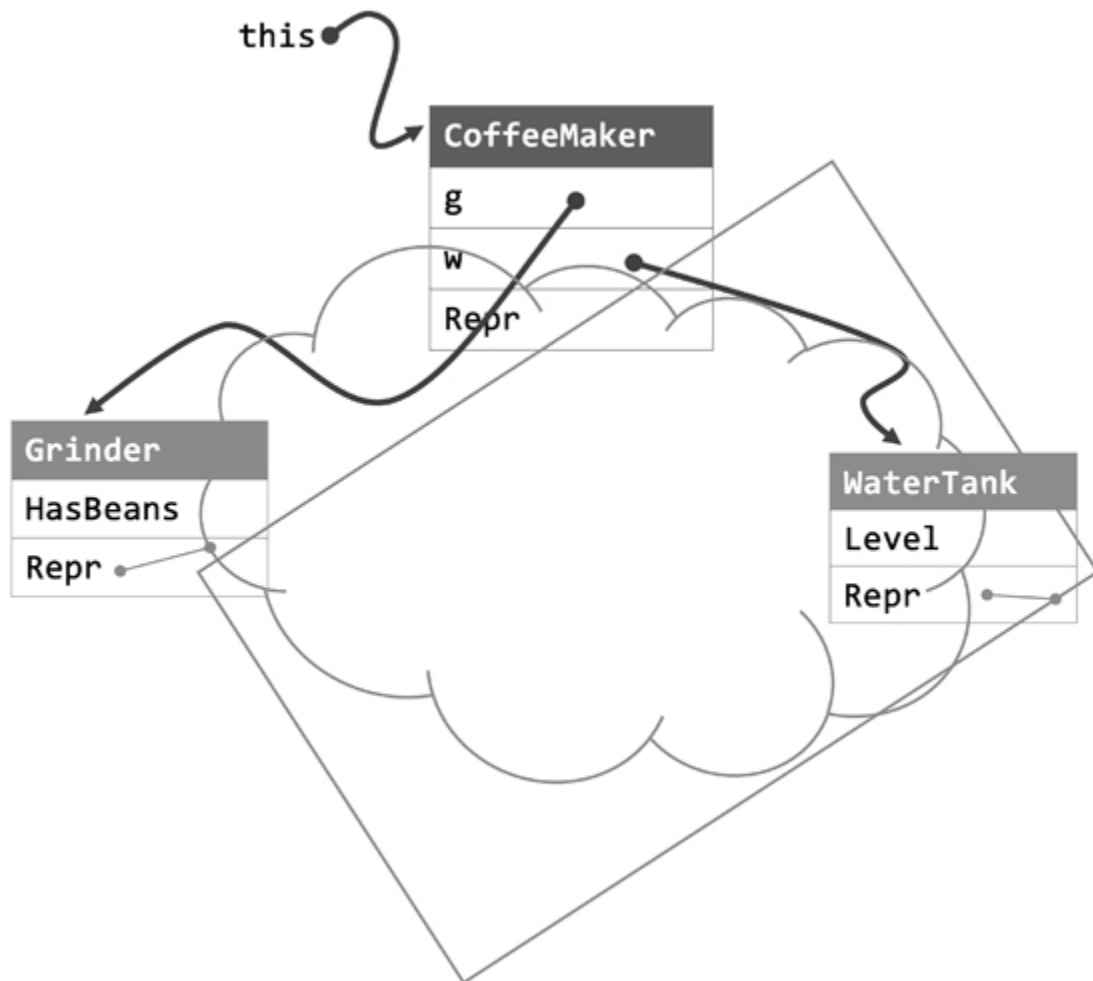
```
assert w.Valid();  
assert g.Repr !! w.Repr; // assertion violation  
g.AddBeans();  
assert w.Valid(); // assertion violation  
w.Fill(); // modifies violation
```

The operator `!!` (whose binding power is just higher than `&&` and `||`) says that its set arguments are disjoint.

The new assertion fails, showing that something we expect to always hold is not yet part of our specifications. Let's add this disjointness condition to the invariant of `CoffeeMaker`. That is, let's conjoin `g.Repr !! w.Repr` to the body of `Valid` in class `CoffeeMaker`.

Distressingly, the disjointness condition we added still does not let us prove `w.Valid()` upon return from `g.AddBeans()`. A visual may help us think this through. The following sketch shows the three objects we know about: a

coffee maker, a grinder, and a water tank. It depicts the set `g.Repr` by a cloud and the set `w.Repr` by a rectangle.



What do we actually know about the sets `g.Repr` and `w.Repr`? If you add the following assertions at the beginning of method `Restock`, you'll see how little we have specified:

```
assert this !in g.Repr; // assertion violation
assert g in g.Repr; // assertion violation
assert w !in g.Repr; // assertion violation
```

In our mental model of `g.Repr`, all three of these assertions hold, but we have neglected to reflect any of these in our specifications. This is regrettably common—it is easy to have in mind some property that we take for granted, but if the property is not recorded in our invariants or other specifications, then we can't make use of them in proofs. In

addition to the disjointness property we figured out above, we'd like to embellish our specifications to make the three assertions above provable.

The property `this !in g.Repr` is needed if we want to rely on the value of `this.w` not changing when we call `g.AddBeans()`. This property is best incorporated into the invariant of `CoffeeMaker`. The body of `Valid()` in class `CoffeeMaker` then looks like:

```
    this in Repr &&  
    g in Repr && g.Repr <= Repr && this !in g.Repr && g.Valid()  
&&  
    w in Repr && w.Repr <= Repr && this !in w.Repr && w.Valid()  
&&  
    g.Repr !! w.Repr
```

Sidebar 16.1

For two sets A and B , the expression $A \text{ !! } B$ is equivalent to saying the intersection of A and B is empty, that is,

$$A * B == \{\}$$

However, in addition to being more concise, the operator `!!` has the advantage of accepting any number of arguments, not just two. As such, it expresses that all of its arguments are pairwise disjoint. For example, if C is also a set, then you can write

$$A \text{ !! } B \text{ !! } C$$

to say that the three sets are pairwise disjoint. To write the same condition using intersections, you would write

$$A * B == \{\} \ \&\& \ B * C == \{\} \ \&\& \ C * A == \{\}$$

I've said before in words that an object is always part of its own representation set. In the bodies of validity predicates, we have therefore written **this in Repr**. However, if the enclosing module's export set hides the body of `Valid` from clients, then clients cannot use this property. To remedy this situation, let's go back to the validity predicates of classes `Grinder` and `WaterTank` in Section 16.2.0 and rewrite them as

```
ghost predicate Valid()  
  reads this, Repr  
  ensures Valid() ==> this in Repr
```

The mention of `Valid()` in its own postcondition refers to the value returned by the function (see Chapter 6). The body of `Valid` is still hidden from clients, but the postcondition is visible. From now on, this postcondition will be part of our validity predicates, so let's immediately add it to `Valid` in `CoffeeMaker` as well.

Exercise 16.10.

An easy typo is to write **ensures this in Repr** in the validity predicate. What would it mean and how is it different from the postcondition I wrote above?

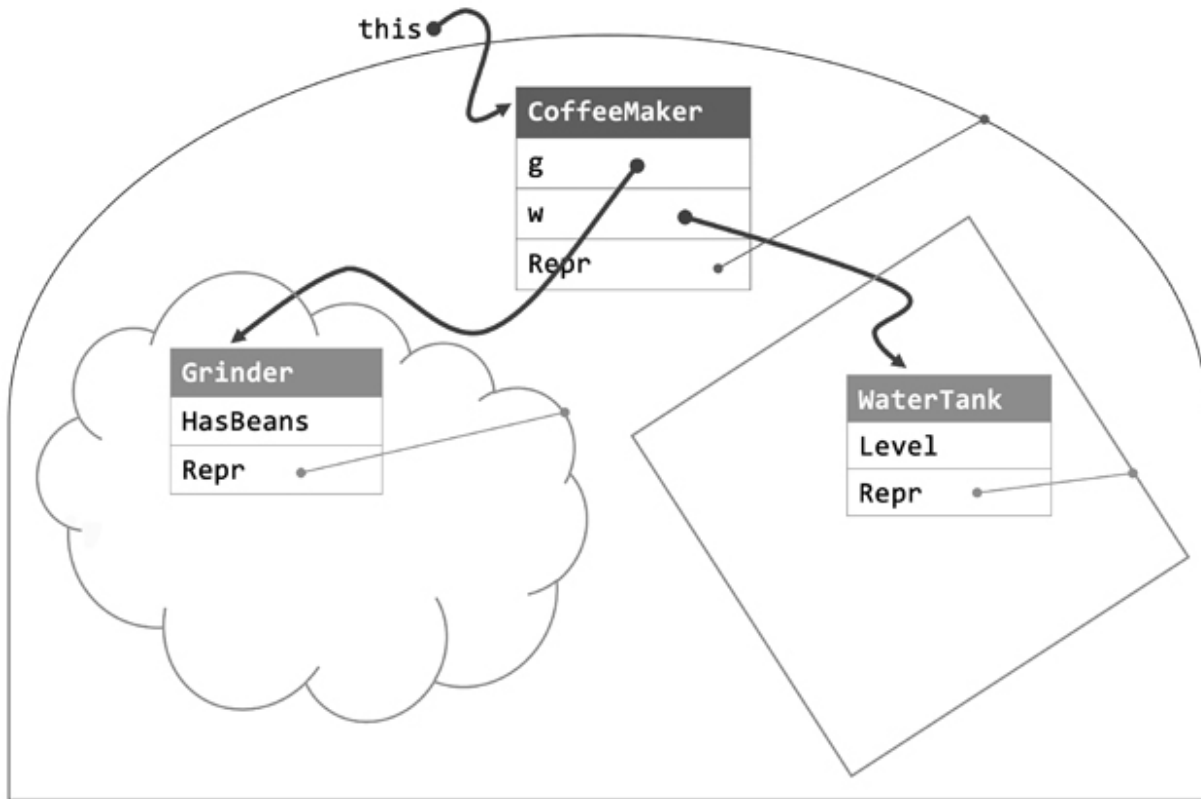
With the new postcondition for validity predicates, the invariant

```
this !in g.Repr
```

for any constituent object `g`, and the disjointness invariant

```
g.Repr !! w.Repr
```

for distinct constituent objects `g` and `w`, we can now depict our objects more precisely:



As this sketch seeks to convey, we now know for sure that **this** is not in `g.Repr` or `w.Repr` (because of the CoffeeMaker invariant), we know that `g` is in `g.Repr` and `w` is in `w.Repr` (because of the postcondition of the validity predicates in Grinder and WaterTank, and the fact that `g.Valid()` and `w.Valid()` are part of the CoffeeMaker invariant), and we know that `g.Repr` and `w.Repr` do not overlap (because of the disjointness invariant in CoffeeMaker). In this sketch, I also depicted `this.Repr`, which contains **this** as well as `g.Repr` and `w.Repr`.

What do these improvements do for our proof of Restock? Here's what we have at the moment:

```
method Restock()
  requires Valid()
  modifies Repr
  ensures Valid() && fresh(Repr - old(Repr)) && Ready()
{
  g.AddBeans();
  w.Fill();
}
```

```
} // postcondition violation
```

The postcondition that is not proved is `Valid()`; more specially, the verifier highlights the subset conditions `g.Repr <= Repr` and `w.Repr <= Repr` as not being provable. But, of course! If the calls to `AddBeans` and `Fill` have the effect of expanding `g.Repr` and `w.Repr`, then we need to expand `this.Repr` as well. This is easily done by adding

```
Repr := Repr + g.Repr + w.Repr;
```

as the last thing in the body of `Restock`. And with that, `Restock` finally verifies!

16.3.4. Updating Dispense

After the work we did to specify the relationships between the various frames, the verification of method `Dispense` presents us with just one problem. And it's a familiar problem: the representation-set subset relations are not maintained. The fix is the same as for `Restock`: we need to update `Repr` to reflect any changes of `g.Repr` and `w.Repr`. With the assignment

```
Repr := Repr + g.Repr + w.Repr;
```

we have completed the verified implementation of `CoffeeMaker`.

Exercise 16.11.

Update the `ChangeGrinder` and `InstallCustomGrinder` methods from Section 16.2.4 and method `RemoveGrinder` from Exercise 16.8 to work with the new version of `CoffeeMaker` we developed in this section.

16.4. Summary

A lot of details go into writing specifications for aggregate objects. I've tried to motivate each part of the specifications. The good news is that the pieces I've shown

occur in similar forms in other aggregate objects. In fact, the specifications so much follow the same form that they can be called idiomatic.

Let me summarize these idiomatic dynamic-frames specifications by describing the typical way to write specifications for aggregate objects. Your own classes may need to use variations of this standard idiom.

16.4.0. Representation set

A class uses a ghost variable to store its representation set:

```
ghost var Repr: set<object>
```

16.4.1. Invariant

The class also defines, also as a ghost, a validity predicate that states the object invariant. It has the form

```
ghost predicate Valid()  
  reads this, Repr  
  ensures Valid() ==> this in Repr  
{  
  this in Repr &&  
  // ...  
}
```

For every field *a* holding a constituent object with a simple frame, the validity condition also includes

```
a in Repr && a.Valid()
```

and for every field *b* holding a constituent object with a dynamic frame, the validity condition also includes

```
b in Repr && b.Repr <= Repr && this !in b.Repr && b.Valid()
```

In addition, various disjointness conditions are part of the validity predicate. For constituent objects *a0* and *a1* with simple frames and constituent objects *b0* and *b1* with dynamic frames, the disjointness conditions can be expressed by

```
a0 != a1 &&
```

```
{a0, a1} !! b0.Repr !! b1.Repr
```

16.4.2. Constructor

The constructor establishes the object's validity and the freshness of the representation set:

```
constructor ()  
  ensures Valid() && fresh(Repr)
```

For constituent objects *a* and *b* with a simple frame and a dynamic frame, respectively, the body of the constructor initializes *Repr* by

```
Repr := {this, a, b} + b.Repr;
```

This assignment is written in the second phase of the constructor, that is, after **new**;

16.4.3. Functions

A function (other than *Valid*, which defines *Repr* for valid objects) is defined with the following precondition and read frame:

```
function F(x: X): Y  
  requires Valid()  
  reads Repr
```

16.4.4. Methods

A *mutating method* has *Repr* as its write frame, requires and maintains validity, and also ensures that any objects added to *Repr* are newly allocated:

```
method M(x: X) returns (y: Y)  
  requires Valid()  
  modifies Repr  
  ensures Valid() && fresh(Repr - old(Repr))
```

Sometimes, a method is used (instead of a function) to compute some results without modifying the state of the

aggregate object. For such *querying methods*, the `modifies` and `ensures` clauses are dropped.

Exercise 16.12.

Follow the development of the programs in this chapter by writing them in the IDE to see what errors are produced in the places where our specifications or code were insufficient. Doing so will help you familiarize yourself with the particular error messages that are reported, and this will help you better know how to respond to similar error messages in your own programs.

Notes

Dynamic frames were invented by Yannis Kassios [69]. The first mechanical verifier to support them used ghost functions to capture representation sets [116]. The `dynamicframes` idiom in Dafny instead uses a ghost field (`Repr`). This burdens the programmer with having to explicitly update the ghost field, but it provides the automated verifier with better clues about when the values of abstraction functions (like `Valid()`) are unaffected by unrelated state updates.

There are alternative techniques for dealing with programs that update the heap. A survey paper on behavioral specifications [59] classifies such techniques according to how they approach representation sets:

- *Explicit* representation sets are programmer-defined. Examples include dynamic frames and the Dafny tool.
- *Implicit* representation sets are derived from predicates in specialized logics. Examples include separation logics [114, 103] and the VeriFast [64] and Viper [98] tools.
- *Predefined* representation sets are derived from predefined heap topologies (commonly, a tree

topology). Examples include the *ownership* relations in Spec# [83] and VCC [30], as well as the regions in WhyML [20].

Explicit representation sets provide flexibility in specifications (as we'll see in the iterator case study in Section 17.3 where representation sets may overlap). Another benefit of dynamic frames is that they don't require much from the language and logic—the idiom in this book only needs **modifies** and **reads** clauses, sets of objects, and ghost variables.

On the other hand, more specialized techniques can reduce clutter. For example, the boilerplate specifications (`Repr`, `Valid()`, ...) we needed in the `CoffeeMaker` example in Section 16.3 are in Spec# typically replaced by a simple **invariant** declaration for the class and a `[Rep]` annotation for each field holding a constituent object [83]. As another example, techniques that use implicit representation sets do not need **modifies** clauses and **fresh** predicates. Also, a technique that keeps track of read and write effects more strictly is often a better match for concurrency (see, e.g., [25, 84, 65]). See the survey paper for more information [59].

In this (and the next) chapter, and also back in Chapter 10, we recorded datastructure invariants using validity predicates. This makes it clear where the invariants are expected to hold. The approach is also flexible, because to specify auxiliary methods where the invariant does not hold on entry or exit, one simply omits the validity predicate from the specification. This approach has its roots in the *state/validity paradigm* of the Extended Static Checker for Modula-3 [39].

For common cases, having a dedicated **invariant** declaration in a class is more straightforward. Many languages, including Eiffel [89], JML [66], and Spec# [83], support such **invariant** declarations directly. Alas, in the

presence of aggregate objects, the simplicity of such **invariant** declarations can be deceptive (see, e.g., [[97](#), [112](#)]).

Chapter 17

Dynamic Heap Data Structures



In this chapter, I'll take you through four case studies where we'll specify and verify programs with dynamically allocated data structures. We'll see some aggregate objects whose representation sets may change over the lifetime of the data structure. Two examples implement arrays with special properties, and the other two implement a map with integer keys and an iterator over such maps.

17.0. Lazily Initialized Arrays

Getting and setting an element at a given index of an array can be done in constant time. Allocating the storage needed for an array can also be done in constant time, if the memory allocator has a contiguous area of unused memory and returns the next portion of it. But what about creating an array (say, of size n) and initializing every element to a given element (say, d)? The most straightforward way to do that uses time linear in n . However, with some additional storage, creating an initialized array can also be done in constant time. That's what we will do in this section. More precisely, we will build an abstraction of an array, where initialized creation, element get, and element update each uses only a constant number of integer operations and a constant number of uninitialized-array allocations.

17.0.0. Basic specification

Here is the specification of the class we will implement:

```
class LazyArray<T(0)> {
  ghost var Elements: seq<T>
  ghost var Repr: set<object>
  ghost predicate Valid()
    reads this, Repr
    ensures Valid() ==> this in Repr
  constructor (length: nat, initial: T)
    ensures Valid() && fresh(Repr)
    ensures |Elements| == length
    ensures forall i :: 0 <= i < |Elements| ==>
      Elements[i] == initial
  function Get(i: int): T
    requires Valid() && 0 <= i < |Elements|
    reads Repr
    ensures Get(i) == Elements[i]
  method Update(i: int, x: T)
    requires Valid() && 0 <= i < |Elements|
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr))
    ensures Elements == old(Elements)[i := x]
}
```

This class declaration follows the same basic structure that I led up to by the end of the previous chapter, but several details are noteworthy.

One detail is the `Elements` field. Like the `Repr` field that we have used here and in the previous chapter, `Elements` is an *abstraction field*, which plays the same role as the abstraction functions in Sections 9.3.1 and 10.0.0. The field is used in specifications as a way to understand what the class does. Because the field is ghost, the class is free to implement the array storage in some other way. We could define `Elements` as a function, but since we're dealing with mutable data structures, defining it as a field often makes it easier to prove that unrelated mutations of the heap don't affect `Elements`.

A second detail is that the constructor takes a parameter `initial` and ensures every element of `Elements` to be `initial`. In other words, the ghost field `Elements` is eagerly initialized to all-`initial` elements.

The third detail is that the postcondition of `Update` uses a *sequence update* expression. For any index `i` into a sequence `q` and any value `x` of the same type as the elements of `q`, the expression `q[i := x]` denotes the sequence that is like `q` except that for index `i` it returns `x`.

Finally, the strangest detail is that the class takes a type parameter `τ` that is marked with the type characteristic `(0)`. This limits instantiations of `τ` to types where the compiler knows how to create a value for the type. Such types are known as *auto-init types*. For the purpose of the present example, you really just need to know that by requiring `τ` to be an auto-init type, we can create an array of `τ` values without having to explicitly initialize these elements (which would take linear time, and we're shooting for a constant-time creation of our `LazyArray`).

Exercise 17.0.

Write a module around the class, and define an appropriate export set for the module.

17.0.1. Test harness

As usual, to test the credibility of our specification, we'll write a small test harness. The tests performed by the following method exercise the initialization provided by the constructor, and check that more than one `Update` is possible without interference between distinct elements, and without interference with a second `LazyArray` object.

```
method LazyArrayTestHarness() {  
  var a := new LazyArray(300, 4);  
  assert a.Get(100) == a.Get(101) == 4;  
  a.Update(100, 9);  
  a.Update(102, 1);  
  assert a.Get(100) == 9 && a.Get(101) == 4 && a.Get(102) == 1;  
  
  var b := new LazyArray(200, 7);  
  assert a.Get(100) == 9 && b.Get(100) == 7;  
  a.Update(100, 2);  
  assert a.Get(100) == 2 && b.Get(100) == 7;  
}
```

It verifies, so this gives us some confidence in our specification.

17.0.2. Immutable state

Before we go on with the implementation, let me take a minute to introduce a useful specification technique. Our class only has one method, `Update`, and you can easily see from its specification that the length of `Elements` never changes. If we want to, we can introduce a name for this length, say `N`. Doing so lets us abbreviate the expression `|Elements|` to just `N`. Since the length never changes, we can declare `N` to be an immutable field:

```
ghost const N: nat
```

This makes it immediately clear that the length never changes, and it eliminates the need to ever write a specification like

```
|Elements| == |old(Elements)|
```

The connection between `N` and `|Elements|` is stated as an object invariant, that is, as a conjunct in the body of the validity predicate. We want to tell clients about this connection, but we want to keep other details of the body of `Valid` private, so we also put this condition in a postcondition of `Valid`:

```
ghost predicate Valid()  
  reads this, Repr  
  ensures Valid() ==> this in Repr && N == |Elements|
```

So, as you see here, introducing an immutable ghost field and equating it to an expression is a handy way to specify that the expression never changes.

