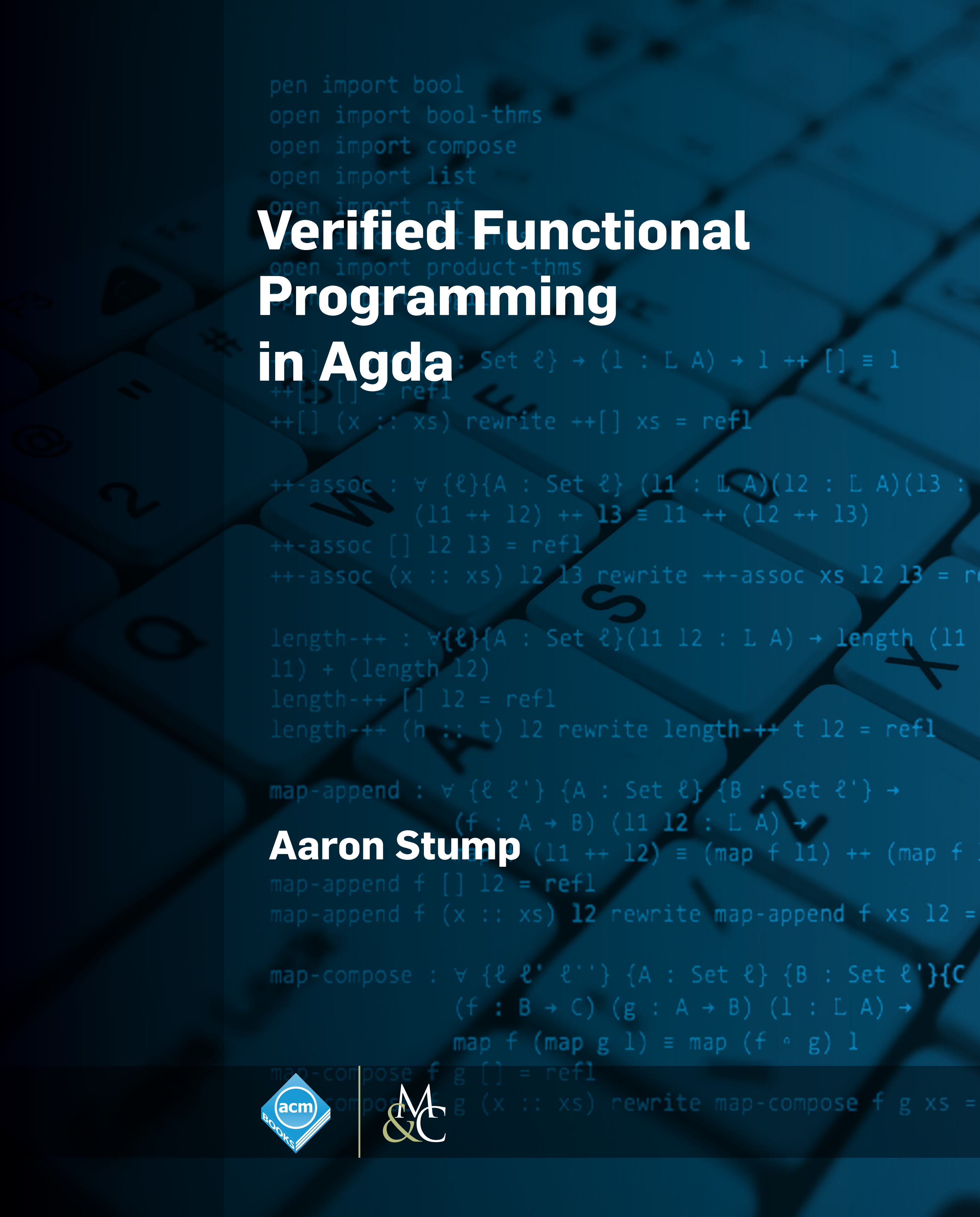

Verified Functional Programming in Agda

Aaron Stump

Verified Functional Programming in Agda

Verified Functional Programming in Agda

Aaron Stump

The University of Iowa

ACM Books #9

Verified Functional Programming in Agda

Aaron Stump

books.acm.org
www.morganclaypool.com

ISBN: 978-1-97000-127-3 hardcover
ISBN: 978-1-97000-124-2 paperback
ISBN: 978-1-97000-125-9 ebook
ISBN: 978-1-97000-126-6 ePub
Series ISSN: 2374-6769 print 2374-6777 electronic