While we're tweaking the specification and thinking about `const`, let's make one more change. As it turns out, the representation set we'll use for `LazyArray` is determined at creation and never changes thereafter. Thus, we can also declare `Repr` as immutable:

```
ghost const Repr: set<object>
```

This saves us specifications about freshness of `Repr` in methods, at the expense of committing not to change the representation set (see Section 16.2.4).

Exercise 17.1.

Update the export set in Exercise 17.0 to accommodate the changes we just made to the class.

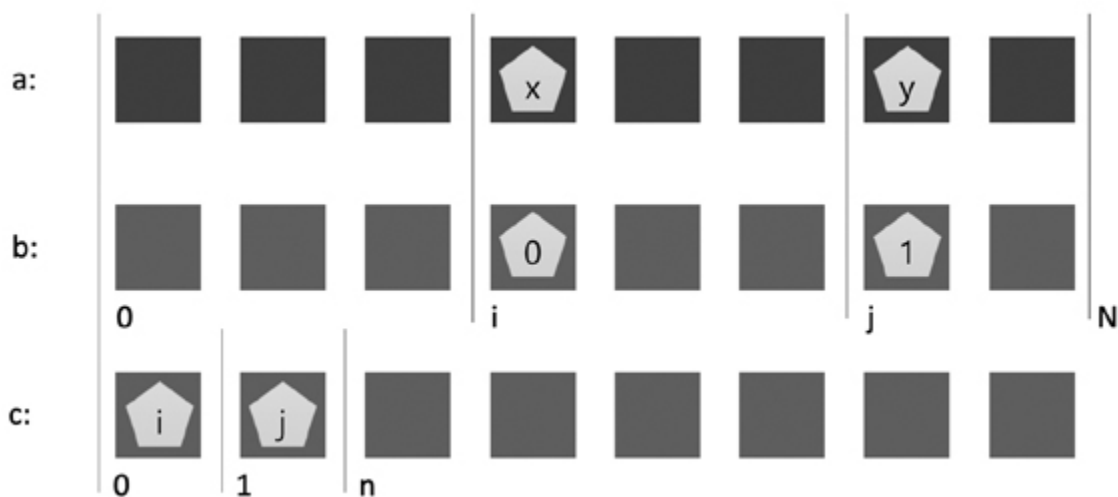
17.0.3. Underlying data structure

We'll use one underlying array, `a`, to store the values of those elements that have been set via `Update`. Since the constructor does not have time to initialize the elements of

a, we'll instead need some cleverness to implement Get in constant time.

Here's what we'll do. In addition to a, we'll use two arrays of integers, b and c, and one integer, n. The integer n keeps track of how many distinct elements have been set by Update, and it also says how many of the leftmost elements of c have been initialized. To record that element i has been set, we'll arrange to set b[i] in the range from 0 to n (that is, b[i] will hold the index of an element of c that we have initialized) and that element in c stores the value i. If Get is ever asked to return the value at an uninitialized index in a, we'll return the initial value passed to the constructor, which we'll store in a field default.

The following diagram illustrates:



With n being 2, the diagram shows a state where 2 elements of a have been set: a[i] is x and a[j] is y. The corresponding elements in b contain indices into c: b[i] is 0 and b[j] is 1. The elements at those indices in c refer back to the respective elements of b: c[0] is i and c[1] is j. Thus, we have $c[b[i]] == i$ and $c[b[j]] == j$.

The situation I described cannot happen by accident among our uninitialized values. If a[k] has not been initialized, then we haven't initialized b[k], either. If the value of b[k] is not in the range from 0 to n, then it's clear

that `b[k]` hasn't been initialized, which tells us `a[k]` hasn't been initialized. And if the uninitialized value of `b[k]` happens to contain the index of an initialized element of `c`, we can still detect that `b[k]` wasn't initialized, because `c[b[k]]` will not be `k`.

Here are the fields in our implementation:

```
const default: T
const a: array<T>
const b: array<int>
const c: array<int>
var n: int
```

Here is the predicate that lets us determine if an index `i` has been initialized:

```
predicate IsInitialized(i: int)
  requires 0 <= i < |Elements| == b.Length == c.Length
  requires 0 <= n <= |Elements|
  reads this, b, c
{
  0 <= b[i] < n && c[b[i]] == i
}
```

Notice that I have defined this predicate without the standard

```
requires Valid()
reads Repr
```

for functions that clients will see. Instead, the precondition is written to include just those conditions that are needed for `IsInitialized` to be well-defined. Similarly, the read frame of `IsInitialized` is given directly in terms of the implementation objects. This is all well and good, since `IsInitialized` is a part of the implementation and not given out to clients.

In fact, we *have* to declare the predicate without mentioning `Valid()`, because we're going to use `IsInitialized` in the definition of `Valid()`. The central part of the object invariant is the following connection between our many fields:

```
forall i :: 0 <= i < N ==>
  Elements[i] == if IsInitialized(i) then a[i] else default
```

If `Valid` mentions `IsInitialized` in this way, then we can't also let `IsInitialized` mention `Valid`, because then we would not be able to prove termination among the mutually dependent predicates.

17.0.4. Object invariant

The invariant of a `LazyArray` object involves many conditions, both data-structure invariants and frame invariants. As usual, we record these as the body of `Valid()`. The frame invariants say that arrays `a`, `b`, and `c` are part of the representation set and are distinct:

```
ghost predicate Valid()
  reads this, Repr
  ensures Valid() ==> this in Repr && N == |Elements|
{
  this in Repr &&
  a in Repr && b in Repr && c in Repr &&
  a != b && b != c && c != a &&
```

Then, we state that the array lengths coincide with the length of the elements in the `LazyArray`, and we specify `N` to be this length:

```
N == |Elements| == a.Length == b.Length == c.Length &&
```

The length of the prefix of `c` that is defined, `n`, is in the range from 0 through `N`:

```
0 <= n <= N &&
```

And, finally, there's the central part of the object invariant that connects the abstraction field `Elements` with the compiled fields:

```
(forall i :: 0 <= i < N ==>
  Elements[i] == if IsInitialized(i) then a[i] else
default)
}
```

17.0.5. Implementation

The implementation of the constructor is straightforward:

```
constructor (length: nat, initial: T)
  ensures Valid() && fresh(Repr)
  ensures N == length && forall i :: 0 <= i < N ==>
    Elements[i] == initial
{
  N := length;
  Elements := seq(length, _ => initial);
  default := initial;
  a, b, c := new T[length], new int[length], new int[length];
  n := 0;
  Repr := {this, a, b, c};
}
```

Given our central invariant, the implementation of Get is also straightforward:

```
function Get(i: int): T
  requires Valid() && 0 <= i < N
  reads Repr
  ensures Get(i) == Elements[i]
{
  if IsInitialized(i) then a[i] else default
}
```

Method Update also uses predicate IsInitialized, and it needs to maintain the central invariant. With some luck, this method is as straightforward to implement as the constructor and Get:

```
method Update(i: int, x: T)
  requires Valid() && 0 <= i < N
  modifies Repr
  ensures Valid() && Elements == old(Elements)[i := x]
{
  if !IsInitialized(i) {
    b[i], c[n] := n, i; // index out of bounds
    n := n + 1;
  }
  a[i] := x;
  Elements := Elements[i := x];
}
```

Notice that `Update` changes not only the compiled data structure, but it also changes the abstraction field `Elements`. This is necessary to maintain the object invariant and establish the postcondition.

Luck or not, method `Update` does not verify as given above. The verifier complains about a possible index out-of-bounds in `c[n]`. The object invariant readily tells us $0 \leq n \leq N == c.Length$, but that leaves the possibility that `n` might equal `c.Length`. We need to prove—to ourselves and to the verifier—that $n < N$ holds when there is an index `i` corresponding to a lazily initialized element.

Here's how we deal with this situation. First, we formulate a lemma that states the property we need (and which we think holds):

```
lemma ThereIsRoom(i: nat)
  requires Valid() && 0 <= i < N && !IsInitialized(i)
  ensures n < N
```

Second, we call this lemma from `Update`, just before setting `c[n]`:

```
if !IsInitialized(i) {
  ThereIsRoom(i);
  b[i], c[n] := n, i;
```

This eliminates the verifier's index-out-of-bounds complaint. Third, we prove the lemma.

Prove the lemma. Yeah, about that...

17.0.6. Proving there is room

Convincing ourselves there's room in `c` (that is, that $n < c.Length$) whenever there's an uninitialized element in `a` (that is, `!IsInitialized(i)` holds for some `i`) is not difficult. The basic reasoning goes like this (where, syntactically in the expressions, the two outermost vertical-bar brackets denote set cardinality, and the middle vertical-bar character is part of the set-comprehension expression):

```

== // n equals the number of initialized elements in a
  |set j | 0 <= j < N && IsInitialized(j)|
< // !IsInitialized(i), so i is in the set below, but not
  // the one above
  |set j | 0 <= j < N|
== // this set has N elements
  N

```

Unfortunately, the verifier needs more help for each of these steps. Let's address each step. I'll do them in reverse order from how they occur above.

Sets of natural numbers

The easiest way to prove that an integer set `set j | 0 <= j < k` has k elements is to construct it inductively. This can be done with a recursive function. I suggest writing this function at the outermost level of the module, outside class `LazyArray`, but you can also write it as a member of the class. (Better yet would be to collect this and similar functions and lemmas into a library module of useful set properties.)

```

ghost function Upto(k: nat): set<int>
  ensures forall i :: i in Upto(k) <==> 0 <= i < k
  ensures |Upto(k)| == k
{
  if k == 0 then {} else Upto(k - 1) + {k - 1}
}

```

The first postcondition of the function talks about membership of the result set, and the second postcondition talks about the cardinality of the set.

Cardinalities of proper subsets

The verifier knows only a limited number of properties about sets, so we should expect to be helping it along. The property we need is that if a set u is a proper subset of a set U , then $|u| < |U|$. We'll state and prove it as follows:

```

lemma SetCardinalities(u: set<int>, U: set<int>, x: int)
  requires u <= U
  ensures |u| <= |U|

```

```

ensures x !in u && x in U ==> |u| < |U|
{
  if u == {} {
  } else {
    var y :| y in u;
    SetCardinalities(u - {y}, U - {y}, x);
  }
}

```

The assign-such-that statement `y :| y in u` (see Section 13.7.5) sets `y` to an arbitrary value satisfying the predicate `y in u`.

Cardinality of set of initialized elements

To correlate `n` with the cardinality of the set of initialized elements, we can let each `LazyArray` object maintain this set as a ghost field and to state an invariant about its cardinality. Then, all we need to do is change this set and `n` together.

In the class, we'll introduce a field `s` for this purpose:

```
ghost var s: set<int>
```

To describe the value of `s` and to connect its cardinality with `n`, we add two conjuncts to the validity predicate:

```

s == (set i | 0 <= i < N && IsInitialized(i)) &&
n == |s|

```

To establish these conditions initially, we replace the assignment `n := 0;` in the constructor with

```
s, n := {}, 0;
```

And to maintain the two conditions as an object invariant, we replace the assignment `n := n + 1;` in method `Update` with

```
s, n := s + {i}, n + 1;
```

Proving the lemma

With these three ingredients in place, we write the proof of the `ThereIsRoom` lemma:

```
lemma ThereIsRoom(i: nat)
```

```

requires Valid() && 0 <= i < N && !IsInitialized(i)
ensures n < N
{
  var S := Upto(N);
  SetCardinalities(s, S, i);
}

```

That completes the development and proof of the LazyArray class.

17.1. Extensible Array

In this section, we'll write a class that represents an extensible array. Beyond construction, the class supports the usual get and update element operations as well as an operation Append that increases the length of the array and puts a given element in the new array slot. We will design the Append operation not to copy any previous array elements. This will make the running time of each operation of our class logarithmic in the number of elements stored.

17.1.0. Specification

We define a class ExtensibleArray, parameterized by the element type. Here is the client's view of the class, using the standard specification idiom with one abstract field (Elements):

```

class ExtensibleArray<T> {
  ghost var Elements: seq<T>
  ghost var Repr: set<object>
  ghost predicate Valid()
    reads this, Repr
    ensures Valid() ==> this in Repr
  constructor ()
    ensures Valid() && fresh(Repr) && Elements == []
  function Get(i: int): T
    requires Valid() && 0 <= i < |Elements|
    ensures Get(i) == Elements[i]

```



```

    reads Repr
method Update(i: int, t: T)
    requires Valid() && 0 <= i < |Elements|
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr))
    ensures Elements == old(Elements)[i := t]
method Append(t: T)
    requires Valid()
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr))
    ensures Elements == old(Elements) + [t]
}

```

Notice that the method postconditions use the standard `fresh(Repr - old(Repr))` since we expect to change the representation set over time.

Exercise 17.2.

Write a test harness for the `ExtensibleArray` class.

17.1.1. Data structure

The basic idea is to store the extensible array's elements in fixed-length arrays. I'll use 256 as the length of each of these arrays. There is a “front” array, where the most recently appended elements are placed. Once the front array is filled up, it gets placed in “the depot”, which is an extensible collection of arrays. As the data structure for this depot, we'll use an extensible array of arrays! That is, an `ExtensibleArray<T>` may contain an `ExtensibleArray<array<T>>`. The diagram in Figure 17.0 illustrates our design.

There are several important decisions to be made about our design, and it's best to give these some thought before we get started writing the object invariant, let alone the implementation. Deciding among design alternatives may require some trial and error, so as you develop your own programs, you may be changing declarations and invariants

several times before settling on something you're happy with.

Nullable types

One decision (which is common enough to warrant its own section heading) concerns how to represent an empty depot. Upon construction of an `ExtensibleArray<T>`, we would have trouble constructing an `ExtensibleArray<array<T>>` object for the depot, because that would make the `ExtensibleArray` constructor recursive. We stand no chance of finding a termination metric for this recursion, because if every constructor invocation makes a recursive call, then we would really have infinite recursion on our hands.

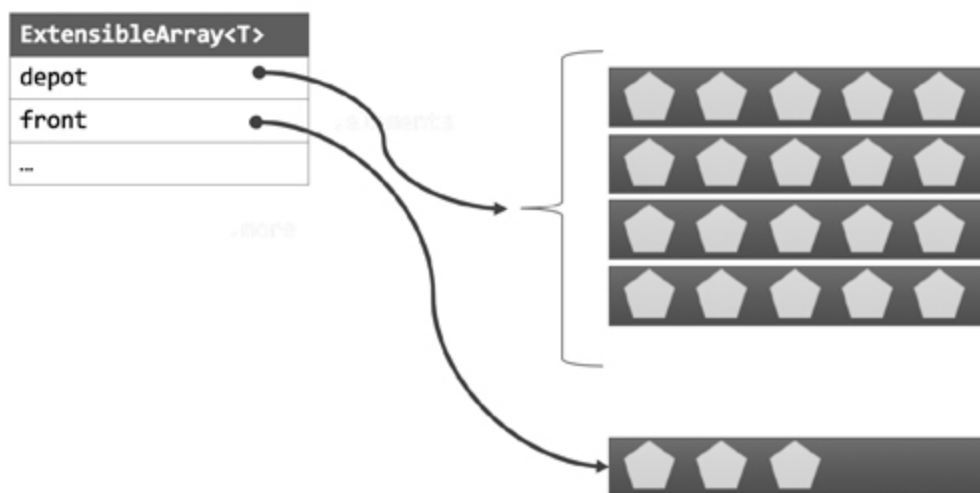


Figure 17.0. Data structure of the extensible array.

Instead, let's construct the `ExtensibleArray<array<T>>` object for the depot when we really need it. The type of our depot field would then need to be something that avoids the need to allocate an object. For example, we could declare the type to be

```
Option<ExtensibleArray<array<T>>>
```

where `Option` is the following datatype:

```
datatype Option<A> = None | Some(A)
```

There is no need to declare such a type, however, because every class `c` in Dafny already gives rise to *two* types: a type named `c` and a type named `c?`. A value of type `c` is a reference to an instance of class `c`. We've used this type for (array and object) references throughout the chapters of this Part of the book. A value of type `c?` is either such a reference or the special value `null`.

Note that the `?` is part of the name of the type—it is not an operator applied to a type. What I have said here about classes also holds for array types. For example, a value of type `array?<T>` is either a reference to a `T`-array or the special value `null`.

So, by declaring `depot` to be of type

```
ExtensibleArray?<array<T>>
```

we can initialize the field to `null` and not have to call the constructor recursively.

Other design decisions

A second decision to make in our design of `ExtensibleArray` is whether to make the front a part of the depot or separate from the depot. Let's make it separate. This seems a little cleaner, since it means every array in the depot has a full 256 elements. It also means we don't need to create the depot object until we've filled the very first array.

An important edge case to make a design decision about is whether to add front to the depot immediately when it has 256 elements, or to wait until we get another `Append` request. Let's do the former. This means the front is never full, whereas every array in the depot is full.

A third design decision is when to allocate the front array. Let's allocate it when it needs to contain elements.

When the front array is not needed, we'll make its field `null`.

Here are the declarations of the fields for the front array and the depot:

```
var front: array?<T>
var depot: ExtensibleArray?<array<T>>
```

We'll use a field to store the total number of elements in the extensible array:

```
var length: int
```

It will also be convenient to have a field that contains the number of elements stored in the depot, because that will let us decide between looking in the front array or the depot:

```
var M: int // shorthand for:
              // if depot == null then 0 else 256 *
|depot.Elements|
```

17.1.2. Object invariant

Having made those design decisions, let's write the object invariant.

Frame invariants

We'll start with the frame invariants. As always, **this** is in the representation set, so we start filling in the body of `Valid()` like this:

```
{
  this in Repr &&
```

The front array is in the representation set, unless it's `null`. And when it is not `null`, its length is always 256:

```
(front != null ==>
  front.Length == 256 && front in Repr) &&
```

There's much more to say about the depot. First, the standard frame inclusion properties:

```
(depot != null ==>
```

```

depot in Repr && depot.Repr <= Repr &&
this !in depot.Repr && depot.Valid() &&

```

The representation sets of `front` (which is just itself) and `depot` (which is `depot.Repr`) are disjoint:

```

front !in depot.Repr &&

```

If you don't include all of these frame invariants, your verification is not likely to succeed. Therefore, it's a good idea to take an extra minute to think carefully about if you have included these standard parts and have thought about the various ways that representation sets can overlap (use Section 16.4 as a checklist).

We have a few more things to say about the arrays in the `depot`. Every such array has length 256:

```

forall j :: 0 <= j < |depot.Elements| ==>
  depot.Elements[j].Length == 256 &&

```

Moreover, every such array is in the representation set of the `ExtensibleArray`, but outside of the representation set of `depot`:

```

depot.Elements[j] in Repr &&
depot.Elements[j] !in depot.Repr &&

```

Finally, every such array is different from `front` and from all the other arrays in `depot`:

```

depot.Elements[j] != front &&
forall k :: 0 <= k < |depot.Elements| && k != j ==>
  depot.Elements[j] != depot.Elements[k]) &&

```

Invariants about length

We decided `M` denotes the total number of elements stored in the `depot`. We also decided `depot` is `null` when the `depot` has no elements. That gives us the following invariant:

```

M == (if depot == null then 0 else 256 * |depot.Elements|)
&&

```

The remaining `length - M` elements of the extensible array are stored in the `front` array. We decided it is `null` when it stores no elements, and we decided to move it into the

depot as soon as it would have 256 elements. That gives us the following two invariants:

```
M <= length < M + 256 &&
(length == M <==> front == null) &&
```

Invariants about elements

The number of elements in the extensible array is stored in length:

```
length == |Elements| &&
```

Of those elements, the first M are stored in the depot, 256 in each successive array:

```
(forall i :: 0 <= i < M ==>
  Elements[i] == depot.Elements[i / 256][i % 256]) &&
```

and the rest are stored in front:

```
(forall i :: M <= i < length ==>
  Elements[i] == front[i - M])
}
```

That is the end of the object invariant. Let's implement the operations.

17.1.3. Implementation

We have four operations to implement: the constructor, function Get, and methods Update and Append. We settled on the specifications of these operations in Section 17.1.0.

The constructor needs to initialize Elements to []. The invariant then tells us what values to assign the other fields:

```
{
  front, depot := null, null;
  length, M := 0, 0;
  Elements, Repr := [], {this};
}
```

Function Get looks in front if the given index is at least M. Otherwise, it finds the appropriate array in the depot and

indexes into it:

```
{
  if M <= i then
    front[i - M]
  else
    var arr := depot.Get(i / 256);
    arr[i % 256]
}
```

Here, we have a recursive call to `Get`, so we have to prove termination. If a function has a given **reads** clause, but no given **decreases** clause, then Dafny starts the default termination metric with the given read frame (followed by one lexicographic component for each function parameter, as we have seen before). So, the default **decreases** clause for `Get` is the lexicographic tuple `Repr, i` (as you can see in the Dafny IDE by the hover text over the name of the function). Since the representation set for `depot` is a strict subset of the representation set of **this**, the first of these lexicographic components decreases with the recursive call. In other words, Dafny proves the termination of `Get` automatically.

The implementation of method `Update` has the same structure as `Get`, but also updates `Elements`:

```
{
  if M <= i {
    front[i - M] := t;
  } else {
    var arr := depot.Get(i / 256);
    arr[i % 256] := t;
  }
  Elements := Elements[i := t];
}
```

Method `Append` wants to put the given element into `front`. But `front` could be `null`, in which case a new array is allocated:

```
{
  if front == null {
    front := new T[256](_ => t);
  }
```

```

    Repr := Repr + {front};
}

```

Since nothing is known about type parameter τ , Dafny requires us to give initial values for the elements of this new array. Luckily, method `Append` has such an element handy, namely `t`, so the method uses it to initialize every element of the new array. (An alternative would have been to declare type parameter τ with the auto-init characteristic (0), as we did with the type parameter of `LazyArray` in Section 17.0.0.)

With `front` being non-`null`, we put the new element in its place, and we update `length` and `Elements` accordingly:

```

front[length - M] := t;
length := length + 1;
Elements := Elements + [t];

```

Suppose we pause for a moment here and wait for the verifier's feedback. The feedback is that postcondition `Valid()` cannot be proved, and the verifier will highlight `length < M + 256` as the condition in `Valid()` that it cannot prove. Indeed, by increasing `length`, `length` may now equal 256. If so, we need to move `front` into the depot and then set `front` to `null`. Here is the test:

```

if length == M + 256 {

```

If the depot is empty, then `depot` is `null`, says our object invariant. So, we start by allocating `depot`, if necessary:

```

    if depot == null {
        depot := new ExtensibleArray();
        Repr := Repr + depot.Repr;
    }

```

The type parameter to `ExtensibleArray` is inferred to be `array<T>` (so the result of the `new` will match the declared type of `depot`), though you can supply it explicitly, if you'd rather see it in the program text.

With `depot` being non-`null`, we append the `front` array to the depot and adjust the remaining fields:


```

        depot.Append(front);
        Repr := Repr + depot.Repr;
        M := M + 256;
        front := null;
    }
}

```

We get one complaint from the verifier, namely that the recursive call to `Append` may not terminate. We need to supply a termination metric to prove termination. We might think of trying **decreases** `Repr`, since that worked so well for `Get`. However, it is not true that `depot.Repr` at the time of the recursive call is smaller than **this**.`Repr` on entry to `Append`. In particular, if `depot` was `null` on entry to `Append`, then the initial **this**.`Repr` is unrelated to the `depot.Repr` at the time of the call. Instead, we use a different termination metric:

```
decreases |Elements|
```

This does decrease, and so we have finished our implementation of `ExtensibleArray`.

17.1.4. Summary

The `ExtensibleArray` class is a minefield of subtle boundary conditions. In any maintainable software module, such conditions are documented. We started the design with an idea (shown in a diagram in Section 17.1.1). We then thought about design decisions we would need to make; for example, about when arrays could be full or empty or `null`. We documented the design decisions by writing them in the object invariant. From there, with relative ease and the help of the verifier, we wrote the code.

17.2. Binary Search Tree for a Map

In this section, we will implement a map from integers to values of some arbitrary type `Data`. The implementation uses

a *binary search tree*, which is a binary tree whose infix traversal yields a sorted list.

The example includes a `Lookup` function that returns the data corresponding to a given integer, if any. To signal whether or not it's returning data, `Lookup`'s result type uses a standard datatype type `Option`, defined as follows:

```
datatype Option<T> = None | Some(T)
```

17.2.0. Specification

As usual, we'll start by writing the specification, to make it clear what we're aiming to do. Here is the client's view of the class:

```
class BinarySearchTree<Data> {  
  ghost var M: map<int, Data>  
  ghost var Repr: set<object>  
  ghost predicate Valid()  
    reads this, Repr  
    ensures Valid() ==> this in Repr  
  constructor ()  
    ensures Valid() && fresh(Repr)  
    ensures M == map[]  
  function Lookup(key: int): Option<Data>  
    requires Valid()  
    reads Repr  
    ensures key in M.Keys ==> Lookup(key) == Some(M[key])  
    ensures key !in M.Keys ==> Lookup(key) == None  
  method Add(key: int, value: Data)  
    requires Valid()  
    modifies Repr  
    ensures Valid() && fresh(Repr - old(Repr))  
    ensures M == old(M)[key := value]  
  method Remove(key: int)  
    requires Valid()  
    modifies Repr  
    ensures Valid() && fresh(Repr - old(Repr))  
    ensures M == old(M) - {key}  
}
```

The `BinarySearchTree` class is parameterized by a type `Data`. Abstractly, an instance of the class represents a map from

integers to `Data` values. This map is declared as the ghost field `M`.

Following the standard specification idiom, the class declares ghost field `Repr` to hold the representation set and predicate `Valid()` to embody the object invariant. The first specification line of the constructor, the first two lines of function `Lookup`, and the first three lines of methods `Add` and `Remove` are boilerplate specifications that use `Repr` and `Valid()`.

The remaining lines of the specifications of the members are specific to the binary search tree. The constructor creates a tree that represents the empty map. Function `Lookup` returns the `Data` value corresponding to a given key, wrapped in the `Some` variant of its return type. If the given key is not in the map, `Lookup` returns `None`. Method `Add` adds or replaces the `Data` value associated with the given key. Method `Remove` removes any `Data` value associated with the given key, leaving all other entries unchanged. (The map domain subtraction expression $m - s$ denotes the map m with the keys from s removed.)

17.2.1. Implementation

The binary search tree is implemented by instances of another class, `Node`. To arrange the `Node` objects in a tree structure, the `Node` class declares `left` and `right` fields to further `Node` objects. Class `Node` needs a way to represent the absence of a left or right child, and class `BinarySearchTree` needs a way to represent the absence of any nodes at all. The easiest way to do this is to use the special value `null`, which is part of the type `Node?`.

As I explained in Section 17.1.1, for every class, like `Node`, Dafny introduces *two* types. One of these types is named `Node` and contains references to `Node` objects. The other type is named `Node?` and contains the values of `Node` as well as the special value `null`. We could use the type `Option<Node<Data>>`

to represent possible references to `Node<Data>` objects, but using `Node?<Data>` is slightly easier, because we don't have to explicitly call a destructor to obtain the reference.

Class `BinarySearchTree` declares a field that references the root node of a tree:

```
var root: Node?<Data>
```

As we'll soon see in Section 17.2.2, class `Node` has members `M`, `Repr`, and `Valid()`, just like `BinarySearchTree`. Using those, we write the object invariant of binary search trees:

```
ghost predicate Valid()
  reads this, Repr
  ensures Valid() ==> this in Repr
{
  this in Repr &&
  (|M.Keys| == 0 ==>
    root == null) &&
  (|M.Keys| != 0 ==>
    root in Repr && root.Repr <= Repr && this !in root.Repr
&&
    root.Valid() &&
    root.M == M)
}
```

Exercise 17.3.

Implement the constructor of class `BinarySearchTree`.

Given the definition of class `Node`, the implementations of members `Lookup`, `Add`, and `Remove` are straightforward. You'll find those as Exercises 17.5, 17.7, and 17.9 below. Next, we'll focus on class `Node`.

17.2.2. Node invariant

In addition to the `M` and `Repr` fields, which play roles analogous to those in `BinarySearchTree`, class `Node` declares fields to hold one key-value pair and possible references to children nodes:

```
class Node<Data> {
```

```

ghost var M: map<int, Data>
ghost var Repr: set<object>
var key: int
var value: Data
var left: Node?<Data>
var right: Node?<Data>

```

The most difficult part of class Node is to design its invariant, where it is easy to omit some parts by mistake. Let me show it here and then walk through its parts:

```

ghost predicate Valid()
  reads this, Repr
  ensures Valid() ==> this in Repr
{
  this in Repr &&
  (left != null ==>
    left in Repr && left.Repr <= Repr &&
    this !in left.Repr &&
    left.Valid() &&
    Less(left.M.Keys, {key})) &&
  (right != null ==>
    right in Repr && right.Repr <= Repr &&
    this !in right.Repr &&
    right.Valid() &&
    Less({key}, right.M.Keys)) &&
  (left != null && right != null ==>
    left.Repr !! right.Repr) &&
  M == Union(Union(map[key := value], left), right)
}

```

The bulk of this definition has to do with the structure of the aggregate Node object. These parts follow the standard specification idiom, as summarized in Section 16.4, but note that what is said about left is guarded by `left != null`, and similarly for right, and the disjointness condition `left.Repr !! right.Repr` is guarded by both left and right being non-`null`. I've mentioned this before, but it's worth saying again: be sure to look over these structural definitions twice, comparing them with Section 16.4 and thinking about the special needs of your data structure. Being reasonably sure that you have the right structural

definitions from the start will save you a lot of frustration later when you try to use and implement the class.

Apart from the structural parts of the invariant, there are 3 lines:

```
Less(left.M.Keys, {key})
Less({key}, right.M.Keys)
M == Union(Union(map[key := value], left), right)
```

The first two of these speak to the ordered property of a binary search tree. To write these conditions symmetrically for left and right, I have defined a predicate `Less(a, b)`, which says that every value in set `a` is less than every value in `b`:

```
ghost predicate Less(a: set<int>, b: set<int>) {
  forall x, y :: x in a && y in b ==> x < y
}
```

The final line of the Node validity condition describes the value of the map `M`. A node's `M` is the combination of `left.M`, `right.M`, and the maplet (singleton `map`) `map[key := value]`. To express this combination, I've defined a function `Union(m, n)`, which extends the map `m` with the map `n.M`, allowing `n` to be `null`:

```
ghost function Union<Data>(m: map<int, Data>,
                           n: Node?<Data>): map<int, Data>
  reads n
{
  if n == null then m else m + n.M
}
```

The functions `Less` and `Union` can be given inside class `Node`. However, since they don't depend on a receiver argument, I think they look nicer at the module level, that is, declared in the same module scope as class `Node`.

Now that we have defined the Node invariant, let's define the methods and functions of class `Node`.

17.2.3. Node implementation

In class `Node`, we still need to define the constructor, function `Lookup`, and methods `Add` and `Remove`, which are called to do the heavy lifting for the corresponding routines in class `BinarySearchTree`. The function and methods are defined recursively over the `Node` data structure. I'll show the specifications and will leave most of the implementations as exercises.

Constructor

The `Node` constructor takes a key-value pair and has the standard specification augmented with a postcondition about the initial value of `M`:

```
constructor (key: int, value: Data)
  ensures Valid() && fresh(Repr)
  ensures M == map[key := value]
```

Exercise 17.4.

Implement the `Node` constructor.

Lookup

The specification of `Lookup` has the same structure as the specification of function `Lookup` in class `BinarySearchTree`:

```
function Lookup(key: int): Option<Data>
  requires Valid()
  reads Repr
  ensures key in M.Keys ==> Lookup(key) == Some(M[key])
  ensures key !in M.Keys ==> Lookup(key) == None
```

Exercise 17.5.

Implement function `Lookup` of class `BinarySearchTree`.

Exercise 17.6.

Implement function `Lookup` of class `Node`.

Add

The specification of method `Add`, too, has the same structure as the one for `Add` in class `BinarySearchTree`:

```

method Add(key: int, value: Data)
  requires Valid()
  modifies Repr
  ensures Valid() && fresh(Repr - old(Repr))
  ensures M == old(M)[key := value]

```

Exercise 17.7.

Implement function Add of class BinarySearchTree. Hint: As part of this implementation, you will need to update the Repr field (or else you won't satisfy the specification and the verifier will complain). The simplest way to do that is to end the method body with

```
Repr := Repr + root.Repr;
```

Before I give you the exercise to implement Add in class Node, I want to say something about the termination of its recursive calls. These recursive calls are (spoiler alert!)

```
left.Add(key, value);
```

and

```
right.Add(key, value);
```

where key and value are the parameters of Add itself. By default, Dafny will pick

decreases key

as the termination metric for Add, but this metric does not decrease for the recursive calls. The trees rooted at left and right are contained in the tree rooted at **this**, so we can try for some termination metric that's based on the size or shape of these trees. Conveniently, we already have such a value, namely the representation set of each of these trees, which is stored in the Repr field. So, to prove termination of your implementation of Add, you can use

decreases Repr

but you'll have to declare this **decreases** clause explicitly, to override the default that Dafny gives you.

Now, you're wondering: why didn't we have to supply a similar, explicit **decreases** clause for the implementation of `Lookup` (Exercise 17.6)? It's because, for a function that depends on the state, Dafny starts the default **decreases** clause with the set of objects in the given **reads** clause. Since `Lookup` is declared with **reads** `Repr` to justify its dependence on the state, Dafny's gives it a default **decreases** clause of

```
decreases Repr, key
```

where `key` is its formal parameter. The recursive calls in the implementation of `Lookup` do decrease this measure, so termination was proved automatically (if termination was as “obvious” to you as to the verifier, you may not even have thought about having to prove it).

If you wonder what default **decreases** clause the verifier uses, you can in the IDE place the cursor over the name of a method or function declaration that contains a recursive (or mutually recursive) call, and then the hover text that pops up will tell you.

Exercise 17.8.

Implement function `Add` of class `Node`. Hint: remember the **decreases** clause, as I just explained above.

Remove

Finally, let's define method `Remove` in class `Node`. It's more complicated, because it has to deal with the fact that we're representing an empty map with `null`. We'll have the method return the reference to the updated tree, or return `null` if the updated tree would have represented the empty map.

```
method Remove(key: int) returns (n: Node?<Data>)  
  requires Valid()  
  modifies Repr  
  ensures n != null ==> n.Valid() && n.Repr <= old(Repr)  
  ensures var newMap := old(M) - {key};
```

```
(|newMap.Keys| == 0 ==> n == null) &&
(|newMap.Keys| != 0 ==> n != null && n.M == newMap)
```

Note that this specification deviates slightly from the standard method specification. Instead of promising `this.Valid()` upon termination, this method promises `n.Valid()`—and only if `n` is non-`null`. Also, if `n` is non-`null`, the method could promise the more standard

```
fresh(n.Repr - old(this.Repr))
```

However, to simplify matters slightly, I chose the stronger postcondition above, which does not allow `Remove` to allocate more state. (See Exercise 17.12.)

Exercise 17.9.

Implement function `Remove` of class `BinarySearchTree`.

The implementation of `Remove` in class `Node` takes more effort, so I'll present it in its own subsection.

17.2.4. Node implementation of Remove

As I noted above for the implementation of `Add`, the recursive calls of `Remove` will require an explicitly given **decreases** clause. We would discover this later, but we'll have enough to focus on then, so I suggest we just add it right away:

```
decreases Repr
```

The implementation of `Remove` has four basic cases to consider: the key to be removed may be stored directly in `this`, stored in the left or right sub-tree, or not stored in the tree `this` at all. We'll start with the most difficult:

```
{
  if key == this.key {
```

We're looking at a `Node` whose key-value pair is the one we've been asked to remove. What we do next depends on

the sub-trees of **this**. If there is no left sub-tree, then we can just return **right**—whether or not **right** is **null**:

```
    if left == null {  
        return right;
```

Similarly, if there is no right sub-tree, we just return **left**:

```
    } else if right == null {  
        return left;
```

When there are two sub-trees, we need to do some surgery on these trees. Our plan will be to move the key-value pair with the smallest key in the right subtree to **this**. We'll do this in three steps—find the minimum in **right**, set the key-value stored directly in **this** to that minimum, and then remove that minimum key from the right subtree:

```
    } else {  
        var (k, val) := right.Min();  
        this.key, this.value := k, val;  
        right := right.Remove(k);  
    }
```

Here is the definition of **Min**, whose postcondition is similar to what we've seen earlier in the book for computing minima:

```
function Min(): (int, Data)  
    requires Valid()  
    reads Repr  
    ensures var (k, val) := Min();  
        k in M.Keys && M[k] == val &&  
        forall k' :: k' in M.Keys ==> k <= k'  
{  
    if left == null then  
        (key, value)  
    else  
        left.Min()  
}
```

Function **Min** returns a key-value pair as a 2-tuple, which is a built-in datatype. Note how that 2-tuple is conveniently deconstructed both in the method body of **Remove** and in the specification of **Min**.

Exercise 17.10.

Why is the precondition of `Min` needed?

Continuing with the implementation of `Remove`, we have the cases where the given key can only be stored in the left (respectively, right) sub-tree:

```
    } else if key < this.key && left != null {  
        left := left.Remove(key);  
    } else if this.key < key && right != null {  
        right := right.Remove(key);  
    }
```

In the fourth case, the given key would have been stored in the left (respectively, right) sub-tree, but that sub-tree is empty, so the given key does not exist in the tree represented by `this`, and therefore `Remove` is a no-op.

To wrap up the method, we update `M` and return the receiver `Node`:

```
    M := M - {key};  
    return this;  
}
```

The code we wrote seems plausible, but the verifier complains. The complaint is for the postcondition where `|newMap.Keys| == 0`. If we think about what we're trying to achieve, `newMap` could be empty only if it started off as the singleton map with key `key`. This happens only if both `left` and `right` are `null`. This ought to follow from what the validity condition tells us about `M`:

```
M == Union(Union(map[key := value], left), right)
```

Function `Union` involves the map merge operation (+), for which we later get the set of `.Keys`, so perhaps the verifier does not know enough about the cardinality of these things. Let's help the verifier along to see if that will help it construct the proof. (After all, we do believe the implementation of `Remove` is correct, right? Or do we?)

Postcondition of `Union`

Let's add to `Union` the following postcondition:

```
ensures var u := Union(m, n);  
  |m.Keys| <= |u.Keys|
```

Yay! Using the postcondition, the verifier is able to prove the correctness of `Remove`. But it's not able to prove the postcondition itself, so we'll have to help it along further.

Looking at the body of `Union`, we see that the map returned has all the key-value pairs of the given map `m`. Let's see if the verifier can confirm that. To ask the verifier, we want to add an **assert**, but there's no good place to put it. Let's give a name to the map comprehension in the **else** branch and write an assert about that name before we return it. The new body of `Union` is

```
if n == null then  
  m  
else  
  var u := m + n.M;  
  assert m.Keys <= u.Keys;  
  u
```

The verifier is happy with the asserted condition, but it's still complaining about the similar-looking condition in the postcondition. Apparently, the verifier doesn't know the connection between subset and cardinality. Bother! Oh, well, let's state the property we need as a lemma:

```
lemma SubsetCardinality(a: set, b: set)  
  requires a <= b  
  ensures |a| <= |b|
```

and let's call this lemma from the once-again-updated body of `Union`:

```
if n == null then  
  m  
else  
  var u := m + n.M;  
  SubsetCardinality(m.Keys, u.Keys);  
  u
```

Yay! This makes the postcondition of `Union` verify.

Set-cardinality lemma

To prove lemma `SubsetCardinality`, let's try an inductive proof that calls the lemma recursively on a slightly smaller set `b`:

```
{
  if a != b {
    var x :| x in b - a;
    SubsetCardinality(a, b - {x});
  }
}
```

Yay! The verifier knows enough about these set operations to prove the lemma, which we needed to prove the postcondition of `Union`, which we needed to prove the correctness of `Remove` in class `Node`, which we called from `Remove` in class `BinarySearchTree`, which is now fully proved correct!

Exercise 17.11.

Write an alternative proof for lemma `SubsetCardinality` that removes an element from both `a` and `b` in the recursive call.

Exercise 17.12.

Change the first postcondition of `Node.Remove` to

ensures `n != null ==> n.Valid() && fresh(n.Repr - old(Repr))`

and adjust the method implementation to make it verify.

17.3. Iterator for the Map

Several examples in this book have been implementations of various kinds of *containers*, for example, sets, arrays, and maps. A common operation on a container is to iterate through its elements. In this section, we specify and implement a class `Iterator` whose `GetNext()` method

incrementally returns the elements of a `BinarySearchTree` from Section 17.2.

A central point of this example is that the iterator directly accesses the underlying `BinarySearchTree`. This is typical for iterators of a given data structure. The dependence on the container data structure gives rise to an efficient implementation of the incremental `GetNext()` method, but causes concern if the container were to be modified during the course of the iteration. In the specification I will show, any modification of the container will in effect render any outstanding iterator object useless. This is easily accomplished by defining the iterator validity to depend on the representation set of the container (or, more precisely, by not excluding the possibility of the iterator validity depending on the representation set of the container).

17.3.0. Specification

Here is a client's view of the `Iterator` class:

```
class Iterator<Data> {
  const bst: BinarySearchTree<Data>
  ghost var RemainingKeys: set<int>
  ghost const Repr: set<object>
  ghost predicate Valid()
    reads this, Repr
    ensures Valid() ==>
      this in Repr && bst.Valid() && RemainingKeys <=
bst.M.Keys
  constructor (bst: BinarySearchTree<Data>)
    requires bst.Valid()
    ensures Valid() && fresh(Repr - bst.Repr)
    ensures this.bst == bst && RemainingKeys == bst.M.Keys
  method GetNext() returns (r: Option<(int, Data)>)
    requires Valid()
    modifies Repr - bst.Repr
    ensures Valid()
    ensures match r
      case None =>
        old(RemainingKeys) == RemainingKeys == {}
```

```

    case Some((k, val)) =>
      k in old(RemainingKeys) && bst.M[k] == val &&
      RemainingKeys == old(RemainingKeys) - {k}
}

```

As you can see from the constant field `bst`, an iterator is associated with a fixed `BinarySearchTree`. The state of the iterator includes `RemainingKeys`, which is the subset of the container's keys that the iterator has not yet returned. The iterator constructor initializes `RemainingKeys` to the keys of the container, and each call to `GetNext()` removes and returns one of those keys. The postcondition of `Valid()` advertises to all clients that iterator validity implies container validity as well as the subset relation between `RemainingKeys` and `bst.M.Keys`.

To signal that all keys have been returned, `GetNext()` returns `None`, where the type `Option` is the datatype defined at the beginning of Section 17.2. The postcondition says that `None` is returned only when set `RemainingKeys` is empty. Otherwise, `GetNext()` returns a pair `(k, val)`, where `bst.M[k] == val`. Note that `GetNext()` does not guarantee anything about the order in which the key-value pairs are returned (see Exercise 17.14).

The representation set of the iterator, `Repr`, is specified to be fresh upon construction of the iterator, except possibly for overlapping with the representation set of the container. This permits iterator validity to depend on the state of the container's data structure. Because of this permitted dependency, it is not possible to guarantee the validity of the iterator after a change of `bst.Repr`. 🐼

The iterator's representation set never changes (that is, the set doesn't change, but the state of the objects in the representation set may). Therefore, I declared `Repr` as a **const** (see the notes about immutable fields in Section 16.2.4). The `GetNext()` method is allowed to change the state of the part of the iterator's representation set that lies outside the container's representation set. In other words,

the iterator can read the container data structure, but not modify it.