DOIs:	10.1145/2841316	Book
	10.1145/2841316.2841317	Preface
	10.1145/2841316.2841318	Chapter 1
	10.1145/2841316.2841319	Chapter 2
	10.1145/2841316.2841320	Chapter 3
	10.1145/2841316.2841321	Chapter 4
	10.1145/2841316.2841322	Chapter 5
	10.1145/2841316.2841323	Chapter 6
	10.1145/2841316.2841324	Chapter 7
	10.1145/2841316.2841325	Chapter 8
	10.1145/2841316.2841326	Chapter 9
	10.1145/2841316.2841327	Chapter 10
	10.1145/2841316.2841328	Appendixes
	10.1145/2841316.2841329	References/Index

A publication in the ACM Books series, #9
Editor in Chief: M. Tamer Özsu, *University of Waterloo*
Area Editor: Laurie Hendren, *McGill University*

First Edition

10 9 8 7 6 5 4 3 2 1

For Madeliene and Seraphina

Contents

Preface

Programming languages are one of the absolutely essential parts of Computer Science. Without them, programmers could not express their ideas about how to process information in all the amazing and intricate ways necessary to support the continuing explosion of applications for computing in the 21st century. Most students of Computer Science, certainly in the U.S., start learning a programming language right at the beginning of their studies. We may go on to learn analysis of sophisticated algorithms and data structures, graphics, operating systems, databases, network protocols, robotics, and many other topics, but programming languages are the practical starting point for Computer Science.

Programming is an intellectually challenging task, and creating high-quality programs that do what they are supposed to do, and that can be maintained and adapted over time, is very difficult. In this book, we will focus on how to write programs that work as desired. We will not be concerned with maintainability and adaptability directly (though of course, these topics cannot be totally ignored when writing any significant piece of software). The state of the art for achieving correct software for most industrial programming at major corporations is testing. Unit testing, system testing, regression suites—these are powerful and effective tools for rooting out bugs and thus ensuring software quality. But they are limited. A famous aphorism by Turing award winner Edsger Dijkstra is that "Program testing can be used to show the presence of bugs, but never to show their absence!" [1972]. For there is only a finite number of tests you can run, but in practice, the number of possible inputs or system configurations that should be tested is infinite (technically, modern computers only have finitely many states since they have only finite memories, but the number is beyond astronomical, and should be considered infinite for practical purposes).

Verified Programming

This book is about a new way to program, where instead of just testing programs on a finite number of inputs, we write mathematical proofs that demonstrate their correctness on all possible inputs. These proofs are written with a specific syntax, in the programming language itself. The compiler checks those proofs when it type-checks and compiles the code. If all the proofs are correct, then we can be sure that our program really does satisfy the property we have proved it has. Testing can still be an important way of assuring code quality, because writing proofs is hard, and often it may be more practical just to do some testing. But we have now a new tool that we can use in our quest to build high-quality software: we can *verify* software correctness by writing mathematical proofs about it.

While these ideas have been known in the Programming Languages and Verification research communities for many decades, it has proven difficult to design languages and tools that really make this vision of verified programming a feasible reality. In the past 10–15 years, however, this has changed. New languages and tools, building on new research results in programming language theory and new insights into the design of effective languages for verified programming, have made it realistic, for the first time in the history of programming, to prove properties of interesting programs in reasonably general-purpose programming languages, without a PhD in logic and without a huge and tedious engineering investment.

Agda

In this book, we will learn how to write verified programs using one such advanced programming language called Agda. Agda is a research language developed over a number of years at Chalmers University of Technology in Gothenburg, Sweden. The current version, Agda 2, was designed and implemented by Ulf Norell as part of his doctoral dissertation [Norell 2007]. Since then, it has been improved and developed further by a long list of authors, including (from the Agda package information):

> Ulf Norell, Nils Anders Danielsson, Andreas Abel, Makoto Takeyama, Catarina Coquand, with contributions by Stevan Andjelkovic, Marcin Benke, Jean-Philippe Bernardy, James Chapman, Dominique Devriese, Peter Divanski, Fredrik Nordvall Forsberg, Olle Fredriksson, Daniel Gustafsson, Alan Jeffrey, Fredrik Lindblad, Guilhem Moulin, Nicolas Pouillard, Andrés Sicard-Ramírez and many more.

The basic mode for working with Agda is to use the EMACS text editor as a rudimentary Integrated Development Environment (IDE). Text is entered into a buffer in EMACS, and then with commands like Ctrl+c Ctrl+l (typed in sequence), the text in that buffer is sent to Agda for type checking and proof checking. An amazing feature of Agda and its EMACS editing mode is that they support Unicode characters. So rather than just using alphanumeric characters and basic punctuation, you can use a much larger character set. Unicode supports almost 110,000 characters. Usually, we write programs using basic Latin characters, like those shown in Figure 1.