17.3.1. Test harness

To check that the specification of the iterator is usable and operates as I have claimed, consider the following method:

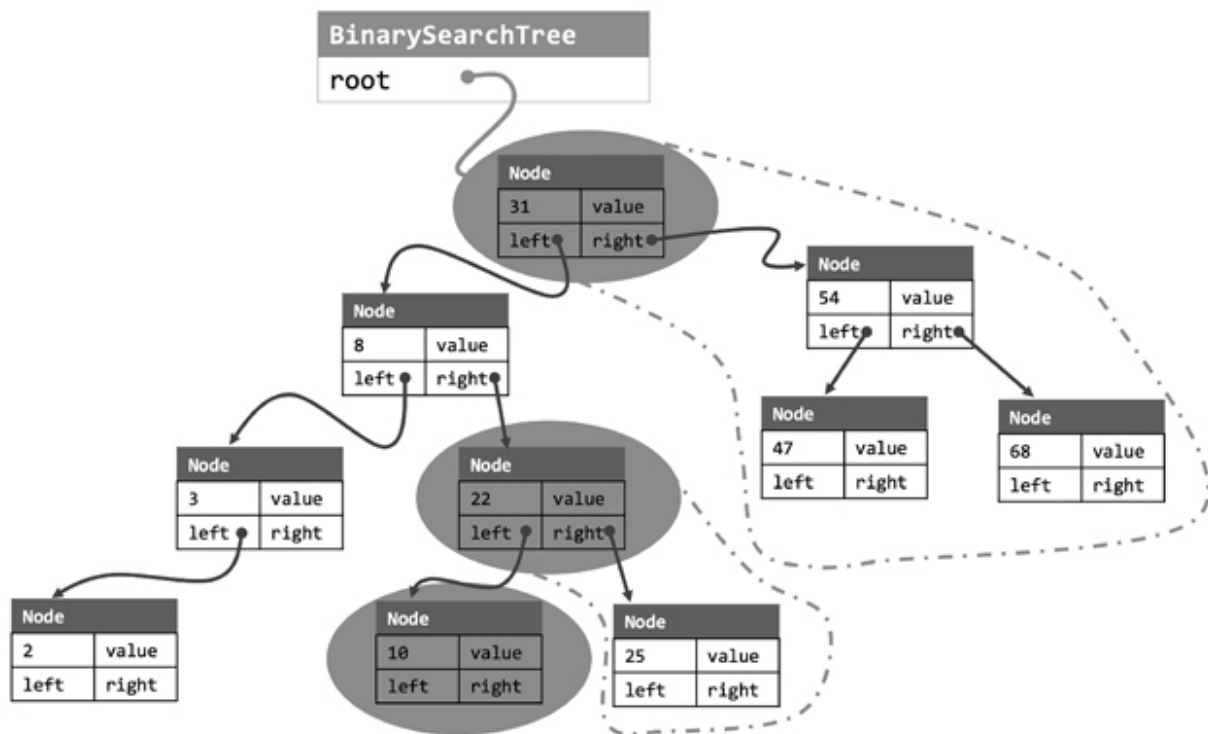
```
method      TestHarness<Data>(a:      BinarySearchTree,      b:
BinarySearchTree,
                                d: Data)
  requires a.Valid() && b.Valid() && a.Repr != b.Repr
  modifies a.Repr
{
  var iter0 := new Iterator(a);
  var iter1 := new Iterator(a);
  var iter2 := new Iterator(b);
  var o := iter0.GetNext();
  o := iter0.GetNext();
  o := iter1.GetNext();
  o := iter0.GetNext();
  o := iter2.GetNext();
  a.Add(15, d);
  o := iter2.GetNext(); // still fine
  o := iter0.GetNext(); // error: container has been modified
}
```

This test harness starts with two valid, disjoint containers *a* and *b*. It then creates two iterators for *a* and one iterator for *b*. As long as the two containers are not changed, it is possible to call the `GetNext()` method on any of the iterators. After container *a* has been modified (by the call to `Add`), the iterator for *b* can still be used. However, it is not possible to prove the precondition `iter0.Valid()` for the final call to `GetNext()` in the example, because `iter0.Valid()` depends on the state that was modified by the call to `Add`. So, in effect, the call to `Add` on *a* renders iterators *iter0* and *iter1* unusable.

17.3.2. Stack of remaining work

The implementation of the iterator performs a pre-order traversal of the nodes in the container. It does so incrementally, so with each call to `GetNext`, the iterator finds the next unvisited node. To keep track of what is left to be done, the iterator maintains a stack of nodes. For each node on the stack, the iterator has yet to visit the key-value pair in the node as well as the right sub-tree of the node.

The following diagram illustrates a binary search tree and the state of an iterator:



This iterator's stack has three nodes, shown in shaded ovals in the diagram. What remains for this iterator to return is the node (whose key is) 10, then node 22 and its right sub-tree, and then node 31 and its right sub-tree.

We'll use a datatype to represent the stack:

```
datatype List<T> = Nil | Cons(T, List<T>)
```

The fact that such a `List` is immutable means we don't have to worry about specifying framing for it. Moreover, since

datatype values are always finite, it is easy to prove termination as we traverse this list.

We add a field to the `Iterator` class to hold the stack:

```
var stack: List<Node<Data>>
```

Let me mention two alternatives to using datatype `List`. One alternative would be to use instances of some class of linked-list nodes. If the `head` and `tail` fields of such a class are declared to be immutable (using `const`), then we don't have to think of framing. However, we would still need to allocate such nodes using `new`, which is not possible in functions, and we'd have to set up more machinery to prove termination when traversing the list. A second alternative would be to use Dafny's built-in `seq` type. That makes framing and termination easy, but having to work with integer indices into such a sequence is more complicated than we need, since a stack is accessed only at one end.

17.3.3. Iterator invariant

We declare the iterator invariant using predicate `Valid`, as usual:

```
ghost predicate Valid()  
  reads this, Repr  
  ensures Valid() ==>  
    this in Repr && bst.Valid() &&  
    RemainingKeys <= bst.M.Keys  
{  
  this in Repr &&  
  bst in Repr && bst.Repr <= Repr && this !in bst.Repr &&  
  bst.Valid() &&  
  RemainingKeys <= bst.M.Keys &&  
  SValid(stack, RemainingKeys)  
}
```

where the last conjunct says that `stack` correctly represents one key-value pair for each key in `RemainingKeys`.

Predicate `SValid` is declared as follows:

```
ghost predicate SValid(st: List<Node<Data>>, R: set<int>)
```

```
reads bst, bst.Repr
```

The **reads** clause of this predicate expresses that it depends on the objects that represent `bst`. To even mention `this.bst.Repr` in the **reads** clause, the **reads** clause must also include `this.bst`, because the state of the object referenced by `this.bst` has an influence on what set of objects `this.bst.Repr` denotes. Using the same argument, one might say that to even mention `this.bst` in the **reads** clause, the **reads** clause must also include `this`. However, this is not so, because `bst` is an immutable field. Therefore, since there is no state change that could cause `this.bst` to refer to another object, the function is allowed to mention it without mentioning it in the **reads** clause.

The definition of `sValid` has two cases, depending on the variant of `st`. One is `Nil`:

```
{  
  match st  
  case Nil =>  
    R == {}  
}
```

The empty stack `Nil` correctly represents one key-value pair for each key in `R` when `R` is empty. There is more to say about the other variant, `Cons`. First up is the structural invariant, which says that `node` and its representation are part of `bst.Repr` and that `node` is valid:

```
case Cons(node, next) =>  
  node in bst.Repr && node.Repr <= bst.Repr && node.Valid()  
&&
```

The next conjunct says that `node.M` agrees with `bst.M`, that is, that every key-value pair represented by `node` is one represented by `bst`:

```
(forall k :: k in node.M.Keys ==>  
  k in bst.M.Keys && bst.M[k] == node.M[k]) &&
```

Next, the `sValid` predicate defines a bound variable `m` as the key-value pair stored directly in `node` combined with the key-value pairs represented by the right sub-tree of `node`:

```
var m := Union(map[node.key := node.value], node.right);
```

We now have that the stack `Cons(node, next)` represents the key-value pairs in `m.Keys` and whatever key-value pairs are represented by the stack `next`. Thus, the last two conjuncts of `SValid` say that `R` can be partitioned into `m.Keys` and `R - m.Keys` such that `node` and `next` represent those respective key-value pairs:

```
    m.Keys <= R &&  
    SValid(next, R - m.Keys)  
}
```

17.3.4. Pushing a node onto the stack

We need to implement the constructor and `GetNext` method of `Iterator`. Both of these will push a `Node` onto the stack. Let's start by writing a method `Push`, which is the most complicated part of our work ahead.

Specification

A call `Push(nn)` will add to the stack all key-value pairs represented by the tree rooted at node `nn`. We allow `nn` to be `null`, in which case nothing is pushed onto the stack.

```
method Push(nn: Node?<Data>)
```

For a method that is called by clients, we expect `Valid()` to be a precondition. However, `Push` is a worker method that is called when the invariant may be temporarily broken. Therefore, instead of requiring `Valid()`, we pick the pieces of `Valid()` that `Push` needs, namely: the container is valid, the iterator object is not part of the representation set of the container (since we are going to modify the fields of the iterator object and want to make sure this does not destroy validity of the container), and the stack starts off correctly representing the keys in `RemainingKeys`:

```
requires bst.Valid() && this !in bst.Repr  
requires SValid(stack, RemainingKeys)
```

We also need three properties of `nn`, if it is non-`null`. First, we need that `nn` is part of the representation set of `bst` and that `nn` is valid:

```
requires nn != null ==>  
  nn in bst.Repr && nn.Repr <= bst.Repr && nn.Valid()
```

Second, the map represented by `nn` must agree with the map represented by `bst`:

```
requires nn != null ==>  
  forall k :: k in nn.M.Keys ==>  
    k in bst.M.Keys && bst.M[k] == nn.M[k]
```

Third, since we are going to add `nn.M.Keys` to the stack that already represents `RemainingKeys` and we want the stack to represent a key only once, we need these sets to be disjoint:

```
requires nn != null ==> RemainingKeys !! nn.M.Keys
```

Phew! Now, method `Push` is to modify the iterator object:

```
modifies this
```

maintaining the invariant relation between `stack` and `RemainingKeys`:

```
ensures SValid(stack, RemainingKeys)
```

with the effect of enlarging `RemainingKeys` with `nn.M.Keys`:

```
ensures RemainingKeys  
  == old(RemainingKeys) + if nn == null then {} else  
nn.M.Keys
```

Body

Operationally, `Push` needs to push `nn` and its transitive left children onto the stack. For example, a call to `Push(bst.root)` on an empty stack will push the nodes reachable by `.left` fields from `bst.root`, giving the state suggested by the diagram in Figure 17.1. We could do this recursively, but if we're just willing to write the loop specification, this is nicely, and more efficiently, done by iteration.

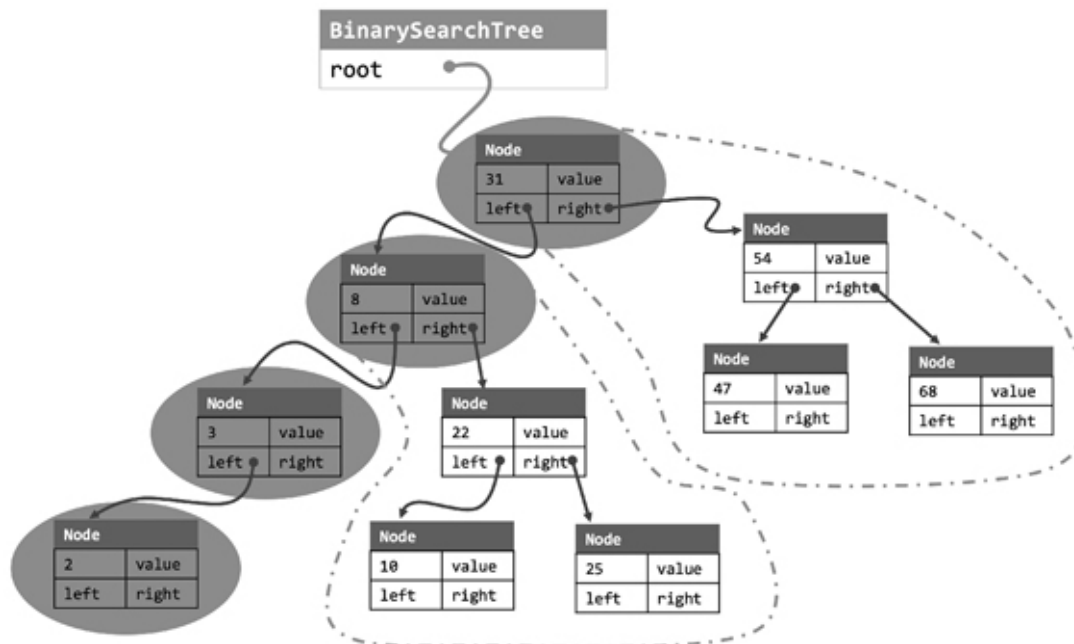


Figure 17.1. The stack, shown as the nodes in shaded ovals, after calling `Push(bst.root)` on the tree `bst`.

So, instead of calling `Push` recursively with a new value for `nn`, we'll use a loop where each iteration takes on a new value for `nn`. In other words, we want to use `nn` as the loop index. However, `nn` is a formal in-parameter, so we are not allowed to assign it in the method body. Instead, we introduce a local variable for this purpose:

```
{
  var node := nn;
```

The loop is going to make progress toward the last postcondition by making the set given by the `if-then-else` expression smaller with each iteration. It will be convenient to give that expression a name:

```
ghost var Final := if node == null then {} else
node.M.Keys;
```

We'll write what remains to be done in a ghost variable:

```
ghost var S := Final;
```

Loop specification

Using `node`, `Final`, and `S`, we write the loop specification from the pieces of the method specification. We continue iterating until we have processed all transitive left children of `nn`:

```
while node != null
```

The method is to maintain the validity of the stack, so we make each iteration do the same (that is, we use the *always* Loop Design Technique 13.1):

```
invariant SValid(stack, RemainingKeys)
```

We'll need the same three properties of `node` as the method does of `nn`. Here are two of them:

```
invariant node != null ==>  
    node in bst.Repr && node.Repr <= bst.Repr &&  
node.Valid()
```

```
invariant node != null ==>  
    forall k :: k in node.M.Keys ==>  
        k in bst.M.Keys && bst.M[k] == node.M[k]
```

For the third property, we'll use our local variable `S`, whose value is also given by an invariant:

```
invariant S == if node == null then {} else node.M.Keys  
invariant RemainingKeys !! S
```

Finally, we turn the last postcondition into a what's-yet-to-be-done invariant (Loop Design Technique 12.1), where `S` is decreased with each loop iteration:

```
invariant RemainingKeys + S == old(RemainingKeys) + Final  
decreases S
```

Loop body

All we have left to do is write the loop body. Let's define a ghost variable `m` to stand for the key-value pairs the iteration is pushing onto the stack:

```
{  
    ghost var m := Union(map[node.key := node.value],  
node.right);
```


Now, we simply push node onto stack while moving node left:

```
stack, node := Cons(node, stack), node.left;
```

Finally, we move the keys in `m.Keys` from `S` to `RemainingKeys`, which completes not just the loop, but also the entire `Push` method:

```
    RemainingKeys, S := RemainingKeys + m.Keys, S - m.Keys;
  }
}
```

Exercise 17.13.

Write a recursive implementation of `Push`.

17.3.5. Constructor

Writing `Push` was the hard part. To implement the constructor, we initialize the fields to represent an empty stack, and then call `Push` to add all of the container's key-value pairs to the stack:

```
constructor (bst: BinarySearchTree<Data>)
  requires bst.Valid()
  ensures Valid() && fresh(Repr - bst.Repr)
  ensures this.bst == bst && RemainingKeys == bst.M.Keys
{
  this.bst := bst;
  stack, RemainingKeys := Nil, {};
  Repr := {this} + bst.Repr;
  new;
  Push(bst.root);
}
```

Note that the call to `Push` is done in the second phase of the constructor, that is, after `new`;. Before that, the iterator's `const` fields (which are used by `Push`) are not available (see Section 16.3.2).

17.3.6. Method `GetNext`

Let me remind you of the type signature of method `GetNext`; refer to Section 17.3.0 for its specification.

method GetNext() **returns** (r: Option<(int, Data)>)

The body of the method is divided into two cases, depending on whether or not all key-value pairs have been returned yet. The first case is easy:

```
{  
  match stack  
  case Nil =>  
    return None;
```

In the second case, we prepare the return value to be the key-value pair stored in the node at the top of the stack:

```
  case Cons(node, next) =>  
    r := Some((node.key, node.value));
```

(Note the double parentheses. The outer parentheses belong to the constructor `Some`. The inner parentheses belong to the 2-tuple constructor.)

Next, we pop `node` off the stack (which is done by setting `stack` to its tail, `next`) and make a corresponding adjustment to `RemainingKeys`:

```
    ghost var m := Union(map[node.key := node.value],  
node.right);  
    stack, RemainingKeys := next, RemainingKeys - m.Keys;
```

This removed too much from `RemainingKeys`, because the specification of `GetNext` says we are to remove just the one key-value pair we return. So, we push `node`'s right subtree back onto the stack, which can be done by one call to `Push`:

```
    Push(node.right);  
  }
```

With that, we have finished the implementation of class `Iterator`.

Exercise 17.14.

Strengthen the specification of `GetNext()` so that it guarantees the returned keyvalue pair has the smallest key of any remaining key-value pair. Then, make any

necessary changes in the implementation and proof to establish the stronger specification.

17.4. Summary

In this chapter, I showed four examples of programs that use dynamically allocated data structures, some of which evolve over time. Two of them were variations on arrays; the others were an implementation of a map and an associated iterator. All of these examples use the specification pattern of recording the object invariant in a predicate `Valid()` that is specified to be a pre- and postcondition of every client-facing method. Framing, which talks about the portion of the heap that is used or modified, is specified by defining the representation set of an aggregate object. The flexibility of dynamic frames was put to good use in the `Iterator` class, where the frames of multiple iterators and a collection can overlap.

Notes

The lazily initialized array in Section 17.0 is from a classic algorithms textbook (exercise 2.12 of [1]).

Functional languages have always supported `Option` types, which let you distinguish the presence of a value from the absence of a value. Since the type `Option<T>` is different from `T`, the types of a program make it clear where a value may be absent. Traditional imperative languages have instead used a special reference value `null` to denote the absence of a value, but—horror! 🤪—without any way for program declarations to distinguish between “definitely present” and “possibly absent”. The first imperative language to distinguish between nullable and non-null reference types was `Spec#` [46, 47], an extension of `C#`.

Type safety of non-null reference types (and, for that matter, of any type for which the compiler cannot easily furnish a value) is achieved by restricting the construction of objects and other composite values. For example, Dafny's two-phase constructors enforce that all non-null fields have been initialized before allowing general use of the new object.

The Iterator example in Section 17.3 avoids a problem that Java programmers know as `ConcurrentModificationException`. It occurs when a collection is modified while there's an active iterator for the collection. To understand the problem in more detail, let's take it from the beginning: The public interface of a collection offers operations for retrieving particular elements from the collection. Each such operation traverses the data structure from the top. To retrieve several elements, it may be more efficient to move a cursor through the data structure, rather than traversing the data structure from the top each time. Iterators for a collection are designed to do just that. Now, an operation that changes the collection will cause mutations of the collection's underlying data structure. After such a mutation, there's no telling what a previously recorded cursor denotes. The `next()` operation of an iterator in Java therefore starts by checking the integrity of the cursor (for example, by looking at some run-time counter that is incremented with each collection update) and throws a `ConcurrentModificationException` if the collection has changed.

As we observed in Section 17.3.1, our specifications allow the verifier to detect this situation. In a nutshell, the validity of the iterator depends on the collection, so any change to the collection renders the iterator invalid.

There's more to note about the interaction between a collection and its iterators. The design allows any number of iterators to be created for a collection. Each iterator depends on the collection, but the iterators do not depend

on one another. Thus, the iterators can be used independently of each other. If the program decides to update the collection, the only proof obligation is to show that the collection is still valid. In particular, there is no need to somehow inform the iterators that the collection is about to be updated. Instead, an iterator gets involved only when the iterator is being used. At that time, one needs to prove that the iterator is still valid, which, with our specifications, is possible only if one can prove there has been no change to the collection. You can observe all of these things in the test harness in Section 17.3.1, so go back and take another look. Pay attention to the precondition of each operation and what is required to prove it.

Dynamic frames flexibly allows representation sets to overlap, which is useful in the Iterator example. When using proof techniques with implicit or predefined representation sets (see the Notes section at the end of Chapter 16), there is no need to supply **modifies** or **reads** clauses. However, there's work to do when creating iterators, because an active iterator needs to claim a piece of the “permission” to, or “ownership” of, the collection. Transferring such permissions or ownership can be done using ghost code. There is also work to do when abandoning an iterator, because the collection then needs to regain all its permissions or ownership in order to undertake further mutating operations.

Reference Material

Appendix A

Dafny Syntax Cheat Sheet

This appendix shows snippets of Dafny syntax. These are intended to jog your memory of, or to suggest, how to use various constructs in Dafny, not to give you a tutorial introduction of the constructs. The snippets are therefore given without much explanation. To find uses of the constructs in this book, consult the Index. For full details, see the Dafny reference manual [[36](#)].

A.0. Declarations

A Dafny program is a hierarchy of nested modules. Dependencies among modules are announced by **import** declarations. A program's import relation must not contain cycles. The export set of a module determines which of the module's declarations are visible to importers.

```
module MyModule {  
  export  
    provides A, B, C  
    reveals D, E, F  
  import L = LibraryA // L is a local name for imported module  
  LibraryA  
  import LibraryB // shorthand for: import LibraryB = LibraryB  
  
  // declarations of types and module-level members...  
}
```

The outermost module of a program is implicit. Therefore, small programs can define methods and functions without needing to wrap them inside a **module** declaration.

A.0.0. Types and type declarations

Here are some example type declarations:

```
datatype Color = Brown | Blue | Hazel | Green
datatype Unary = Zero | Suc(Unary)
datatype List<X> = Nil | Cons(head: X, tail: List<X>)
```

```
class C<X> {
  // class member declarations...
}
```

```
type OpaqueType
```

```
type TypeSynonym = int
```

The x in these examples is a type parameter.

Examples of types:

bool	int	nat	real
set <X>	seq <X>	multiset <X>	map <X, Y>
char	string	$X \rightarrow Y$	
$()$	(X, Y)	(X, Y, Z)	
array <X>	array? <X>	array2 <X>	
object	object?	MyClass <X>	MyClass? <X>

The types shown here with parentheses denote 0-, 2-, and 3-tuples.