003	004	005	006	007
0 0030	@ 0040	P 0050	` 0060	p 0070
1 0031	A 0041	Q 0051	a 0061	q 0071
2 0032	B 0042	R 0052	b 0062	r 0072
3 0033	C 0043	S 0053	c 0063	s 0073

Figure 1 **Basic Latin characters.**

1008	1009	100A	100B
𐂀 10080	𐂐 10090	𐂠 100A0	𐂰 100B0
𐂁 10081	𐂑 10091	𐂡 100A1	𐂱 100B1
𐂂 10082	𐂒 10092	𐂢 100A2	𐂲 100B2
𐂃 10083	𐂓 10093	𐂣 100A3	𐂳 100B3
𐂄 10084	𐂔 10094	𐂤 100A4	𐂴 100B4

Figure 2 **Linear B ideograms.**

Imagine the cool programs we could write if we were allowed to use Linear B ideograms (part of Unicode, though we would need to install a special font package to get these into Agda), such as those shown in Figure 2.

Okay, maybe we do not actually want to program with ancient symbols for agricultural commodities, but it is nice to be able to use mathematical notations, particularly when we are stating and proving properties of programs. So the Unicode support in Agda is a great feature, though it takes a little practice to get used to entering text in Unicode. The Agda mode for EMACS recognizes certain names that begin with a backslash, as you type them. It then converts them to the special symbol associated with those names. For example, if you type `\all` into EMACS in Agda mode, EMACS will magically change it to ∀ (the logical symbol that is used when expressing that "for all" elements of some particular type, some formula is true).

Functional Programming

There is another important aspect of programming in Agda: Agda is a **functional programming** language. Functional programming is a programming paradigm, different from the object-oriented or imperative paradigms. There are two meanings to "functional programming."

The weak sense. Functions are central organizing abstractions of the language. Furthermore, they can be defined anonymously, passed as input arguments, or returned as output values.

The strong sense. Every function that you define in the programming language behaves like a mathematical function, in the sense that if you call it with the same inputs, you are guaranteed to get the same outputs. Such languages are called *pure* functional languages. Languages that are functional in the weak but not the strong sense are sometimes called *impure* functional languages.

While many languages can be argued to be functional languages in the weak sense, very few are functional in the strong sense. For how can a function like `gettimeofday()` behave like a mathematical function? It takes no arguments, and if everything is working properly, it should *always* give you a different answer! It would seem to be impossible to support such functions in a pure functional programming language. Other examples include input/output and *mutable* data structures like arrays where a function can, as a side effect of its operation, change which values are stored in the data structure. Amazingly, handling side effects in a pure functional language is possible, and we will see a hint of this in this book. Thoroughly exploring how side effects are supported in a pure functional language is, however, outside the scope of our subject here. The pure functional language for which this issue has been most deeply resolved is Haskell. Agda is implemented in Haskell, and imports a small portion of Haskell's solution to the problem.

Functional programming is a beautiful and natural fit for the task of verified programming. We know reasonably well how to reason about mathematical functions, using basic principles and tools of logical reasoning. Examples include reasoning about “and”, “or”, “not”, “implies”, and other propositional connectives; equational reasoning, like concluding that $f\ x$ must equal $f\ y$ if $x = y$; mathematical induction (which we will review later in the book); and more. If programs can essentially be viewed as mathematical functions—as they can in a pure functional language—then all these tools apply, and we are well on our way to a reasonably simple approach to formal reasoning about the behavior of programs.

In Agda, as we will see, there is one further restriction on programs. Not only must they be pure—so using mutable data structures like arrays, or input/output, must be done carefully (the way Haskell does)—Agda also must be able to tell that they are guaranteed to terminate on all inputs. For many programs we will want to write this is easily done, but sometimes it is a nuisance. And, of course, there are some programs like Web browsers or operating systems that are intended to run indefinitely. In addition, one can prove that for any scheme to limit programs just to ones that terminate on all inputs (these are called *uniformly terminating*), there will be programs that actually do terminate on all inputs but which that scheme cannot recognize. This is a deep result, with a nontrivial proof, but it shows that we cannot hope to recognize exactly the uniformly terminating programs. So Agda will prevent us from writing some programs we might wish to write, though it allows us to disable terminationg checking globally or for specific functions. In practice, for many natural programs we’d like to write, uniform termination can be easily checked by Agda. So it is not as burdensome as you might think, to write only uniformly terminating programs.

One feature of Agda which we will not explore in this book is the lazy representation of infinite data structures, like infinite lists and trees. This is an important part of functional programming in Haskell, to which Agda compiles. Such data structures are known as *coinductive*, and one writes *corecursive* programs operating on them. The theory and practice of corecursive programming has received significant attention in recent years [Abel and Pientka 2013]. Agda currently supports several distinct approaches to this style of programming. Partly because there is not (quite?) yet a settled solution for this problem, at least in Agda—and partly because the subject is yet more advanced than what is covered here—I have chosen not to include coinduction and corecursive programming in this book.

Types

At the heart of Agda’s approach to verified functional programming is its type system. In mainstream typed programming languages like Java, we have basic datatypes like `int` and `boolean`, and also container types like `List<A>`, for lists of elements of type

A. Agda's type system is much more expressive, and supports two uses of types that are not found in industrial languages.

Types indexed by data values. Instead of just having a type `list A` for lists of elements of type `A`, Agda's data-indexed types allow us to define a type `vector A n` of vectors of length `n` containing elements of type `A`. The index `n` to the `vector` type constructor is a natural number ($\{0, 1, 2, \ldots\}$) giving the length of the vector. This makes it possible to specify interesting properties of functions, just through the types of their inputs and outputs. A very well-known example of this is the type of the `append` operation on vectors, which is essentially:

$$\texttt{append} : \texttt{vector}\ A\ n \rightarrow \texttt{vector}\ A\ m\ \rightarrow \texttt{vector}\ A(n+m)$$

The length of the output vector is equal to the sum of the lengths of the input vectors—and we can express this using the length index to `vector`. Whenever a property like this relationship between input and output lengths for vector append is established through the types of the inputs and outputs to the function, that property is said to have been **internally verified**. The verification is internal to the function definition itself, and is confirmed when the function's code is type-checked. We will see several examples of this in Chapter 5.

Types expressing logical formulas. Agda considers specifications like "for any list `L`, the reverse of the reverse of a list `L` is equal to `L`", which we will write as $\forall L \rightarrow \texttt{rev}(\texttt{rev}L) \equiv L$ in Agda, to be just fancy kinds of types. To prove a formula F expressed as a type, we just have to write a program that has type F. That is because, as mentioned before, Agda requires all programs to terminate on all inputs. So if we can write a function which takes in a list L and produces a proof that `rev` (`rev` L) equals L, then that function is guaranteed to succeed for any list L: the function cannot run forever, or throw an exception (Agda does not have these), or fail for any other reason. Such a function can thus be regarded as a proof of the formula. In this case, we say that the type is **inhabited**: there is some program that has that type. If we prove a property about a function like `rev` by showing a type like this one is inhabited, then we have **externally verified** the function. We have written our code (for `rev`, in this case), and then externally to that code we have written a proof of a property about it.

Agda's support for both internal and external verification of functions opens up a really interesting domain of possibilities for writing verified code. We will see examples of these two verification styles in what follows, both of which rest on Agda's powerful type system.

A Few Questions

Why Study This?

Some readers of this book may be enthusiastically interested in learning how to write verified functional programs and how to use Agda. But maybe someone else is wondering why this is worthwhile. It may sound at least somewhat intellectually interesting, but shouldn't we be learning how to write software for real-world applications like Web services, or maybe embedded systems like pacemakers or power substations controllers, or electronic control units (ECUs) in cars?

In the not-so-distant future, it is my prediction and firm belief that a significant body of industrial practitioners will be using a language similar to Agda to implement such systems. Why? Because all those example systems just mentioned have been shown to be vulnerable to malicious computer attacks in recent years (see, e.g., Fu and Blum [2013]). And in the arms race against black-hat hackers, the ability to prove that code is absolutely correct (with respect to some specification) is too powerful a defensive weapon to remain unused. At some point, the costs of security-critical bugs—which can be extremely high, perhaps unquantifiable (imagine a voting-machine bug that allows someone to swing the election for U.S. President)—will be so high and the tools for language-based verification will have become usable enough that mainstream industry will begin to adopt them.

A second answer is that functional programming itself is gaining momentum in industry. A significant number of major companies, including Twitter, LinkedIn, and quite a few more, have adopted the Scala programming language. Scala is a descendant of Java incorporating functional-programming features and idioms. It also has quite a complex type system. OCaml and Haskell are two other influential functional languages that are seeing increased use in industry. Certain sectors, notably finance, value the more concise and mathematical expression of computational ideas which is often possible with functional code, as this makes it easier for domain experts (e.g., securities traders) to communicate with the developers providing their tools. And because the paradigm of pure functional programming provides a much simpler model for concurrent computation than imperative programming—it is easy to reason about concurrent executions of functions that have no side effects, while reasoning about side effects quickly becomes very complicated with concurrent code—Haskell is finding applications at companies like Facebook, where massively concurrent coding is the norm [Metz 2015]. And, basic ideas from functional programming appear in many other very mainstream languages. For example, JavaScript and Python both have anonymous functions, which (as already mentioned) are central to functional programming. Apple's Swift language for programming iOS devices is another highly visible industrial language adopting important ideas from functional programming [Eidhof et al. 2014].