A.0.1. Member declarations

```
method M(a: A, b: B) returns (c: C, d: D)
  requires Pre
  modifies obj0, obj1, objectSet
  ensures Post // old(E) refers to the value of E on entry to
the method
  decreases E0, E1, E2
```

A **constructor** (in a **class**) or **lemma** has the same syntax as a **method**. For an anonymous **constructor**, omit the name M .

```
function F(a: A, b: B): C
```



```
requires Pre  
reads obj0, obj1, objectSet  
ensures Post // F(a, b) refers to the result of the function  
decreases E0, E1, E2
```

If `c` is `bool`, then the first line of the function declaration can be written as

```
predicate F(a: A, b: B)
```

Declarations of fields and constants:

```
var b: B // mutable field, can be used only in classes  
const n: nat  
const greeting: string := "hello"  
const year := 1402
```

`function`, `predicate`, `var`, and `const` declarations can be preceded by `ghost`.

A.1. Statements

Each primitive statement ends with a `;` (semi-colon). In contrast, a statement with a body (enclosed in curly braces) does not end with a `;`.

Declaration of local variables:

```
var x: X;
```

The “`: x`” can be omitted if the type can be inferred. When declaring more than one variable, the `:` (colon) binds stronger than `,` (comma). That is,

```
var x, y: Y;
```

declares `y` to have type `Y` and leaves the type of `x` to be inferred.

Assignments:

```

x := E;      // := is pronounced "gets" or "becomes" (NOT
              "equals"!)
x, y := E,   // simultaneous assignment
F;
x :| E;      // assign x a value that makes E hold (assign
              such that)

```

A declaration of a variable and an assignment to the same variable can be combined into one statement, like **var** *x* := *E*;

Dynamic allocation of objects and arrays:

```

c := new C(...);
a := new T[n];
a := new T[n](i => ...);

```

Method calls with 0, 1, and 2 out-parameters:

```

MethodWithNoResults(E, F);
x := MethodWithOneResult(E, F);
x, y := MethodWithTwoResults(E, F);

```

Other primitive statements:

```
assert E; return; return E, F, G; new;
```

Some composite statements:

```

if E {
  // statements...
} else {
  // statements...
}

```

```

if {
  case E0 => // statements...
  case E1 => // statements...
}

```

```

match E {
  case Pattern0(x, y) => // statements...
  case Pattern1(z, _) => // statements...
}

```

```

while Guard
  invariant Inv

```

```

    modifies obj0, obj1, objectSet
    decreases E0, E1, E2
{
    // statements...
}

forall x: X | Range {
    // assignment statement
}

calc {
    E0;
    == { assert HintWhyE0EqualsE1; }
    E1;
    == { LemmaThatExplainsWhyE1EqualsE2(); }
    E2;
}

```

In the **if** statement (unlike in the **if-then-else expression**), the **else** branch is optional, and the curly braces are required. When an **if-case** or **match** statement is given last in a statement list, the curly braces that surround the **cases** can be omitted. Without the curly braces, each **case** is stylistically not indented but kept flush with the **if** or **match** keyword. The **forall** statement is an aggregate statement that simultaneously performs the given assignment statement for every value of x that satisfies *Range*. The **calc** statement is used to write a structured proof calculation.

A.2. Expressions

Figure A.0 shows common operators. Operators in the same section have the same binding power, and the sections are ordered from lowest to highest binding power.

<==>				iff (lowest binding power)
==>		<==		implication, reverse implication
&&				and, or
==		!=		equality, disequality
<		<=		inequality comparisons
		=>		
in		!in		collection membership
!!				set disjointness
+		-		plus/union/concatenation/merge, minus
*		/		multiplication/intersection, division, modulus
		%		
_ as int				conversion to integer
!		-		boolean not, unary negation
_.x				member selection
-[_]		-[_ := _]		element selection, update
-[_ .. _]				subrange
-[_ .. _]		-[_ ..]		take, drop
-[_..]				array-elements to sequence

Figure A.0. Operator binding powers.

For sets, <= denotes subset, + denotes union, * denotes intersection, and - denotes set difference. For multisets, those operators denote the analogous multiset operations. For sequences, <= denotes prefix and + denotes concatenation. For maps, + denotes map merge (where the right-hand operand takes priority) and - denotes map domain subtraction. The operator < is the strict version of <=.

In the member-selection expression E.x, E is an expression (typically a reference or datatype value) and x is a member of the type of E.

The expression E[J] selects member J from E, where E is an array, sequence, or map and J either denotes an index into the array or sequence or denotes a key in the map. For a multiset E, E[J] denotes the multiplicity of element J. The elements of a tuple are selected using numerically named members; for example, the 3 members of a triple E are selected by E.0, E.1, and E.2.

If E is a sequence, map, or multiset, the update expression $E[J := V]$ returns a collection like E except that element J , key J , or the multiplicity of J , respectively, has been replaced by V .

For an array or sequence E , the subsequence expression $E[lo..hi]$ is the sequence of $hi - lo$ elements from E starting at lo . If the lower bound is omitted, it defaults to 0, and if the upper bound is omitted, it defaults to the length of the array or sequence. For an array E , the expression $E[...]$, which has the same meaning as $E[0..E.Length]$, obtains the sequence of all elements of E .

If E is a set, multiset, or sequence, then the expression $|E|$ denotes the total number of elements of E (which is known as the *cardinality* of the set or multiset, and the length of the sequence). The expression $E.Keys$ denotes the set of keys in a map E . The number of elements in an array E is written $E.Length$, and the lengths of the dimensions of a 2-dimensional array E are written $E.Length0$ and $E.Length1$.

The following table shows tuples, set displays, multiset displays, sequence displays, and map displays with 0 and 3 elements (or fewer for the set, if some of a , b , and c are equal):

<code>()</code>	<code>(a, b, c)</code>
<code>{}</code>	<code>{a, b, c}</code>
<code>multiset{}</code>	<code>multiset{a, b, c}</code>
<code>[]</code>	<code>[a, b, c]</code>
<code>map[]</code>	<code>map[x := a, y := b, z := c]</code>

Here are some literals and other expressions:

```

44      1.618    'D'      "hello"
this   null    old(E)  fresh(E)
seq(E, i => ...)      // sequence comprehension

if E then E0 else E1

```

```

match E {
  case Pattern0(x, y) => E0
  case Pattern1(z, _) => E1
}

```

```

assert E0; E1           // like E1, but first asserts E0
MyLemma(); E             // like E1, but first calls MyLemma()

```

```

var x := E0; E1         // pronounced "let x be E0 in E1"

```

```

set x: X | Range
forall x: X :: Expr    // Expr typically has the form E0 ==> E1
exists x: X :: Expr    // Expr often uses &&, seldom ==>

```

In the **if-then-else** expression (unlike in the **if statement**), the **else** branch is required, and there are no curly braces around E0 and E1 (except if they happen to be set-display expressions).

Unless you're nesting one **match** expression inside another, you can omit the curly braces. Without the curly braces, each **case** is stylistically not indented but kept flush with the **match** keyword.

Appendix B

Boolean Algebra

This appendix gives a review of Boolean algebra and some proofs.

B.0. Boolean Values and Negation

The boolean type is made up of two elements, **false** and **true**.

The unary operation *negation*, also known as *not*, goes between these two elements. Common programming languages write it as **!**, and mathematical texts often render it as $\neg$. We have

!true = false

Negation is an *involution*, which is to say that it is its own inverse:

Double Negation: **!!X = X**

In the formula above and throughout this appendix, variables like **x** are to be understood as any expression of the appropriate type (here, boolean).

Double Negation tells us that **true** and **!!true** have the same value. The equality above tells us **!true** is **false**, so **!!true** is **!false**. This shows

true = !false

When a boolean formula equals **true**, we say that it *holds*.

B.1. Conjunction

Conjunction, commonly known as *logical and*, is a binary operator notated as `&&` or $\wedge$. The expression `X && Y` is **true** just when both `X` and `Y` are. As such, **true** is a *left unit element* of `&&`, and **false** is a *left zero element* of `&&`:

Unit: **true** && X = X
 Zero: **false** && X = **false**

Conjunction is *idempotent*, which means that repeated operands don't change the boolean value, *commutative*, which means that the order of operands doesn't change the value, and *associative*, which means that parentheses can be moved around:

Idempotent: X && X = X
 Commutative: X && Y = Y && X
 Associative: X && (Y && Z) = (X && Y) && Z

Since the placement of parentheses for an associative operator doesn't matter, we usually leave them out altogether. In particular, we write just `X && Y && Z` for either of the two expressions in previous line.

By commutativity, the left unit of `&&` is also a right unit, and the left zero is also a right zero.

The following properties make a connection between `!` and `&&`:

Distribution: !(X && !(Y && Z)) = !(X && !Y) && !(X && !Z)
 Law of Excluded Middle: X && !X = **false**

The last of these says that `X` and `!X` cannot both be **true**.

Exercise B.0.

(a) Plug in **false** for `x` in the Distribution property and use the previous properties to show that the Distribution property holds. (b) Plug in **true** for both `y` and `z` in the Distribution property and use the previous properties to show that the Distribution property holds.

B.2. Predicates and Well-Definedness

When we reason about programs, we write expressions that *in a given program state* may evaluate to **false** or **true**. Such boolean expressions are called *predicates*. For example, $a < 100$ is a predicate, and so is its negation, $!(a < 100)$ (that is, $100 \leq a$). The formulas I wrote above (and all similar formulas in this appendix) also hold when x , y , and z are predicates. Therefore, you sometimes hear “Boolean algebra” being called “predicate calculus”; for our purposes, I’ll use such terms to mean the same thing (but if you want to be more clear about the difference, see, e.g., [\[42\]](#)).

When we work with predicates, we may need to worry about when expressions are *well-defined*. For example, the predicate $a == c / d$ is defined only if d is non-zero. Most programming languages (including Dafny) offer a frugal notation for making sure expressions are well-defined: the *short-circuit boolean operators* $\&\&$, $||$, and $==>$. For these operators, the well-definedness of the right-hand operand is modulated by the *value* of the left-hand operand.

For example, the expression

$100 \leq d \ \&\& \ a == c / d$

is well-defined, because the right-hand operand $a == c / d$ is well-defined if $100 \leq d$ is **true**. In contrast,

$a == c / d \ \&\& \ 100 \leq d$

is not well-defined. In other words, as far as well-definedness is concerned, we read the formulas from left to right. So, even though the *value* of $x \ \&\& \ y$ is always the same as $y \ \&\& \ x$, requirements of well-definedness may prevent us from writing down one (or both) of these variants.

I make the notion of well-definedness precise in Section 2.12. In the rest of this appendix, I will only be concerned

with the value of boolean expressions, which is not affected by the short-circuit nature of operators.

B.3. Disjunction and Proof Format

Disjunction, commonly known as *logical or*, is a binary operator notated as `||` or $\vee$. The expression `X || Y` is **true** just when at least one of `X` and `Y` is. We can define it in terms of negation and conjunction, using a rule attributed to De Morgan:

De Morgan's Law: `!(X && Y) = !X || !Y`

From this connection, several properties follow:

Unit: `false || X = X`

Zero: `true || X = true`

Idempotent: `X || X = X`

Commutative: `X || Y = Y || X`

Associative: `X || (Y || Z) = (X || Y) || Z`

Because of commutativity, the left unit and left zero are also a right unit and right zero, respectively.

Let's write a proof. We will prove an alternative formulation of De Morgan's Law:

De Morgan's Law: `!(X || Y) = !X && !Y`

We write the proof as a sequence of steps, each one of which uses a previous definition or a previously proved property.

```
!(X || Y)
= { Double negation (applied in 2 places) }
  !(!!X || !!Y)
= { The first formulation of De Morgan's Law, given above }
  !(!!X && !!Y)
= { Double negation }
  !X && !Y
```

Since each step of this “proof calculation” preserves equality, we conclude that the first line equals the last.

Thus, we have proved the alternative formulation of De Morgan's Law.

Let's write one more proof, this time of the associativity of `||`:

$$\begin{aligned} & X \ || \ (Y \ || \ Z) \\ = & \quad \{ \textit{Double negation (applied in 2 places)} \} \\ & !!X \ || \ !(Y \ || \ Z) \\ = & \quad \{ \textit{De Morgan's Law} \} \\ & !(X \ \&\& \ !(Y \ || \ Z)) \\ = & \quad \{ \textit{De Morgan's Law (alternative formulation)} \} \\ & !(X \ \&\& \ (!Y \ \&\& \ !Z)) \\ = & \quad \{ \textit{Associativity of \&\&} \} \\ & !((X \ \&\& \ !Y) \ \&\& \ !Z) \\ = & \quad \{ \textit{De Morgan's Law (alternative formulation)} \} \\ & !(!X \ || \ Y) \ \&\& \ !Z \\ = & \quad \{ \textit{De Morgan's Law} \} \\ & !(X \ || \ Y) \ || \ !!Z \\ = & \quad \{ \textit{Double negation (applied in 2 places)} \} \\ & (X \ || \ Y) \ || \ Z \end{aligned}$$

Exercise B.1.

Section B.0 argued that `true` is `!false`. Write that proof as a proof calculation.

Exercise B.2.

Prove the Unit, Zero, Idempotent, and Commutative properties of `||` stated above.

Exercise B.3.

Prove the two additional variations of De Morgan's Law:

a) $X \ || \ Y \ = \ !(X \ \&\& \ !Y)$

b) $X \ \&\& \ Y \ = \ !(!X \ || \ !Y)$

And distributes over `or`, and `or` distributes over `and`:

Distribution: $X \ || \ (Y \ \&\& \ Z) \ = \ (X \ || \ Y) \ \&\& \ (X \ || \ Z)$

Distribution: $X \ \&\& \ (Y \ || \ Z) \ = \ (X \ \&\& \ Y) \ || \ (X \ \&\& \ Z)$

Here's a proof of the first of these:

```

X || (Y && Z)
=      { Exercise B.3 }
!(!X && !(Y && Z))
=      { Distribution of ! and && }
!(!X && !Y) && !(!X && !Z)
=      { De Morgan's Law (applied in 2 places) }
(X || Y) && (X || Z)

```

Exercise B.4.

Prove the other distribution property (and distributes over or).

Finally, here an alternative formulation of the Law of Excluded Middle:

Law of Excluded Middle: $X \mid\mid !X = \text{true}$

Exercise B.5.

Prove this formulation of the Law of Excluded Middle.

B.4. Implication

Logical implication is a binary operator, denoted $\implies$ or $\Rightarrow$. The expression $X \implies Y$ is **true** just when X is **false** or Y is **true**. Stated differently, $X \implies Y$ says that Y is **true** whenever X is. You can read it as “ X implies Y ”, “ X only if Y ”, “ X is stronger than Y ”, or “ Y is weaker than X ”. In $X \implies Y$, X is known as the *antecedent* and Y is known as the *consequent*.

Implication: $X \implies Y = !X \mid\mid Y$

The $\implies$ operator has lower binding power (that is, has lower operator precedence) than $\&\&$ and $\mid\mid$. This means that an expression like

```
(W || X) ==> (Y && Z)
```

can be written equivalently without parentheses:

```
W || X ==> Y && Z
```

For this reason, $\implies$ is 3 characters wide in Dafny syntax; visually, this has the effect of spreading out the operands of

$\implies$ further than the operands of $\&\&$ and $\|\|$, as a visual queue to first apply those operators whose operands are closer together. Let me exaggerate this effect further by writing the formula thus:

$W \|\| X \implies Y \&\& Z$

The $\implies$ operator is *right associative*, which is to say that $X \implies Y \implies Z$

means

$X \implies (Y \implies Z)$

Here are some nice properties of implication—so nice that each has a name (and one of them even has a Latin name 🧐):

Modus Ponens: $X \&\& (X \implies Y) = X \&\& Y$

Contrapositive: $X \implies Y = !Y \implies !X$

Shunting: $X \&\& Y \implies Z = X \implies !Y \|\| Z$

Distribution: $X \|\| Y \implies Z = (X \implies Z) \&\& (Y \implies Z)$

Exercise B.6.

Prove (a) Modus Ponens, (b) Contrapositive, (c) Shunting, (d) Distribution, and

e) $X \|\| (!X \implies Y) = X \|\| Y$

f) $X \implies Y \&\& Z = (X \implies Y) \&\& (X \implies Z)$

B.5. Proving Implications

Here are two more nice properties of implication:

Reflexive: $X \implies X$

Transitive: $(X \implies Y) \&\& (Y \implies Z) \implies (X \implies Z)$

Let's prove these properties. Note that each of the formulas has the shape of an implication. This is common among proof obligations that we encounter, so let's consider some techniques for proving them.

A straightforward technique for showing that an implication formula holds is to prove that it equals **true**. For example, we can prove Reflexive by the following proof calculation:

```

X ==> X
=      { Definition of ==> }
!X || X
=      { Law of Excluded Middle }
true

```

Sometimes, the implication formula we want to prove is big, so starting a proof calculation from the entire formula can be cumbersome. A second technique to prove an implication like $A \Rightarrow B$ is to “assume A ” and then prove B . This amounts to proving that B holds, and using properties of A in that proof. (I won't give all the formal details here, but the next example should be illustrative.) A third technique to prove an implication like $A \Rightarrow B$ is to start a proof calculation from A , using steps of equality and steps of implication and eventually ending up with B . Using the second and third of these techniques, here is a proof of Transitive: We assume $(X \Rightarrow Y) \ \&\& \ (Y \Rightarrow Z)$ and then prove $X \Rightarrow Z$ by

```

X
==> { Using the assumption X ==> Y }
Y
==> { Using the assumption Y ==> Z }
Z

```

Exercise B.7.

Prove that each of the following formulas holds.

- a) $(P(x) \ \&\& \ Q(y) \Rightarrow R(x, y)) \ \&\& \ !R(x, y) \ \&\& \ P(x) \Rightarrow !Q(y)$
- b) $(5 < x \Rightarrow y == 10) \ \&\& \ y < 7 \ \&\& \ (y < 1000 \Rightarrow x \leq 5) \Rightarrow x == 5$

For (b), you need to use some simple properties of arithmetic.

B.6. Free Variables and Substitution

The variables occurring in an expression can be either *free* or *bound*. A bound variable is one whose scope is limited to some part of the expression and is not available at the level of the expression. As an example from common mathematics, the expression

$$k + \sum_{i=0}^{<n} F(i, j)$$

mentions four variables, *i*, *j*, *k*, and *n*. Of these variables, *i* is bound, because its scope is limited to the summation, whereas *j*, *k*, and *n* are free. As another example, the variables *x* and *y* in expression

`x < 100 && 20 <= Fib(y)`

are both free.

A bound variable can be renamed to any “fresh” name, that is, a name that is not in use. For example, the expression in the first example above can, without change in meaning, be written as

$$k + \sum_{w=0}^{<n} F(w, j)$$

where *i* has been renamed *w*. The operation of changing the name of a bound variable is cryptically known as *alpha-renaming*. The fact that alpha-renaming never changes the value of an expression reflects the idea that bound variables are “private” to that limited scope where they are defined.

For any expressions *E* and *F* and variable name *x*, the *substitution expression* *E*[*x* := *F*] denotes expression *E* in

which all free occurrences of x have been replaced by F . For example,

```
(x < 100 && 20 <= Fib(y))[y := 2*x + y] =
  x < 100 && 20 <= Fib(2*x + y)
```

There is a restriction, which is that substitution can be applied only if it does not give rise to *variable capture*. Variable capture happens when a free variable of F would become a bound variable of the result of the substitution. For example, if we tried to apply the substitution $[j := 2*i]$ to the first example above, then we would get

$$k + \sum_{i=0}^{<n} F(i, 2 * i)$$

where the i in the substitution has become bound to the summation. Such a substitution is not allowed, because the particular names used for bound variables is not supposed to matter. The point is that the i in $[j := 2*i]$ refers to some variable that should not be mixed up with the bound variable i in the expression being substituted into.

If you want to apply a substitution $[x := F]$ to an expression E without fear of variable capture, you can first alpha-rename all bound variables in E . For example, to apply the substitution $[j := 2*i]$ to the first example expression above, you can first rename the bound i to w and then apply the substitution. The result is then

$$k + \sum_{w=0}^{<n} F(w, 2 * i)$$

When we use substitution, we always (and implicitly) apply alpha-renaming to avoid any problems with variable capture. You may sometimes hear the phrase “capture-avoiding substitution” for this.

Exercise B.8.

Apply the substitution $[x := x - y]$ to the following expressions. Remember to first alpha-rename, if necessary.

- a) $2 * x * t < 100$
- b) $y + x$
- c) $x + y$
- d) $t + 5$
- e)
- f)
- g)

Substitution can also be done simultaneously for several variables. For example, $E[x, y := F, G]$ says to replace all free occurrences of x with F and all free occurrences of y with G at the same time. For example,

$$(F(x) + G(y))[x, y := x + y, x - y] = F(x + y) + G(x - y)$$

Exercise B.9.

Apply the substitution $[x, y, z := x + y, 10, x - y]$ to each of the following expressions.

- a) $y + 4 * x$
- b) $\text{Factorial}(y + t) + z$
- c) $x^2 + y^2 == z^2$

B.7. Universal Quantification

A *universal quantification* is like conjunction on steroids. It is used to express that a predicate $P(x)$ holds for all values of x . It is written

forall $x :: P(x)$

or, in math, $\forall x \cdot P(x)$. The variable x is bound to the expression. Commonly, the type of x is understood from the

context and is then omitted. If needed, the type of x can be written explicitly, as in

forall $x: T :: P(x)$

Here are some important properties that universal quantification enjoys:

Distribution:

$(\text{forall } x :: E \ \&\& \ F) = (\text{forall } x :: E) \ \&\& \ (\text{forall } x :: F)$

One-point rule:

$(\text{forall } x :: x == F ==> E) = E[x := F]$

provided x does not occur free in F

Unused variable:

$(\text{forall } x :: E) = E$

provided x does not occur free in E and the type of x is nonempty

Constant:

$(\text{forall } x :: \text{true}) = \text{true}$

Exercise B.10.

Prove the following property, which is often called Range Split:

$(\text{forall } x :: P \ || \ Q ==> R) =$
 $(\text{forall } x :: P ==> R) \ \&\& \ (\text{forall } x :: Q ==> R)$

Exercise B.11.

Suppose the type of x is nonempty and that x does not occur free in E , F , or P . Prove

a) $(\text{forall } x :: x == E \ || \ x == F ==> R) =$
 $R[x := E] \ \&\& \ R[x := F]$

b) $P \ \&\& \ (\text{forall } x :: Q) = (\text{forall } x :: P \ \&\& \ Q)$

Exercise B.12.