Agda is, in some ways, the most advanced functional programming language in existence, and so if you learn how to program in Agda, you will have a very strong foundation for programming in other languages that use the idioms of functional programming. Beyond such pragmatic considerations, though, I hope you will find that it is an amazing experience to write code and be able to prove definitively that it satisfies its specification.

What Background Is Needed?

This book is intended for students of Computer Science, either undergraduates or beginning graduate students without significant prior background in verified functional programming. Students who already know functional programming will likely find they can move quickly through the explanations of basic programs on pure functional data, and focus on how to write proofs about these. But knowledge of functional programming is not expected; indeed, the book seeks to provide an introduction to this topic. The main background I am assuming is knowledge of some other programming language (not Agda), such as Python, Java, C/C++, or other mainstream languages. Since most Computer Science majors, at least in the U.S, learn Java, I will often try to explain something in Agda by relating it to Java.

While verification is based heavily on logic, the contents of an undergraduate course in discrete math should be sufficient. We will review basic forms of logical reasoning like propositional reasoning, equational reasoning, use of quantifiers, and induction, in the context of Agda. Agda is based on constructive logic, which differs somewhat from the usual classical logic considered in discrete math courses. Constructive proofs give explicit computational evidence for formulas. So in Agda, doing logic is literally just programming. For students of Computer Science, logical reasoning will likely be more natural in Agda than in previous coursework.

Finally, in order to have some interesting programs to write and reason about, I am assuming knowledge of basic algorithms and data structures like lists and trees. An undergraduate algorithms course should be sufficient for this.

How Do I Install Agda?

You can find instructions on how to install Agda on the Agda wiki:
http://wiki.portal.chalmers.se/agda/pmwiki.php

Agda is implemented in Haskell, and runs on all major platforms (Mac, Linux, and Windows). As mentioned earlier, the usual mode of programming in Agda is to write and type-check Agda code in the EMACS text editor, which you may also need to install and configure. See the instructions on the Agda wiki for all this. In addition to being implemented in Haskell, the Agda compiler can translate Agda programs into Haskell, which can then be compiled by the Haskell compiler to native executables. It is not necessary to know Haskell to learn Agda, though occasionally we will see code where ideas from Haskell or connections from Agda to Haskell are used.

The code in this book has been confirmed to work with the latest version of Agda (2.4.2.2 at the time of this writing).

What Is the Iowa Agda Library?

The chapters of this book are organized around different ideas and abstractions from functional programming, as implemented in a library I have been working on for several years, called the Iowa Agda Library (IAL). Agda has its own standard library, which you can download from the Agda wiki. I have learned a lot about verified programming in Agda from this standard library, and incorporated several ideas and idioms from it into my version. For purposes of learning the language, though, I have found the Agda standard library a bit too advanced, and somewhat challenging to navigate. You can obtain version 1.2 of the IAL, on which this book is based, via subversion (or browse it with a Web browser) here:
http://svn.divms.uiowa.edu/repos/clc/projects/agda/ial-releases/1.2

If you access this via subversion, use username “guest” and password “guest”. This library currently has a completely flat structure, so all files mentioned in the chapters to come can be found in that directory.

What Is Not Covered?

Though Agda is absolutely central to this book, nevertheless this is not intended to be a book about Agda. There are important features of Agda that are not covered. For the biggest example, I do not attempt to cover coinductive types—which are types for lazy infinite data structures. As noted previously, there are several competing proposals for how to support these in Agda, and so it seems premature to focus on one. There are other features like irrelevant arguments and instance arguments that I am not covering, and no doubt many Agda tricks and features known only to more advanced users than I. Records and modules are covered, though there is surely more to say about them in Agda than you will find here. For those new to functional programming or new to type theory—the intended audience for this book—omission of these more advanced features should not detract from the essential topics. Another very important idea in pure functional programming is that of a *monad*. This is an elegant abstraction which is all but essential for larger-scale pure functional programming, to handle side effects in as clean and elegant a way as is currently known. Monads are central to functional programming in Haskell, for example. Regrettably, I have not managed to cover that abstraction in this book. See tutorials on Haskell like the one of Hudak et al. for more on monads [Hudak et al. 2000].

What Are Some Other Materials about Agda?

On the "Documentation" page of the Agda wiki, you can find a number of other tutorials about Agda, such as that by Norell and Chapman [2009]. Research papers presenting some of the features not covered in this book include the following.