Prove

a) **!forall** $x :: \text{false}$
provided the type of x is nonempty

b) $(\text{forall } x :: \text{false} ==> P)$

c) $(\text{forall } x :: P) ==> P[x := E]$

Exercise B.13.

(a) Prove

$$(\text{forall } x :: P) \mid\mid (\text{forall } x :: Q) \implies (\text{forall } x :: P \mid\mid Q)$$

(b) Give an example of P and Q for which

$$(\text{forall } x :: P \mid\mid Q) \implies (\text{forall } x :: P) \mid\mid (\text{forall } x :: Q)$$

does *not* hold.

Exercise B.14.

Suppose the type of x is nonempty and x does not occur in P . Prove

$$a) \quad P \mid\mid (\text{forall } x :: Q) = (\text{forall } x :: P \mid\mid Q)$$

$$b) \quad P \implies (\text{forall } x :: Q) = (\text{forall } x :: P \implies Q)$$

Exercise B.15.

Given $m < n$, prove

$$a) \quad (\text{forall } i :: m \leq i < n \implies P(i)) = P(m) \ \&\& \ (\text{forall } i :: m + 1 \leq i < n \implies P(i))$$

$$b) \quad (\text{forall } i :: m \leq i < n \implies P(i)) = (\text{forall } i :: m \leq i < n - 1 \implies P(i)) \ \&\& \ P(n - 1)$$

$$c) \quad (\text{forall } i :: 0 \leq i < 3 \implies P(i)) = P(0) \ \&\& \ P(1) \ \&\& \ P(2)$$

B.8. Existential Quantification

The dual of universal quantification is *existential quantification*, which is like disjunction on steroids. It is used to express that a predicate $P(x)$ holds for *some* value of x . It is written

exists $x :: P(x)$

or, in math, $\exists x \cdot P(x)$. We can define it using a generalized form of De Morgan's Law, formulated here in two different ways:

De Morgan's Law for Quantifiers:

$\neg(\text{forall } x :: P) = (\text{exists } x :: \neg P)$

$\neg(\text{exists } x :: P) = (\text{forall } x :: \neg P)$

Exercise B.16.

Prove the second formulation of De Morgan's Law for Quantifiers from the first.

Existential quantification has properties analogous to those of universal quantification:

Exercise B.17.

Prove

- a) $(\text{exists } x :: E \ \&\& \ F) \implies (\text{exists } x :: E) \ \&\& \ (\text{exists } x :: F)$
- b) $(\text{exists } x :: x == F \ \&\& \ E) = E[x := F]$
provided x does not occur free in F
- c) $(\text{exists } x :: E \ \&\& \ F) = E \ \&\& \ \text{exists } x :: F$
provided x does not occur free in E and the type of x is nonempty
- d) $(\text{exists } x :: E) = E$
provided x does not occur free in E and the type of x is nonempty
- e) $(\text{exists } x :: \text{false}) = \text{false}$

Exercise B.18.

Suppose x does not occur free in P . Prove

$P \ || \ (\text{exists } x :: Q) = (\text{exists } x :: P \ || \ Q)$

Exercise B.19.

Prove

- a) $\text{exists } x :: \text{true}$
provided the type of x is nonempty
- b) $\neg(\text{exists } x :: \text{false} \ \&\& \ P)$
- c) $P[x := E] \implies (\text{exists } x :: P)$

Exercise B.20.

Prove

- a) $(\text{exists } x :: P) \parallel (\text{exists } x :: Q) \implies (\text{exists } x :: P \parallel Q)$
- b) $(\text{exists } x :: P \parallel Q) \implies (\text{exists } x :: P) \parallel (\text{exists } x :: Q)$

Exercise B.21.

Suppose the type of x is nonempty and x does not occur in P . Prove

- a) $P \parallel (\text{exists } x :: Q) = (\text{exists } x :: P \parallel Q)$
- b) $P \implies (\text{exists } x :: Q) = (\text{exists } x :: P \implies Q)$

Exercise B.22.

Given $m < n$, prove

- a) $(\text{exists } i :: m \leq i < n \ \&\& \ P(i)) = P(m) \parallel (\text{exists } i :: m + 1 \leq i < n \ \&\& \ P(i))$
- b) $(\text{exists } i :: m \leq i < n \ \&\& \ P(i)) = (\text{exists } i :: m \leq i < n - 1 \ \&\& \ P(i)) \parallel P(n - 1)$
- c) $(\text{exists } i :: 0 \leq i < 3 \ \&\& \ P(i)) = P(0) \parallel P(1) \parallel P(2)$

Exercise B.23.

Suppose x does not occur free in Q . Prove

$$(\text{exists } x :: P) \implies Q = (\text{forall } x :: P \implies Q)$$

Exercise B.24.

For x and y of type integer, prove

- a) $\text{exists } x :: x * x == 9$
- b) $\text{forall } x :: 0 \leq x < 100 \implies \text{exists } y :: 0 \leq y < 10 \ \&\& \ y * y \leq x < (y + 1) * (y + 1)$
- c) $\text{exists } x :: \text{forall } y :: 0 \leq y \implies x^y == x$
- d) $\text{forall } x :: \text{exists } y :: x < y$

Notes

Logicians like to experiment with definitions of different kinds of “logics”. This can lead to insights into how expressive certain sets of ground principles are, and lets different “logics” be compared for expressiveness. Some such logics have influenced the design of programming languages and type systems. For example, *linear logic* has influenced type systems that let you keep track of resources (like memory) [[124](#), [115](#)].

Obvious as it may seem for simple booleans, the Law of Excluded Middle is a popular omission in some proof systems for predicates or types. Systems that omit it are called *constructive* [[34](#), [111](#)], whereas those that include it have the name *intuitionistic*. In Dafny, the Law of Excluded Middle applies.

Appendix C

Answers to Select Exercises

Answer to Exercise 2.34

Yes, provided it also crashes your program. For example, by the equation

$$\mathcal{SP}[\text{assert true, CandyCrushScore} == 50] = \text{CandyCrushScore} == 50$$

we know that *if* we start with a score of 50 and *if* the program terminates without crashing, then the score remains 50. However, the equation says nothing about the post-state if the statement crashes and $\mathcal{SP}$ does not tell us when a statement crashes.

Answer to Exercise 2.35

$$\begin{aligned} \text{NOCRASH}[x := E, P] &= \text{true} \\ \text{NOCRASH}[\text{assert } E, P] &= P \implies E \\ \text{NOCRASH}[S; T, P] &= \text{NOCRASH}[S, P] \ \&\& \ \text{NOCRASH}[T, \mathcal{SP}[S, P]] \\ \text{NOCRASH}[\text{if } B \{ S \} \text{ else } \{ T \}, P] &= \text{NOCRASH}[S, P \ \&\& \ B] \ \&\& \ \text{NOCRASH}[T, P \ \&\& \ !B] \end{aligned}$$

Answer to Exercise 3.10

- a) $x > x - 2$
- b) $x - 2 < x \ \&\& \ 0 \leq x$

Answer to Exercise 3.13

For Outer, use **decreases** a.
For Inner, use **decreases** a, b.

Answer to Exercise 4.1

```
function ReverseColors(t: BYTree): BYTree {  
  match t  
  case BlueLeaf => YellowLeaf  
  case YellowLeaf => BlueLeaf  
  case Node(left, right) => Node(ReverseColors(left),  
ReverseColors(right))  
}
```

Answer to Exercise 4.5

The precondition of the destructor can be met using either an **if-then-else** expression:

```
if t.Node? then t.left == u else false
```

or using the short-circuit boolean operator **&&**:

```
t.Node? && t.left == u
```

Note that `t.Node?` must be checked before using the destructor; it would be an error to write

```
t.left == u && t.Node? // error
```

Answer to Exercise 5.6

```
lemma Ack1(n: nat)  
  ensures Ack(1, n) == n + 2  
{  
  if n == 0 {  
    // trivial  
  } else {  
    calc {  
      Ack(1, n);  
      == // def. Ack  
      Ack(0, Ack(1, n - 1));  
      == // def. Ack(0, _)  
      Ack(1, n - 1) + 1;  
      == { Ack1(n - 1); } // induction hypothesis  
      (n - 1) + 2 + 1;  
      == // arithmetic
```



```

    n + 2;
  }
}
}

```

Answer to Exercise 5.13

The following proof spells out each case in detail:

```

lemma {:induction false} OceanizeIdempotent(t: BYTree)
  ensures Oceanize(Oceanize(t)) == Oceanize(t)
{
  match t
  case BlueLeaf =>
    calc {
      Oceanize(Oceanize(BlueLeaf));
      == // def. Oceanize
      Oceanize(BlueLeaf);
    }
  case YellowLeaf =>
    calc {
      Oceanize(Oceanize(YellowLeaf));
      == // def. Oceanize
      Oceanize(BlueLeaf);
      == // def. Oceanize
      BlueLeaf;
      == // def. Oceanize
      Oceanize(YellowLeaf);
    }
  case Node(left, right) =>
    calc {
      Oceanize(Oceanize(Node(left, right)));
      == // def. Oceanize
      Oceanize(Node(Oceanize(left), Oceanize(right)));
      == // def. Oceanize
      Node(Oceanize(Oceanize(left)), Oceanize(Oceanize(right)));
      == { OceanizeIdempotent(left); }
      Node(Oceanize(left), Oceanize(Oceanize(right)));
      == { OceanizeIdempotent(right); }
      Node(Oceanize(left), Oceanize(right));
      == // def. Oceanize
      Oceanize(Node(left, right));
    }
}

```

Answer to Exercise 5.14

```
lemma {:induction false} OceanizeUpsBlueCount(t: BYTree)
  ensures BlueCount(t) <= BlueCount(Oceanize(t))
{
  match t
  case BlueLeaf =>
  case YellowLeaf =>
  case Node(left, right) =>
    calc {
      BlueCount(Node(left, right));
    == // def. BlueCount
      BlueCount(left) + BlueCount(right);
    <= { OceanizeUpsBlueCount(left); }
      BlueCount(Oceanize(left)) + BlueCount(right);
    <= { OceanizeUpsBlueCount(right); }
      BlueCount(Oceanize(left)) + BlueCount(Oceanize(right));
    == // def. BlueCount
      BlueCount(Node(Oceanize(left), Oceanize(right)));
    == // def. Oceanize
      BlueCount(Oceanize(Node(left, right)));
    }
}
```

Answer to Exercise 5.15

Here is the `calc` statement for the `Cons` case:

```
calc {
  EvalList(SubstituteList(args, n, c), op, env);
== // args == Cons(e, tail)
  EvalList(SubstituteList(Cons(e, tail), n, c), op, env);
== // def. SubstituteList
  EvalList(Cons(Substitute(e, n, c),
    SubstituteList(tail, n, c)), op, env);
== // def. EvalList
  var v0, v1 :=
    Eval(Substitute(e, n, c), env),
    EvalList(SubstituteList(tail, n, c), op, env);
  match op
  case Add => v0 + v1
  case Mul => v0 * v1;
== { EvalSubstitute(e, n, c, env); }
  var v0, v1 :=
    Eval(e, env[n := c]),
```

```

    EvalList(SubstituteList(tail, n, c), op, env);
  match op
  case Add => v0 + v1
  case Mul => v0 * v1;
== { EvalSubstituteList(tail, op, n, c, env); }
  var v0, v1 :=
    Eval(e, env[n := c]),
    EvalList(tail, op, env[n := c]);
  match op
  case Add => v0 + v1
  case Mul => v0 * v1;
== // def. EvalList
  EvalList(Cons(e, tail), op, env[n := c]);
== // args == Cons(e, tail)
  EvalList(args, op, env[n := c]);
}

```

Answer to Exercise 5.16

```

lemma EvalEnv(e: Expr, n: string, env: map<string, nat>)
  requires n in env.Keys
  ensures Eval(e, env) == Eval(Substitute(e, n, env[n]), env)
{
  EvalSubstitute(e, n, env[n], env);
  assert env == env[n := env[n]]; // needed for extensionality
}

```

Answer to Exercise 6.0

```

function Length'<T>(xs: List<T>): nat {
  if xs == Nil then 0 else 1 + Length'(xs.tail)
}

```

```

lemma LengthLength'<T>(xs: List<T>)
  ensures Length(xs) == Length'(xs)
{
}

```

This lemma is proved automatically by Dafny—you only need to write the empty lemma body: {}.

Answer to Exercise 6.3

```

lemma {:induction false} AppendNil<T>(xs: List<T>)
  ensures Append(xs, Nil) == xs

```

```

{
  match xs
  case Nil =>
  case Cons(x, tail) =>
    calc {
      Append(xs, Nil);
      == // def. Append
      Cons(x, Append(tail, Nil));
      == { AppendNil(tail); }
      Cons(x, tail);
      ==
      xs;
    }
}

```

Answer to Exercise 6.6

```

lemma UnitsAreTheSame()
  ensures L == R
{
  calc {
    L;
    == { RightUnit(L); }
    F(L, R);
    == { LeftUnit(R); }
    R;
  }
}

```

Answer to Exercise 6.8

```

function LiberalTake<T>(xs: List<T>, n: nat): List<T>
{
  if n == 0 || xs == Nil then
    Nil
  else
    Cons(xs.head, LiberalTake(xs.tail, n - 1))
}

```

```

function LiberalDrop<T>(xs: List<T>, n: nat): List<T>
{
  if n == 0 || xs == Nil then
    xs
  else
    LiberalDrop(xs.tail, n - 1)
}

```

```
}
```

```
lemma TakesDrops<T>(xs: List<T>, n: nat)  
  requires n <= Length(xs)  
  ensures Take(xs, n) == LiberalTake(xs, n)  
  ensures Drop(xs, n) == LiberalDrop(xs, n)  
{  
}
```

Answer to Exercise 6.14

In the postcondition of `AtFind`, the second argument to `At` is `Find(xs, y)`. The intrinsic specification of `Find` tells us that `Find(xs, y)` does not exceed `Length(xs)`. Moreover, since `||` is a short-circuit operator, `At` is called only if the first disjunct (`Find(xs, y) == Length(xs)`) does not hold. These facts imply the precondition of the call to `At`.

For `BeforeFind`, the second argument to `At` is `i`. Since `==>` is a short-circuit operator, `At` is called only if the antecedent `i < Find(xs, y)` holds. The intrinsic specification of `Find` thus lets us establish the precondition of the call to `At`.

Answer to Exercise 7.2

```
predicate Less(x: Unary, y: Unary) {  
  y != Zero && (x == Zero || Less(x.pred, y.pred))  
}
```

Answer to Exercise 7.3

```
lemma LessTrichotomous(x: Unary, y: Unary)  
  // 1 or 3 of them are true:  
  ensures Less(x, y) <==> x == y <==> Less(y, x)  
  // not all 3 are true:  
  ensures !(Less(x, y) && x == y && Less(y, x))  
{  
}
```

Note that this solution uses the fact that `<==>` is associative, not chaining. That is,

```
Less(x, y) <==> x == y <==> Less(y, x)
```

can for example be parenthesized as

$\text{Less}(x, y) \iff (x == y \iff \text{Less}(y, x))$

Answer to Exercise 7.5

```
lemma {:induction false} AddCorrect(x: Unary, y: Unary)
  ensures UnaryToNat(Add(x, y)) == UnaryToNat(x) +
UnaryToNat(y)
{
  match y
  case Zero =>
  case Suc(y') =>
    calc {
      UnaryToNat(Add(x, y));
    == // y == Suc(y')
      UnaryToNat(Add(x, Suc(y')));
    == // def. Add
      UnaryToNat(Suc(Add(x, y')));
    == // def. UnaryToNat
      1 + UnaryToNat(Add(x, y'));
    == { AddCorrect(x, y'); }
      1 + UnaryToNat(x) + UnaryToNat(y');
    == // def. UnaryToNat
      UnaryToNat(x) + UnaryToNat(Suc(y'));
    == // y == Suc(y')
      UnaryToNat(x) + UnaryToNat(y);
    }
}
```

```
lemma {:induction false} SucAdd(x: Unary, y: Unary)
  ensures Suc(Add(x, y)) == Add(Suc(x), y)
{
  match y
  case Zero =>
  case Suc(y') =>
    calc {
      Suc(Add(x, Suc(y')));
    == // def. Add
      Suc(Suc(Add(x, y')));
    == { SucAdd(x, y'); }
      Suc(Add(Suc(x), y'));
    == // def. Add
      Add(Suc(x), Suc(y'));
    }
}
```

```

}

lemma {:induction false} AddZero(x: Unary)
  ensures Add(Zero, x) == x
{
  match x
  case Zero =>
  case Suc(x') =>
    calc {
      Add(Zero, Suc(x'));
      == // def. Add
      Suc(Add(Zero, x'));
      == { AddZero(x'); }
      Suc(x');
    }
}

```

Answer to Exercise 9.0

```

module ImmutableQueue {
  import LL = ListLibrary

  type Queue<A>
  function Empty(): Queue
  function Enqueue<A>(q: Queue, a: A): Queue
  function Dequeue<A>(q: Queue): (A, Queue)
    requires q != Empty<A>()

  ghost function Length(q: Queue): nat
  lemma EmptyCorrect<A>()
    ensures Length(Empty<A>()) == 0
  lemma EnqueueCorrect<A>(q: Queue, x: A)
    ensures Length(Enqueue(q, x)) == Length(q) + 1
  lemma DequeueCorrect(q: Queue)
    requires q != Empty()
    ensures Length(Dequeue(q).1) == Length(q) - 1
}

```

Answer to Exercise 9.5

In `Client()`, we wrote that equality in an **assert**, which is a ghost statement. In ghost contexts, Dafny supports equality for all types.

Answer to Exercise 9.6

```
module QueueExtender {
  import IQ = ImmutableQueue
  function TryDequeue<A>(q: IQ.Queue, default: A): (A,
    IQ.Queue)
  {
    if IQ.IsEmpty(q) then (default, q) else IQ.Dequeue(q)
  }
}
```

Answer to Exercise 10.0

To prove this program correct, we need an auxiliary assertion that reminds the verifier of what it knows about `Elements(pq)` after the two insertions.

```
module PriorityQueueTestHarness {
  import PQ = PriorityQueue
  method Test(x: int, y: int) {
    PQ.EmptyCorrect(); var pq := PQ.Empty();
    PQ.InsertCorrect(pq, x); pq := PQ.Insert(pq, x);
    PQ.InsertCorrect(pq, y); pq := PQ.Insert(pq, y);
    assert PQ.Elements(pq) == multiset{x,y};
    PQ.IsEmptyCorrect(pq); PQ.RemoveMinCorrect(pq);
    var (a, pq') := PQ.RemoveMin(pq);
    PQ.IsEmptyCorrect(pq'); PQ.RemoveMinCorrect(pq');
    var (b, pq'') := PQ.RemoveMin(pq');
    assert {a,b} == {x,y} && a <= b;
  }
}
```

With all these lemmas, we get the work done, but the result ain't pretty. We'll work on prettifying the situation in Section 10.3.

Answer to Exercise 10.2

```
lemma {:induction false} BinaryHeapStoresMinimum(pq: PQueue, y:
int)
  requires IsBinaryHeap(pq) && y in Elements(pq)
  ensures pq.x <= y
{
  // By the definition of Elements, we consider the three
  // cases that "y in Elements(pq)" gives rise to.
```



```

if
case y == pq.x =>
  // trivial
case y in Elements(pq.left) =>
  calc {
    pq.x;
    <= // def. IsBinaryHeap
      pq.left.x;
    <= { BinaryHeapStoresMinimum(pq.left, y); }
    y;
  }
case y in Elements(pq.right) =>
  calc {
    pq.x;
    <= // def. IsBinaryHeap
      pq.right.x;
    <= { BinaryHeapStoresMinimum(pq.right, y); }
    y;
  }
}

```

Answer to Exercise 10.3

See Section 10.3.0.

Answer to Exercise 10.4

```

lemma BalanceConsequence(pq: PQueue)
  requires !IsEmpty(pq) && IsBalanced(pq)
  ensures pq.left == Leaf ==> pq.right == Leaf
{
}

```

Answer to Exercise 11.7

```

UpWhileLess:
  invariant i <= N
  decreases N - i

```

```

UpWhileNotEqual:
  invariant i <= N
  decreases N - i

```

```

DownWhileNotEqual:

```

```

invariant 0 <= i
decreases i

```

DownWhileGreater:

```

invariant 0 <= i
decreases i

```

Answer to Exercise 12.5

```

method FastExp(b: nat, n: nat) returns (p: nat)
  ensures p == Exp(b, n)
{
  p := 1;
  var d, k := b, n;
  while k != 0
    invariant p * Exp(d, k) == Exp(b, n)
    {
      calc {
        Exp(d, k);
        == { assert k == 2 * (k / 2) + k % 2; }
        Exp(d, 2 * (k / 2) + k % 2);
        == { ExpAddExponent(d, k / 2 * 2, k % 2); }
        Exp(d, 2 * (k / 2)) * Exp(d, k % 2);
        == { ExpSquareBase(d, k / 2); }
        Exp(d * d, k / 2) * Exp(d, k % 2);
      }
      if k % 2 == 1 {
        assert Exp(d, k % 2) == d;
        p := p * d;
      }
      d, k := d * d, k / 2;
    }
}

```

Answer to Exercise 12.8

```

lemma {:induction false} AppendSumUp(lo: int, hi: int)
  requires lo < hi
  ensures SumUp(lo, hi - 1) + F(hi - 1) == SumUp(lo, hi)
  decreases hi - lo
{
  if lo == hi - 1 {
  } else {
    AppendSumUp(lo + 1, hi);
  }
}

```

```
}
```

Answer to Exercise 13.3

```
forall i, j :: 0 <= i < j < a.Length ==> a[i] < a[j]
```

Answer to Exercise 13.14

Add a case:

```
case a[m] == b[n] =>  
  return 0;
```

and (optionally) change the `<=` in each of the other guards to `<`.

Answer to Exercise 13.19

Because `lo` and `hi` are known to be at least 0, you can write

```
mid := lo + (hi - lo) / 2;
```

Answer to Exercise 14.0

Add

```
requires a.Length == 1 ==> left == right
```

Answer to Exercise 14.2

The following postcondition makes the method verify:

```
ensures a[i] == old(a[i]) + 1
```

Answer to Exercise 14.3

The following assignment implements the specification of `OldVsParameters`:

```
y := 25 - a[i];
```

Answer to Exercise 14.8

Hint: Method `DoubleArray` is like `CopyArray`, with two differences. One difference is to insert `2 *` in front of the `src`

term in the method specification, loop specification, and loop body. The other difference is that the invariant that talks about elements of `src` being unchanged must only quantify over the indices in the range `n <= i < src.Length`.

Answer to Exercise 14.13

```
method CopyMatrix<T>(src: array2<T>, dst: array2<T>)
  requires src.Length0 == dst.Length0
  requires src.Length1 == dst.Length1
  modifies dst
  ensures forall i, j :: 0 <= i < dst.Length0 && 0 <= j <
dst.Length1 ==>
    dst[i, j] == old(src[i, j])
{
  forall i, j | 0 <= i < dst.Length0 && 0 <= j < dst.Length1 {
    dst[i, j] := src[i, j];
  }
}
```

```
method TestHarness() {
  var m := new int[2, 1];
  m[0, 0], m[1, 0] := 5, 7;
  CopyMatrix(m, m);
  // the following assertion will not hold if you forget
  // 'old' in the specification of CopyMatrix
  assert m[1, 0] == 7;
  var n := new int[2, 1];
  CopyMatrix(m, n);
  assert m[1, 0] == n[1, 0] == 7;
}
```

Answer to Exercise 15.5

Insert the following code before the assignment to `pivot`:

```
var p0, p1, p2 := a[lo], a[(lo + hi) / 2], a[hi - 1];
if {
  case p0 <= p1 <= p2 || p2 <= p1 <= p0 =>
    a[(lo + hi) / 2], a[lo] := a[lo], a[(lo + hi) / 2];
  case p1 <= p2 <= p0 || p0 <= p2 <= p1 =>
    a[hi - 1], a[lo] := a[lo], a[hi - 1];
  case p2 <= p0 <= p1 || p1 <= p0 <= p2 =>
    // nothing to do
```

```
}
```

Answer to Exercise 15.7

Here are three hints. First, here is a good specification:

```
requires IsSorted(a, 0, a.Length) && IsSorted(b, 0, b.Length)
ensures fresh(c) && c.Length == a.Length + b.Length
ensures IsSorted(c, 0, c.Length)
ensures multiset(c[..]) == multiset(a[..]) + multiset(b[..])
```

You'll have to define the predicate `IsSorted`. The postcondition `fresh(c)` lets the caller know that the returned array is independent of any array that the caller may have already had. The condition `c.Length == a.Length + b.Length` follows from the last postcondition, but the proof requires induction, so callers will probably appreciate the condition being stated directly in the postcondition.

Second, for the implementation, use the *replace constant by variable* Loop Design Technique 12.0 to replace the implicit upper-bound constants `a.Length`, `b.Length`, and `c.Length` in `a[..]`, `b[..]`, and `c[..]` by variables, say, `i`, `j`, and `k`. As the last line of your implementation, you will need to help the verifier with the hint

```
assert a[..i] == a[..] && b[..j] == b[..] && c[..k] == c[..];
```

Third, in your loop invariant, you will need a condition similar to the `SplitPoint` predicate we used for Selection Sort and Quicksort. I suggest something like

```
predicate Below(a: seq<int>, b: seq<int>) {
  forall i, j :: 0 <= i < |a| && 0 <= j < |b| ==> a[i] <= b[j]
}
```

Good luck!

Answer to Exercise 16.8

```
method RemoveGrinder() returns (grinder: Grinder)
  requires Valid()
  modifies Repr
  ensures Valid() && fresh(Repr - old(Repr))
  ensures grinder.Valid() && grinder in old(Repr) - Repr
```

```

{
  grinder := g;
  g := new Grinder();
  Repr := Repr + {g} - {grinder};
}

```

Answer to Exercise 17.0

Here is a good export set for the module:

```

export
  reveals LazyArray
  provides LazyArray.Elements, LazyArray.Repr
  provides LazyArray.Valid
  provides LazyArray.Get, LazyArray.Update

```

Answer to Exercise 17.3

The body of the constructor is

```

M, Repr, root := map[], {this}, null;

```

Answer to Exercise 17.7

The body of `BinarySearchTree.Add` is

```

if root == null {
  root := new Node(key, value);
} else {
  root.Add(key, value);
}
M := root.M;
Repr := Repr + root.Repr;

```

Answer to Exercise 17.10

Without knowing the node is valid, nothing would be known about the relation between the fields of `this` (which is needed, for example, to prove the postcondition `k in M.Keys`), or the relation between the fields of `this` and the fields of `left` (which is needed, for example, to prove termination of the recursive call).

References

- [0] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
- [1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [2] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification — The KeY Book — From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.
- [3] Krzysztof R. Apt. Ten years of Hoare's logic: A survey — Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.
- [4] Linard Arquint, João C. Pereira, Peter Müller, and Felix Wolf. Gobra.
<https://www.pm.inf.ethz.ch/research/gobra.html>.
- [5] Vytautas Astrauskas, Aurel Bílý, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. Prusti.
<https://www.pm.inf.ethz.ch/research/prusti.html>.
- [6] Ralph-Johan Back. Proving total correctness of nondeterministic programs in infinitary logic. *Acta Informatica*, 15:233–249, 1981.

- [7] Ralph-Johan Back. A method for refining atomicity in parallel algorithms. In Eddy Odijk, Martin Rem, and Jean-Claude Syre, editors, *PARLE '89: Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, volume 366 of *Lecture Notes in Computer Science*, pages 199–216. Springer, 1989.
- [8] Ralph-Johan Back and Joakim von Wright. Duality in specification languages: A lattice-theoretical approach. *Acta Informatica*, 27(7):583–625, 1990.
- [9] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.
- [10] Ralph-Johan R. Back. *On the Correctness of Refinement Steps in Program Development*. PhD thesis, University of Helsinki, 1978. Report A-1978-4.
- [11] Roland Backhouse, editor. *Special issue on The Calculational Method*, volume 53(3). Information Processing Letters, February 1995.
- [12] Roland C. Backhouse. *Program Construction and Verification*. Series in Computer Science. Prentice-Hall International, 1986.
- [13] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: The Spec# experience. *Communications of the ACM*, 54(6):81–91, June 2011.
- [14] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The dogged pursuit of bug-free C programs: the Frama-C

software analysis platform. *Communications of the ACM*, 64(8):56–68, 2021.

- [15] Gertrud Bauer and Markus Wenzel. Computational reasoning revisited (an Isabelle/Isar experience). In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, page 7590. Springer, 2001.
- [16] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development — Coq’Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.
- [17] Richard Bird. *Pearls of Functional Algorithm Design*. Cambridge University Press, 2010.
- [18] Richard S. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, 1989.
- [19] Garrett Birkhoff. *Lattice Theory*. Colloquium Publications, Volume 25. American Mathematical Society, 1967.
- [20] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. Why3 — Where programs meet provers.
<https://why3.lri.fr>.
- [21] François Bourdoncle. Abstract debugging of higher-order imperative languages. In Robert Cartwright, editor, *Proceedings of the ACM SIGPLAN’93 Conference on Programming Language Design and Implementation (PLDI)*, pages 46–55. ACM, 1993.

- [22] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda — A functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer, 2009.
- [23] Robert S. Boyer and J Strother Moore. A *Computational Logic*. ACM Monograph Series. Academic Press, Inc., 1979.
- [24] W. Braun and M. Rem. A logarithmic implementation of flexible arrays. Memorandum MR83/4, University of Technology Eindhoven, 1983.
- [25] Stephen Brookes and Peter W. O’Hearn. Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65, 2016.
- [26] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005.
- [27] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.
- [28] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report 92/480/18, Department of Computer Science, The University of Calgary, June 1992.
- [29] Edward Cohen. *Programming in the 1990s: An Introduction to the Calculation of Programs*. Texts

and Monographs in Computer Science. Springer-Verlag, 1990.

- [30] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010*, volume 6174 of *Lecture Notes in Computer Science*, pages 480–494. Springer, 2010.
- [31] David R. Cok, Gary T. Leavens, and Mattias Ulbrich. *JML Reference Manual (2nd edition)*, 2021.
https://www.openjml.org/documentation/JML_Reference_Manual.pdf.
- [32] David R. Cok and Serdar Tasiran. Practical methods for reasoning about Java 8's functional programming features. In Ruzica Piskac and Philipp Rümmer, editors, *Verified Software. Theories, Tools, and Experiments — 10th International Conference, VSTTE 2018*, volume 11294 of *Lecture Notes in Computer Science*, pages 267–278. Springer, 2018.
- [33] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In Michael I. Schwartzbach and Thomas Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 415–426. ACM, 2006.
- [34] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.
- [35] Dafny — A verification-aware programming language.
<https://dafny.org>.

- [36] *Dafny Documentation*, 2022. <https://dafny.org/dafny>.
- [37] Stijn de Gouw, Frank S. de Boer, and Jurriaan Rot. Proof pearl: The KeY to correct and stable sorting. *Journal of Automated Reasoning*, 53(2):129–139, 2014.
- [38] Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bubel, and Reiner Hähnle. OpenJDK's `java.utils.Collection.sort()` is broken: The good, the bad and the worst case. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification — 27th International Conference, CAV 2015*, volume 9206 of *Lecture Notes in Computer Science*, pages 273–289. Springer, 2015.
- [39] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, 1998.
- [40] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.
- [41] Edsger W. Dijkstra and W. H. J. Feijen. *A Method of Programming*. Addison-Wesley, 1988.
- [42] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.
- [43] Claire Dross, Pavlos Efstathopoulos, David Lesens, David Mentré, and Yannick Moy. Rail, space, security: Three case studies for SPARK 2014. In *7th European Congress on Embedded Real Time Software and Systems (ERTS² 2014)*, 2014.

- [44] Eiffel Software. Eiffel. <https://www.eiffel.com>.
- [45] Marco Eilers and Peter Müller. Nagini. <https://www.pm.inf.ethz.ch/research/nagini.html>.
- [46] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In Ron Crocker and Guy L. Steele Jr., editors, *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003*, pages 302–312. ACM, 2003.
- [47] Manuel Fähndrich and Songtao Xia. Establishing object invariants with delayed types. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007*, pages 337–350. ACM, 2007.
- [48] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods in System Design*, 48(3):152–174, 2016.
- [49] Jean-Christophe Filliâtre and Nicolas Magaud. Certification of sorting algorithms in the Coq system. In *Theorem Proving in Higher Order Logics: Emerging Trends*, 1999.
- [50] Robert W. Floyd. Assigning meanings to programs. *Proceedings Symposium on Applied Mathematics*, 19:19–31, 1967.
- [51] M. Foley and C. A. R. Hoare. Proof of a recursive program: Quicksort. *The Computer Journal*, 14(4):391–395, 1971.

- [52] Four ferries. <https://fourferries.com>, 2022.
- [53] F*. <https://www.fstar-lang.org>.
- [54] James Gosling, Bill Joy, and Guy Steele. *The Java™ Language Specification*. Addison-Wesley, 1996.
- [55] David Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1981.
- [56] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Texts and Monographs in Computer Science. Springer-Verlag, 1994.
- [57] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.
- [58] David Harel. Proving the correctness of regular deterministic programs: A unifying survey using dynamic logic. *Theoretical Computer Science*, 12:61–81, 1980.
- [59] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew J. Parkinson. Behavioral interface specification languages. *ACM Computing Surveys*, 44(3):16:1–16:58, 2012.
- [60] Eric C. R. Hehner. do considered od: A contribution to the programming calculus. *Acta Informatica*, 11:287–304, 1979.
- [61] Eric C. R. Hehner. *A Practical Theory of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

- [62] Eric C. R. Hehner. Specified blocks. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005*, volume 4171 of *Lecture Notes in Computer Science*, pages 384–391. Springer, 2005.
- [63] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969.
- [64] Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, August 2008.
- [65] Bart Jacobs, Frank Piessens, K. Rustan M. Leino, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In Bernhard K. Aichernig and Bernhard Beckert, editors, *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005)*, pages 137–147. IEEE Computer Society, 2005.
- [66] The Java Modeling Language (JML).
<http://www.jmlspecs.org>.
- [67] Cliff B. Jones. *Systematic Software Development using VDM*. Series in Computer Science. Prentice Hall International, second edition, 1990.
- [68] Anne Kaldewaij. *Programming: The Derivation of Algorithms*. Series in Computer Science. Prentice Hall International, 1990.
- [69] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and

Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, August 2006.

- [70] Shmuel Katz and Zohar Manna. A closer look at termination. *Acta Informatica*, 5:333–352, 1975.
- [71] Matt Kaufmann and J Strother Moore. ACL2 version 8.4. <https://www.cs.utexas.edu/users/moore/acl2>.
- [72] Leslie Lamport. How to write a proof. *The American Mathematical Monthly*, 102(7):600–608, 1995.
- [73] Leslie Lamport. The TLA⁺ home page. <https://lamport.azurewebsites.net/tla/tla.html>, March 2022.
- [74] Butler W. Lampson, James J. Horning, Ralph L. London, James G. Mitchell, and Gerald L. Popek. Report on the programming language Euclid. Technical Report CSL-81-12, Xerox PARC, October 1981.
- [75] Lean theorem prover. <https://leanprover.github.io>.
- [76] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, April 2010.
- [77] K. Rustan M. Leino. Automating induction with an SMT solver. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation — 13th International Conference, VMCAI 2012*, volume 7148 of *Lecture*

Notes in Computer Science, pages 315–331. Springer, 2012.

- [78] K. Rustan M. Leino. Accessible software verification with Dafny. *IEEE Software*, 34(6):94–97, 2017.
- [79] K. Rustan M. Leino and Paqui Lucio. An assertional proof of the stability and correctness of natural mergesort. *ACM Transactions on Computational Logic*, 17(1):6:1–6:22, 2015.
- [80] K. Rustan M. Leino and Daniel Matichuk. Modular verification scopes via export sets and translucent exports. In Peter Müller and Ina Schaefer, editors, *Principled Software Development — Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday*, pages 185–202. Springer, 2018.
- [81] K. Rustan M. Leino and Rosemary Monahan. Reasoning about comprehensions with first-order SMT solvers. In Sung Y. Shin and Sascha Ossowski, editors, *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC)*, pages 615–622. ACM, March 2009.
- [82] K. Rustan M. Leino and Michał Moskal. Usable auto-active verification. In Tom Ball, Lenore Zuck, and N. Shankar, editors, *Workshop on Usable Verification*, November 2010.
<https://fm.csl.sri.com/UV10/index.shtml>.
- [83] K. Rustan M. Leino and Peter Müller. Using the Spec# language, methodology, and tools to write bug-free programs. In Peter Müller, editor, *Advanced Lectures on Software Engineering, LASER Summer School 2007/2008*, volume 6029 of *Lecture Notes in Computer Science*, pages 91–139. Springer, 2008.

- [84] K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009*, volume 5502 of *Lecture Notes in Computer Science*, pages 378–393. Springer, 2009.
- [85] K. Rustan M. Leino and Nadia Polikarpova. Verified calculations. In Ernie Cohen and Andrey Rybalchenko, editors, *Verified Software: Theories, Tools, Experiments — 5th International Conference, VSTTE 2013, Revised Selected Papers*, volume 8164 of *Lecture Notes in Computer Science*, pages 170–190. Springer, 2014.
- [86] LiquidHaskell. <https://ucsd-progsys.github.io/liquidhaskell-blog>.
- [87] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Electrical Engineering and Computer Science Series. MIT Press, 1986.
- [88] Ralph L. London. Certification of algorithm 245 [M1]: Treesort 3: Proof of algorithms — A new kind of certification. *Communications of the ACM*, 13(6):371–373, 1970.
- [89] Bertrand Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice Hall International, 1988.
- [90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.
- [91] J Strother Moore and Claus-Peter Wirth. Automation of mathematical induction as part of the history of

logic. Technical Report SR201302, SEKI, 2014.

- [92] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.
- [93] Carroll Morgan. *Programming from Specifications*. Series in Computer Science. Prentice Hall International, second edition, 1994.
- [94] Carroll Morgan. (in-)formal methods: The lost art — A users’ manual. In Zhiming Liu and Zili Zhang, editors, *Engineering Trustworthy Software Systems — First International School, SETSS 2014. Tutorial Lectures*, volume 9506 of *Lecture Notes in Computer Science*, pages 1–79. Springer, 2016.
- [95] Carroll C. Morgan. Data refinement using miracles. *Information Processing Letters*, 26(5):243–246, January 1988.
- [96] Joseph M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.
- [97] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62(3):253–286, 2006.
- [98] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation — 17th International Conference, VMCAI 2016*, volume 9583

of *Lecture Notes in Computer Science*, pages 41–62. Springer, 2016.

- [99] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, October 1989.
- [100] Greg Nelson, editor. *Systems Programming with Modula-3*. Series in Innovative Technology. Prentice-Hall, Englewood Cliffs, NJ, 1991.
- [101] Greg Nelson. LIM and Nanoweb. Technical Report HPL-2005-41, HP Labs, February 2005.
<https://www.hpl.hp.com/techreports/2005/HPL-2005-41.pdf>.
- [102] Tobias Nipkow. Structured proofs in Isar/HOL. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*, pages 259–278. Springer, 2002.
- [103] Peter O’Hearn. Separation logic. *Communications of the ACM*, 62(2):86–95, 2019.
- [104] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
- [105] OpenJML — Does your program do what it is supposed to do? <https://www.openjml.org>.
- [106] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.
- [107] Sam Owre, John M. Rushby, N. Shankar, and David W. J. Stringer-Calvert. PVS system.
<https://pvs.csl.sri.com>, January 2021.

- [108] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
- [109] David J. Pearce, Lindsay Groves, and Mark Utting. Whiley — A programming language with extended static checking. <http://whiley.org>.
- [110] Simon Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98 report. <https://www.haskell.org/onlinereport>, February 1999.
- [111] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In Michael G. Main, Austin Melton, Michael W. Mislove, and David A. Schmidt, editors, *Mathematical Foundations of Programming Semantics, 5th International Conference*, volume 442 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 1989.
- [112] Nadia Polikarpova, Julian Tschannen, Carlo A. Furia, and Bertrand Meyer. Flexible invariants through semantic collaboration. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods — 19th International Symposium*, volume 8442 of *Lecture Notes in Computer Science*, pages 514–530. Springer, 2014.
- [113] John C. Reynolds. *The Craft of Programming*. Series in Computer Science. Prentice-Hall International, 1981.

- [114] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74. IEEE Computer Society, 2002.
- [115] Rust — A language empowering everyone to build reliable and efficient software. <https://www.rust-lang.org>.
- [116] Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. Automatic verifier for Java-like programs based on dynamic frames. In José Luiz Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008*, volume 4961 of *Lecture Notes in Computer Science*, pages 261–275. Springer, March–April 2008.
- [117] SPARK 2014 — Expanding the boundaries of safe and secure programming. <https://www.adacore.com/about-spark>, 2002.
- [118] Stainless — Formal verification for Scala. <https://stainless.epfl.ch>.
- [119] Christian Sternagel. Proof pearl — A mechanized proof of GHC's mergesort. *Journal of Automated Reasoning*, 51(4):357–370, 2013.
- [120] Aaron Stump. *Verified Functional Programming in Agda*. ACM and Morgan & Claypool, New York, NY, USA, 2016.
- [121] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Nadia Polikarpova. AutoProof: Auto-active functional verification of object-oriented programs. In *Tools and Algorithms for the Construction and Analysis of Systems — 21st International Conference, TACAS*

2015, volume 9035 of *Lecture Notes in Computer Science*, pages 566–580. Springer, 2015.

- [122] Jan L. A. van de Snepscheut. *What computing is all about*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.
- [123] Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. Theorem proving for all: Equational reasoning in Liquid Haskell (functional pearl). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018*, pages 132–144. ACM, 2018.
- [124] Philip Wadler. Linear types can change the world! In Manfred Broy and Cliff B. Jones, editors, *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods*, page 561. North-Holland, April 1990.
- [125] Markus Wenzel. *Isabelle/Isar — A versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, Technische Universität München, 2002.
- [126] Thomas Wies, Damien Zufferey, Siddharth Krishna, and Ruzica Piskac. GRASShopper.
<https://cs.nyu.edu/~wies/software/grasshopper>.