- *On the Bright Side of Type Classes*, on a feature called instance arguments, which is intended to replace type classes used in languages like Haskell to support ad-hoc polymorphism (also known as operator overloading) [Devriese and Piessens 2011].
- *MiniAgda: Integrating Sized and Dependent Types*, on integrating *sized types*, a type-based approach to termination-checking, with dependent typing as found in Agda [Abel 2010].
- *Irrelevance in Type Theory with a Heterogeneous Equality Judgement*, showing how to integrate existing ideas on irrelevant terms (terms which can be erased during compilation or even to check definitional equality) with Agda [Abel 2011].

What Is Some Other Reading on Type Theory More Generally?

Type theory, upon which Agda is based, is a deep field of theoretical Computer Science, going back many decades. Indeed, its roots are in constructive logic and lambda calculus, fields which have been under development since the early 20th century; see Cardone and Hindley, and Troelstra for interesting historical perspectives on these topics [Cardone and Hindley 2006, Troelstra 2011]. The particular branch of type theory on which Agda has been developed is that of Martin-Löf type theory, developed by the Swedish philosopher and logician Per Martin-Löf; see Nordström, Petersson, and Smith for a readable and informative introduction to Martin-Löf type theory [Nordström 1990]. Developing on a different branch of the type theory tree, but with many connections and influences, is Coq, a theorem prover based on type theory, developed at INRIA, France; see Bertot and Castéran for a thorough and accessible introduction [Bertot and Castéran 2004]. Several other excellent books either about Coq or strongly based on Coq are *Software Foundations* and *Certified Programming with Dependent Types* [Pierce et al. 2015, Chlipala 2011].

Why This Book?

Students of programming owe the authors of Agda a big debt of gratitude for creating a very nice tool based on a powerful and expressive theory of verified functional programing called **type theory**. In the course of trying to learn more about verified functional programming in Agda, I found that while there are a number of helpful tutorials and other sources online for learning Agda, and a very responsive email list, a more systematic description of how to do verified functional programming in Agda was missing. I am hoping that this book will contribute to the Agda community by helping fill that gap. There are many others who have much more expertise than I do on advanced functional programming idioms, and on using Agda. Nevertheless, I hope that you, the reader, can benefit from what I have learned about this topic, and that you will find verified programming in Agda as interesting and enjoyable as I have.

A Note for Instructors

As you well appreciate, learning programming generally requires doing a substantial amount of it and, of course, the same can be said for theorem proving. While I have included modest handfuls of exercises at the ends of the chapters, you will likely need much more for the graded work of a semester-long course. My strategy for devising homework based on this book, for the past three semesters (at the time of writing) at The University of Iowa, has been as follows.

The first two, or maybe three, homeworks are focused on very small programming and proving tasks, along the lines of: write a function to return every other element of a list, and then prove that the length of the resulting list is less than or equal to that of the starting list. One can begin with booleans or other non-recursive datatypes, and non-recursive programs and proofs over them and then move on to recursive datatypes and recursive programs (inductive proofs). The IAL has many examples, which unfortunately you may feel you have to exclude from re-assigning for homework, since the solutions are publicly available in the IAL. I find it is still not too hard to come up with a good array of small problems along these lines.

Subsequent homeworks become more like mini projects, using existing code in the IAL (tries, Braun trees, etc.) to solve problems on roughly the scale as the case study in Chapter 8 (maybe a little smaller or a little more ambitious, depending on student preparation and level). I find that these problems generally will not involve much if any external verification. At best, there is some lightweight internal verification, and most of the focus is just on pure functional programming. In my case, I suspect this is largely due to the challenge of identifying a reasonably interesting property or invariant to try to prove statically, which is still within reach for students to be able to complete with a reasonable amount of time and effort. It is not that such properties do not exist. Rather, I have found that they are hard enough to find as part of the task of creating an interesting assignment, that I tend not to manage to do it. Furthermore, many undergraduates have enough trouble grasping pure functional programming that further challenges are unnecessary.

Acknowledgments

I am grateful to the undergraduate students of CS:3820 (previously 22c:111) at The University of Iowa for feedback that helped improve this book, and contributions to the IAL. I'd especially like to recognize John Bodeen for his work on the gratr parser generator discussed in Chapter 7, and Tom Werner for outstanding contributions to the IAL. Thanks to my colleagues at The University of Iowa, particularly my chair Alberto Segre, for supporting my experiment to introduce Agda into the required undergraduate CS curriculum. I am also grateful to Laurie Hendren of the ACM Books Editorial Board for her support of this project, and to Diane Cerra of Morgan & Claypool for her work in bringing the manuscript to publication. Without them this would still just be a draft PDF! Thank you to the reviewers of an earlier draft of this manuscript: Jesper Cockx, Ulf Norell, and Stephanie Weirich. Their superbly informed constructive criticism greatly improved this book. I am grateful to my wife for her support as I finished up this book in the summer of 2015. I also wish to acknowledge the support of the Catholic community of St. Wenceslaus, Iowa City. AMDG.