Index

$\succ$ (exceeds, reduces to), 70
($\emptyset$) (auto-init type), 389, 402
($==$) (equality-supporting type), 146, 159, 202, 309
 $\rightarrow$ (arrow type), 280
 $!!$ (disjoint sets), 378, 380, 383
 $|_I$
 multiset cardinality, 295, 300
 sequence length, 68, 279
 set cardinality, 394, 395
 $!in$ (not in), 125, 299, 379, 383
 $:|$ (assign such that), 314, 395, 429
 $:=$ (assignment), 9, 429
 $;$ (statement terminator), 30, 429
 $\Leftrightarrow$ (iff, equivalence), 163, 362 associative, not chaining, 450
 $_$ (don't-care name), 176, 333
 $_{[..]}$ (take), 279
 $_{[..]}$ (sequence from array), 279
 $_{[\dots]}$ (subsequence), 267, 279
 $_{[\dots]}$ (drop), 279
 $_{[\dots]} := _$
 map update, 124, 404

`seq` update, 349, 389

above, 3

Abrial, Jean-Raymond, 205

absolute difference, 250, 304

abstract over implementation, 195

abstract syntax tree (AST), 90

abstraction, xviii, 189

abstraction field, 388

vs. abstraction function, 388

abstraction function, 195, 204, 208, 388

access control, of declaration, 204

accumulator parameter, 148

Ackermann function, 74, 81, 110

Ackermann, Wilhelm, 81

ACL2, x, 81, 132

adjoint, lower and upper, 54

Agda programming language, 94, 131, 230

aggregate object, 364

invariant of, 385

aggregate statement (`forall`), 334, 430

alpha-renaming, 439

amortized running time, 199

antisymmetric relation, 165

Append function (`List`), 139

- AppendSumUp lemma, 269
- arithmetic overflow, 319
- array, 275
 - as object, 355
 - constructor, 333
 - conversion to multiset, 295
 - conversion to sequence, 279
 - copy elements, 331
 - elements, 276
 - fields of, 355
 - increment elements, 329, 335
 - index, 275
 - initialization, 326, 333
 - length, 276
 - multi-dimensional, 278
 - reverse elements, 332, 335
 - rotate elements, 332, 333, 335
- array** type, 276
- array2** type, 278
- arrow type, 280
- as** operator (cast), 352
- assert** statement, 10, 22
 - in expression, 151
- assign-such-that statement, 314
- assignment might update ... not in ... modifies clause, 325

associative relation, 141, 166, 434

associativity, of xor, 37

At function (List), 144

auto-active verification, x

auto-init type, 376, 389, 402

automatic induction, 100, 132

 turn off, 100

automation, limits of, xv

auxiliary variable, 23

axiom, 99, 142, 173

Axiom of Dependent Choice, 71

Back, Ralph-Johan, 62, 320

Backhouse, Roland C., xx

bag (multiset), 177

balanced tree, 210

below, 3

binary heap, 210

Binary Search algorithm, 288

 Java bug, 319

binary search tree, 207, 403

binary tree, 89, 207, 403

Bird, Richard S., 319

block statement with specifications, 273

Blockchain exercise, 200, 230, 358

- BlueCount function, 85, 122
- body, of declaration, 192
- body-less method or lemma, 99
- bookkeeping notation for program proofs, 36, 242
- Boyer, Robert S., 132, 309
- Boyer-Moore theorem prover, 132
- Braun tree, 210, 229, 230
 - balance property, 210, 212
 - key insight, 214
- break** statement, 282
- Burrows, Michael, 62

- calc** statement, 106
 - default operator, 132
 - in expression, 151
- call might violate context's modifies clause, 377
- Canyon Search algorithm, 304
- captured parameter, 373
- cardinality, *see* $|\_|$, 431
- cardinality of a subset, lemma, 412
- chain (sequence), 67
- chaining operator, xii, 21, 108, 450
- chapter dependency graph, xi
- char**, 352
- characterization of a predicate, 27

- Charity programming language, 94
- checksum, 352
- ChecksumMachine class, 353
- Chlipala, Adam, 230
- class, 351
 - client's view, 357
- class invariant, 374
- client, 13
- coercion, 62
- CoffeeMaker example, 365
- Cohen, Edward, xx
- Coincidence Count algorithm, 294
- color datatype, 338
- commutative relation, 140, 166, 434
- commutativity
 - of multiset intersection, 300
 - of xor, 37
- compilable equality, 146, 201
 - special case, 203
- compile a program, 6
- compiled declaration, 19
- ComputeFib method, 258
- ComputePower method, 264
- ConcurrentModificationException, 423
- conjecture, 99

- conjunction, 3, 433
- consecutive-formula notation, 36
- const**, 142, 360, 372
- constituent object, 365
 - abandoning, 372
 - capturing, 373
 - releasing, 373
- constructive logic, 444
- constructor
 - anonymous, 353
 - for datatype variant, 84
 - for object, 352
 - two-phase, 376, 423
- constructor**, 352
- container, iterator over, 413
- contour lines, 301
- contract, 5, 14, 23
- contrapositive, 199, 310, 313, 437
- Coq proof assistant, 230, 349
- Count function (List), 177
- Count function (**seq**<T>), 309
- counting occurrences, 309
- Counting Sort algorithm, 349
- Cube method, 264

Dafny, name origin, 374

datatype, 83

- algebraic, 84

- coinductive, 94

- destructor, 87

- discriminator, 87

- inductive, 84

- member, 87

- variant, 83, 84

De Morgan's Law, 435

De Morgan's Law for Quantifiers, 285, 287, 442

debug failed verification attempt, 301

debugging with **assert**, *see* verification debugging

decreases clause, 67, 173

- decreases** *, 81

- default, 79

- default for **while true**, 286

- default for function with **reads**, 401, 409

- default for loops, 250

- inspecting default, 80, 250, 291, 401, 409

dependent type, xviii, 230

descending chain condition, 71

Design by Contract, 23

destructor, for datatype, 87

deterministic behavior, 21

- Dijkstra, Edsger W., xvi, xx, 61
- discriminator, for datatype, 87
- disjunction, 3, 435
- display expression, 432
- distribution over $\Rightarrow$, 437
- div and mod, properties, 171
- Drop function (`List`), 142
- Dutch National Flag algorithm, 337, 340, 345
 - connection with Quicksort, 345
- dynamic frames, 5, 374, 382, 423
 - standard idiom, 382, 384, 385, 396

- earliest insertion point, 289
- Eiffel programming language, x, xii, 5, 23, 204, 385
- elision rule for type arguments, 158
- ensures** clause, 14
 - multiple, 16
- entry point of program (`Main`), 6
- enumeration type, 89, 94
- equality, built-in, 201
- equality-supporting type, 201
- error message, how to interpret, 11
- ESC/Modula-3, 385
- Ettinger, Ran, xiii
- Euclid programming language, 23, 336

Event-B, 205
existential quantifier, *see exists* expression
exists expression, 284, 442
export declaration, 192
export set, 192, 204, 357, 427
 inconsistent, 193
exporting an imported module, 197
ExtensibleArray class, 396
extensionality, xxii, 219, 294, 301, 448
extrinsic specification, 140, 159

F*, x, 23, 94, 131, 204, 230
FastExp method, 266
Feijen, Wim H. J., xx, 131
Fibonacci function, 66, 257, 282
field
 immutable, 360
 of class, 351
Filliâtre, Jean-Christophe, xiii, 6
Find function (List), 146
flowchart language, 61
Floyd logic, 29, 61
Floyd, Robert W., xv, 29, 61, 349
for loop, xx
forall expression, 209, 282, 441

- forall** statement, 334, 430
- formal methods, 2
- formula, 27
- Four Ferries, 131
- Frama-C framework, x, 349
- frame, xxi, 5, 321
 - for array, 321
 - for object, 351, 354
- frame bylaw, 322
 - new**, 324, 354
 - object constructor, 356
 - objects, arrays, 322, 355
 - reading state, 325, 355
- frame invariant, 392, 399, 406
- frame specification, typical, 374
- fresh** predicate, 325, 336
 - fresh**(Repr), 371, 374
 - fresh**(Repr - old(Repr)), 372, 374
- “friend” class or module, 204
- function, 17
 - body-less, arbitrary, 142
 - mathematical, 17
 - with boolean result, 19
- function as a value, 280
- function** declaration, 17

function result in its own **ensures**, 140, 182, 192, 353, 380, 382

function vs. method, 21, 22

functional programming, xii, 5

functions are deterministic, 17

Galois connection, 54

garbage collection, 373

generic type, 89

ghost declaration, 19, 429

ghost field, 384, 388, 390, 395

ghost parameter, 309, 311, 315, 318

ghost variable, 315, 317, 318, 382, 385

 using with compiled variables, 19, 311

goal-oriented approach, 32

Gobra verifier, x, 6

GRASShopper, x

Gries, David, xxi, 23, 319

half-open interval, 339

halt, program, 23

hash code, simple, 352

healthiness conditions, 61

heap, 5

heap property, 210, 212

heap-allocated storage, 321
Heapsort algorithm, 349
Hegner, Eric C. R., xix, xxi, 256, 273
Hoare triple, 29, 61
Hoare, C. A. R., xvi, 29, 30, 61, 349
Hofstadter Female and Male sequences, 78
Hofstadter G sequence, 70
hold (be true), 4, 433
how to count, 145, 160

I.H. (induction hypothesis), 119
IDE (integrated development environment), 5
idempotent function, 122, 125
idempotent relation, 434
identity of an object, 351
idiomatic specification, 382
if statement vs. **if-then-else** expression, 57, 430, 432
if-case statement, 13, 65, 296, 308, 456
iff (if and only if), 163
immutable field, 360, 372, 390
 advantage in framing, 372, 417
immutable state, 389
imperative programming, xii, 5, 33
implication, 437
implicit dynamic frames, xxii

- import** declaration, 191, 427
- import relation, 191
 - acyclic, 191, 427
- in situ*, 337
- in-parameter, 9
- induction, 101
 - automatic, 100, 132
 - Principle of, 101
- {:induction **false**}, 100
- induction hypothesis, 101, 131
- induction schema, xvi
- infinite descending chain, 67, 71
- infinite looping, 247
- infinite recursion, 64
- informal methods, 6
- informal reasoning, 296
- information hiding, xviii, 14, 189, 191, 204, 367
 - for objects, 355
- injective function, 93, 162
- inline assertion, 10, 152
- inline function, 333
 - with specifications, 334
- Insertion Sort algorithm, 179, 349
- insufficient memory, 23
- insufficient reads clause, 368

int type, 23

integers, bounded vs. unbounded, 23, 319

integrated development environment (IDE), x

interval

- consecutive, 310

- half-open, 15, 265, 267, 310

- inclusive, 15

intrinsic specification, 140, 158

intrinsic vs. extrinsic, 225

intrinsic-extrinsic spectrum, 225, 226

intuitionistic logic, 444

invariant, 207, 229, 233

- change as we go along, 343

- of data structure, 210, 230

- of object, 351, 354

invariant clause, multiple, 250

involution, 118, 122, 151, 433

irreflexive partial order, 71

irreflexive relation, 71

Isabelle/HOL, 131, 188

Isar proof language, 131

IsEmpty predicate (Queue), 203

IsMin predicate, 209

Iterator class, 413

JML, xii, 23, 385
Joshi, Rajeev, xiii
justify defeat, 282

Kaldewaij, Anne, xxi
Kassios, Ioannis T., 384
KeY system, x, 349
.Keys (**map**<K, V>), 92

lambda expression, *see* inline function
Lamport, Leslie, 205
Language of the Included Miracle (LIM), 62
Larch, 23
Law of Excluded Middle, 434, 437, 444
Law of the Excluded Miracle, 61
Lean theorem prover, 131
Leijen, Daan, xiii
Leino, Kaleb, iv, xiii
lemma, 95

- lemma** declaration, 96
- body-less, as axiom, 142
- in expression, 151
- precondition vs. implication in postcondition, 163, 199, 310

.Length (**array**<T>), 276

- Length function (`List`), 138
- `.Length0` and `.Length1` (`array2<T>`), 278, 301
- Less predicate (`set<int>`), 407
- let binding, 33
- let expression, 92
 - in postcondition, 171, 182
- lexicographic order, 72
- lexicographic tuple, 72
 - length, 154
 - vs. tuple type, 75
- linear arithmetic, 266
- Linear Search algorithm, 281
- `LinearSearch` method, 280
- LiquidHaskell, x, 94, 131, 230
- `List` datatype, 91, 137
 - as stack, 416
- `ListLibrary` module, 190
- logical formula, 27
- logical function, 23
- logical quantifier, 281, 282
- Longest Plateau algorithm, 319
- loop, 235
 - alternative to, 256
 - body, 236, 241
 - body, omitting, xix, 236

- effect, 236
- frame, 241, 246, 328
- guard, 235
- implementation, 241
- index, 238
- leap to the answer, 243, 249
- proof obligations, 250
- specification, xix, 236

loop design technique

- always, 293
- omit a conjunct, 253
- programming by wishing, 254
- replace constant by variable, 258
- replace constant by variable in precondition, 287
- to establish conjunction, 253
- weaken, 308
- yet to be done, 266

loop design technique, use of

- always, 293, 307, 328, 339, 348, 364, 420
- omit a conjunct, 253, 303
- programming by wishing, 253, 260
- replace constant by variable, 258, 283, 289, 292, 327, 456
- replace constant by variable in precondition, 287, 303
- weaken, 308
- yet to be done, 265, 271, 421

- loop invariant, 61, 235
 - as loop specification, xix
 - avoiding, 333
 - maintaining, 243, 244, 250
 - that implies loop guard, 249, 286
 - what has been done, 266
 - what's yet to be done, 266
- loop proof obligation
 - maintaining loop invariant, 247
 - termination, 247
 - using the loop, 247
- loop-use proof obligation, 236, 250, 255

- Müller, Peter, xiii, 6
- machine integer, 319
- Madoko, xiii
- Main method (program entry point), 6, 190
- Majority Vote algorithm, 309
- map** type, 91
 - cardinality, 412
 - domain subtraction (-), 404, 405
 - keys (.Keys), 92
 - merge (+), 407, 412
 - values, 92
- map** update ($\_[_ := \_]$), 404

- `map[]` (empty map), 404
- `map[_ := _]` (singleton `map`), 406, 407
- `maplet` (singleton `map`), 407
- `match` expression, 85
- `match` statement, 119, 339
- mathematical inconsistency, 65, 81
- matrix, 278
 - copy elements, 332, 335
 - initialization, 327, 335
 - transpose, 332, 336
- meddling kids, 65
- member, of class, 351
- memoization, 358
- Merge function, 182
- Merge Sort algorithm, 181, 188, 348
- `method` declaration, 9
- method vs. function, 21, 22
- Meyer, Bertrand, 23
- Min function (binary search tree), 411
- Min method, 16, 292, 322, 323
- minimum, 304, 341, 411
 - of an array, 292
- modifies**
 - evaluated in pre-state, 324
- modifies** clause, 322, 336

- empty, 322
- for loops, 328
- Modula-3 programming language, 204, 385
- modular arithmetic, 319
- modular design, 204
- module, 189, 427
 - client's view, 195
 - interface, 195
 - interface, testing, 197
 - nested, 228
- modus ponens*, 437
- Monahan, Rosemary, xiii
- Moore, J Strother, 132, 309
- Morgan, Carroll, xix, xxi, 6, 62, 205, 320
- Morris, Joseph M., 62
- Moy, Yannick, xiii
- multiplication, difficult automation, xxii
- multiset, 177, 188, 294, 338
 - advice for proofs, 218
 - conversion from sequence, 295
- multiset**(_) function, 295
- multiset**{_} display expression, 295, 298
- mutating method, 384

- Nagini verifier, x

- namespace, 190
- nat** type, 66
- Natural Merge Sort algorithm, 188
- natural number, 66
- negation, 433
- negative test case, 86
- Nelson, Greg, 62
- new** array, 276
 - with initializer, 402
- new** object, 353
- new**; statement, 377, 383, 422
- non-ghost (compiled) declaration, 19
- non-linear arithmetic, xxii, 266
- non-termination, 81
 - proving, 249
- nondeterministic behavior, 13, 21
- null**, 398, 405
- null** vs. **Option**, 405
- nullable reference type, 397, 398, 405, 418, 423

Oberon-2 programming language, 204

object, 351

object constructor

- first phase, 376

- second phase, 377, 383, 422

- object invariant, 374
- oceanize function, 86, 122
- Okasaki, Chris, 204
- old expression, 323, 336, 353
 - common mistake, 324
- one-point rule, 283, 287
- opaque, 14, 21, 140
- OpenJML verifier, x, 81, 159
- operator precedence, 164, 430
- Option type, 358, 398, 403, 414, 423
- Ordered predicate (`List<int>`), 176
- out-parameter, 9
- Owicki, Susan, 23
- ownership, of object, 384

- pair (2-tuple), *see* tuple
- Parnas, David L., 204
- Parno, Bryan, xiii
- partial command, 62
- partial correctness, 64
- partial expression, 58
- partial order, 71
- Partition method, 345
- pattern
 - as LHS, 170

- nested, 176
 - use of `_`, 176
- payload, 83, 94, 118, 157, 207
- Peano numbers, 173
- Peano, Giuseppe, 173
- pentagonal number, 279
- permutation
 - sorting specification, 177, 188
- pivot element in Quicksort, 345
- post-increment, 272
- postcondition, 14
- Power function, 264
- pre-decrement, 272
- precondition, 14
- precondition vs. antecedent in postcondition, 163
- predicate, 19, 27
 - predicate** declaration, 19
- predicate result in its own **ensures**, 380
 - common mistake, 380
- predicate subtype, xviii, 230
- PrependSumDown lemma, 268
- primitive recursive function, 81
- print** statement, 190, 353
- priority queue, 207
- private (access modifier), 204

private implementation decision, 200, 204

programming by wishing, 253, 260

Project function (`List`), 178

proof calculation, 106

proof obligation

- $\mathcal{WP}$ vs. $\mathcal{SP}$, 51, 53, 61

- at call site, 14, 55

- at return point, 14, 131

- divide using `if`, xii

- for termination, 67, 76, 78

- in backward reasoning, 35

- of `:` (assign such that), 314

- of `assert`, 10, 22, 50

- well-definedness, 58

proof pearl, 188

proof-authoring construct, xii

`provides` clause, 192, 204

proving an implication, 173

Prusti verifier, x, 6

public (access modifier), 204

PVS system, 230

quantifier, taming, xxii

querying method, 384

queue, 194

Quicksort algorithm, 343, 349

Radix Sort algorithm, 349

range split, 283, 287

read frame, 325, 336, 347

reads clause, 325, 336

receiver parameter (**this**), 353

reference type, 278

refinement type, xviii, 230

Repr ghost field, 368, 382

- immutable, 372, 390, 415

- lower bound, 368, 373

- specifications about, 371

- update, 382

representation set, 367, 374

- disjoint, 400

requires clause, 14

return statement, 281

reveals clause, 192, 204

Reverse function (**List**), 149

ReverseColors function, 86, 122

Reynolds, John C., xxi

run a program, 6

Saddleback Search algorithm, 319

SameSums lemma, 268

scope, 190

Selection Sort algorithm, 341

separation logic, xxii, 384

seq constructor (**seq**(_, _)), 393

seq type, 68

seq update (**[**_ **:=** _**]**), 389

seq<**char**> (**string**), 352

sequence, 278

- advice for proofs, 298

- concatenation, 279

- constructor, 334

- conversion from array, 279

- conversion to multiset, 295

- display expression, 279

- length, *see* **|_|**

- update expression, 389

set

- pairwise disjoint, 380

- writing proofs about, 395

set cardinality, *see* **|_|**

set comprehension, 394, 395

set<**object**>, 368

short-circuit operator, 59, 369, 434, 446, 450

- shunting, 437
- sibling declaration, 190, 228
- signature, of declaration, 192
- simple frame
 - of array, 321
 - of constituent object, 365, 375, 383
- simultaneous assignment, 13, 37, 40, 92, 263, 429
- skeleton of a data structure, 207
- Slope Search algorithm, 301
- Smith, Graeme, xiii
- Snoc function (`List`), 139
- sort key, 178, 188, 340
- sort specification, 188, 338, 343
 - same elements, 338, 341, 343, 348
 - same-elements postcondition, 177, 188
- sortedness, expressed
 - for `List<int>`, 176
 - for `array<int>`, 288, 294, 304
- SPARK, x, 23, 159, 204, 205
- Spec#, 23, 384, 385, 423
- specification
 - change as we go along, 294, 343, 344, 356
 - form, importance of, 188, 223
 - technique to say a property never changes, 389
- specification construct, xii

- specification statement, 205, 320
- specified blocks, 273
- Split function, 182
- SplitPoint predicate, 343, 345, 456
- SquareFib method, 260
- SquareRoot method, 254
- stability equation, 178
- stability, sorting specification, 178
- stable sort, 178, 188
- Stainless verification system, x
- strict partial order, 71
- strict total order, 71
- strictly stronger, weaker, below, above, 3, 431
- string**, 352, 360
- stronger, 3
- stronger predicate, 3
- strongest postcondition, 28, 32
- structural inclusion, 88
- Stump, Aaron, 230
- subsequence notation, 267, 279
- subset type, xviii, 230
- substitution, 33, 283, 439
 - capture-avoiding, 440
- Sum function (List<int>), 359
- SumDown function, 268, 357

SumUp function, 267, 357

Swamy, Nikhil, xiii

Take function (List), 142

termination

- intrinsic proof, 173

- of loops, 247

termination metric, 67

test harness, 17, 86, 197, 199, 201, 209, 230, 353, 356, 370, 389, 397, 415

testing

- by writing lemmas, 139

- module interface, 197

- specification, 17, 289

theorem, 96

this, 22, 353

- implicit, 353

“through” vs. “to”, 15

TimSort algorithm, 349

Tinelli, Cesare, xiii

TLA⁺, 205

“to” vs. “through”, 15

tokenizer, 359

total correctness, 30, 64

total expression, 58

- total order, 165
- trace, program, 2
- transitive relation, 71, 140, 163
- transparent, 18, 21, 140
- Treesort 3 algorithm, 349
- trichotomous relation, 165
- trombone fanfare of disappointment, 282
- tuple
 - built-in datatype, 411
 - constructor, 168, 182, 200, 215, 411, 422
 - deconstructing, 168, 411
 - lexicographic (in **decreases** clauses), 72
 - structural inclusion, 75
 - type, 167, 215, 411
- two-phase constructor, 376, 423
- two-state lemma, 345
- two-state predicate, 323, 344, 347
- twostate predicate**, 344, 349
- type argument, elided, 158
- type characteristic
 - (==), 159, 202
 - ($\emptyset$), 389, 402
- type parameter, 89, 280, 428
- type signature, 192
- type synonym, 201, 202

- unary number, 161
- underspecification, 15, 361
- uninterpreted (arbitrary) function, 267
- Union function (binary search tree), 407, 412
- unit element, 125, 141, 142, 166, 433, 435
- universal quantifier, *see* **forall** expression

`Valid()`, *see* validity predicate

- validity predicate, 211, 230, 354, 374
 - as invariant, xviii
 - as object invariant, 359, 360, 366, 382, 385, 390, 392, 417, 423
 - in postcondition of `Valid()`, 380, 390
 - intentionally not maintaining, 361
 - intrinsic vs. extrinsic, 211

value type, 278

van de Snepscheut, Jan L. A., xxi

variable capture, 439

VCC verifier, 384

VDM, 23

VeriFast verifier, x, 384

verification debugging, 378

- with **assert**, 297, 330
- with **assert** in a function, 412

Viper, 6, 384

von Wright, Joakim, 62

weaker, 3

weaker predicate, 3

weakest precondition, 28, 32

Welfare Crook algorithm, 319

well-defined expression, 144, 155, 159, 434

 postcondition, 155, 159

well-founded relation, 71, 247

 built into Dafny, 71

what's-yet-to-be-done invariant, 266

what-has-been-done invariant, 266, 309

what-we-have approach, 32

while loop, 235, 236

Whiley programming language, x

Why3 verifier, x, 188, 349

WhyML programming language, 6, 23, 384

width, of operators, 164

wish, 253

worker method, precondition, 418

working backward, xvi, xx, 28, 40, 253, 259

$\mathcal{WP}$ vs. $\mathcal{SP}$, 53

wraparound integer, 319

write frame, 322, 336, 347

xor, 36

zero element, 142, 433, 435