1 Functional Programming with the Booleans

There are few datatypes in computer science simpler than the booleans. They make a good starting point for learning functional programming in Agda. The Agda code displayed below can be found in `bool.agda` in the Iowa Agda Library (IAL; see the preface for the URL for this). We will spend this chapter studying this code. In the next chapter, we will begin our study of theorem proving in Agda, with theorems about the boolean operations we define in this chapter.

1.1 Declaring the Datatype of Booleans

If you open `bool.agda` in EMACS, you should see text that starts out like this:

`bool.agda`

```
module bool where

open import level

----------------------------------------------------------
-- datatypes
----------------------------------------------------------

data 𝔹 : Set where
  tt : 𝔹
  ff : 𝔹
```

The main thing we want to consider here is the declaration of the boolean datatype 𝔹, but there are a few things to look at before that.

Module declaration. Every Agda file needs to contain the definition of a single module. Here, the module is declared by the line

```
module bool where
```

The name of the module is `bool`, which is required to match the name of the file (as it does, since the file is `bool.agda`). Modules are organizational units that can contain code and type declarations, like packages in Java. They can be nested, and can even have parameters which you have to fill in when you import the module. We will see more about Agda's module system later (for example, Section 5.4).

Import statements. To use code contained in another file, we have to import it using `import` and then the name of the module provided by that other file. If we just said `import level`, we would be allowed to use the code and types defined in `level.agda`, but we would have to qualify them with the prefix "`level.`" By writing `open import level`, we are telling Agda we wish to import the `level` module and make use of all the types and code defined there, without writing this qualifying "`level.`" prefix. We will see what the `level` package is providing that is needed for the definition of the booleans.

Comments. Agda follows Haskell in using `--` to indicate the start of a comment that runs to the end of the line (similar to `//` in Java). To comment out a whole block of Agda code, one can put `{-` in front of that block, and `-}` at the end of it. This comments out the whole region, similar to `/*` and `*/` in Java. One advantage of Agda and Haskell's notation over Java's is that comments in Agda and Haskell can be nested, like this:

```
{-
-- a nested comment {- and another -}
-}
```

Now let us consider the definition of the boolean datatype 𝔹:

```
data 𝔹 : Set where                                    bool.agda
  tt : 𝔹
  ff : 𝔹
```

The `data` keyword signals the beginning of a datatype declaration. Datatypes in Agda are for constructing immutable data. As mentioned in the Preface, Agda is a pure functional programming language, and mutable datatypes like arrays or updatable reference cells are not supported (directly) by the language. So you could think of this boolean datatype as similar to the immutable class `Boolean` in Java, for objects containing a boolean value.

Following `data`, we have the name 𝔹 for the new datatype we are defining (for the booleans). There is no backslash command that will enter this symbol in EMACS by default. Appendix C explains how to add some new key combinations to EMACS which will let you include this symbol by typing `\bb`. Then we have "`: Set`", which indicates that 𝔹 itself has type `Set`. Every expression in Agda has a type, and this expression "`Set`" is the type for types. So 𝔹 has type `Set`.

1.1.1 Aside: Type Levels

You might wonder: if every expression in Agda has a type, and if `Set` is the type for types, does that mean that `Set` is its own type? The answer is a bit surprising. It is known that if one literally makes `Set` the type of itself, then the language becomes nonterminating (we can write diverging programs, which is disallowed in Agda). This remarkable result is discussed in a paper by Meyer and Reinhold [1986]. To avoid this, Agda uses a trick: an infinite hierarchy of type levels. `Set` really stands for `Set 0`, and for every natural number $n \in \{0, 1, 2, \ldots\}$, the expression `Set` n has type `Set` $(n + 1)$. This is why we needed to import the `level` module at the beginning of this file, to define the datatype for levels n in `Set` n. For the definition of the boolean datatype 𝔹, all we really need to know is that `data 𝔹 : Set` is declaring a datatype 𝔹, which is itself of type `Set` (which is an abbreviation for `Set 0`).

1.1.2 Constructors `tt` and `ff`

Returning to the declaration of the booleans: we next have the `where` keyword, and finally the definitions of the **constructors** `tt` and `ff`, both of type 𝔹. These constructors are like constructors in object-oriented programming, in that they construct elements of the datatype. But they are primitive operations: there is no code that you write in Agda that defines how `tt` and `ff` work. The Agda compiler will translate uses of `tt` and `ff` to code, which actually ends up creating values in memory to represent true and false. But within the Agda language, constructors of datatypes are primitive operations, which we are to assume create data for us in such a way that we can inspect it later via pattern matching.

1.1.3 Aside: Some Compiler Directives

If you are looking in `bool.agda`, after this declaration of the type 𝔹, you will see some of these comments:

`bool.agda`

```
{-# BUILTIN BOOL  𝔹  #-}
{-# BUILTIN TRUE  tt  #-}
{-# BUILTIN FALSE ff #-}

{-# COMPILED_DATA 𝔹 Bool True False #-}
```

Comments written with the delimiters "`{-#`" and "`#-}`" are directives to the Agda compiler. In this case, they are telling Agda that 𝔹 and its constructors are the definition to use for a built-in notion of booleans that the compiler is supporting. There is also a directive with `COMPILED_DATA`, telling Agda to compile the 𝔹 datatype as the Haskell datatype `Bool`, which has constructors (in Haskell) `True` and `False`.

1.2 First Steps Interacting with Agda

As noted in the Preface, Agda is usually used from within the EMACS text editor. Let us see a few initial commands for interacting with Agda in EMACS. You can find more such commands from the Agda menu item in the EMACS menu bar.

Loading a file. If you are viewing an Agda file (ending in `.agda`) in EMACS, you can tell Agda to process that file by typing Ctrl+c Ctrl+l. This will then enable the following two operations.

Checking the type of an expression. After Agda has successfully loaded a file using the previous command, you can ask it to tell you the type of an expression by typing Ctrl+c Ctrl+d, then typing the expression, and hitting enter. For example, you can ask Agda to see the type of `tt` this way. You will see 𝔹 in response.

Evaluating an expression. To see what value an expression computes to (or *normalizes* to, in the terminology of type theory), type Ctrl+c Ctrl+n, then the expression, and then hit enter. The file must be loaded first. A rather boring example is to ask Agda to normalize `tt`. This expression is already a final value (no computation required), so Agda will just print back `tt`. We will see more interesting examples shortly.

1.3 Syntax Declarations

Let us continue our tour of `bool.agda`. Shortly below the declaration of the 𝔹 datatype of booleans, you will see declarations like this (and a few more):

`bool.agda`

```
infix  7 ~_
infixl 6 _xor_ _nand_
infixr 6 _&&_
infixr 5 _||_
```

These are syntax declarations. Agda has an elegant mechanism for allowing programmers to declare new syntactic notations for functions they wish to define. For example, we are going to define a boolean "and" operation && just a little below in this file. This operation is also called **conjunction**, and its arguments are called **conjuncts**. We would like to be able to write that operation in infix notation (meaning, with the operator between its arguments), like this:

```
tt && ff
```

Furthermore, we would like Agda to understand that the negation operator ~ grabs its argument more tightly than the conjunction operator &&. So if we write

```
~ tt && ff
```

we would like Agda to treat this as

```
(~ tt) && ff
```

as opposed to

```
~ (tt && ff)
```

We do this by telling Agda that the negation operator ~ has higher precedence than &&. Each operator can be assigned a number as a precedence with a syntax declaration like the ones displayed previously. In this case, we have told Agda that negation has precedence 7 and conjunction has precedence 6:

```
infix  7 ~_
infixr 6 _&&_
```

Operators with higher precedence grab their arguments more eagerly than operators with lower precedence. Syntax declarations can also specify the *associativity* of the operators. If we have an expression like

```
tt && ff && tt
```

should this be viewed as

```
(tt && ff) && tt
```

or as

```
tt && (ff && tt)
```

Here, the syntax declaration for conjunction uses the `infixr` keyword, which means that conjunction associates to the right. So the second parse displayed is the one that is chosen. Of course, for conjunction it does not matter which associativity is used, since either way we get the same result. Actually, this is proved in `bool-thms2.agda` as `&&-assoc`. We will learn how to prove theorems like this in Agda in the next chapter. Some operators are not associative, of course. An example is subtraction, since $(5 - 3) - 1$ has value 1, while $5 - (3 - 1)$ has value 3. But it is up to us to declare how Agda should parse uses of infix operators like conjunction or subtraction. This is a matter of syntax. Proving that either associativity (left or right) gives the same result is a matter of the semantics (meaning) of the operators.

Why are there underscores _ written next to these operators? These tell Agda where the arguments of the operator will appear. By writing `_&&_`, we are telling Agda that conjunction's arguments will appear to the left and the right, respectively, of the conjunction symbol `&&`. So we are saying that `&&` is an infix operator by writing it `_&&_`. Similarly, by writing `if_then_else_`, we are saying that the three arguments of this operator—the boolean guard, the then-part, and the else-part—appear in the usual positions with respect to the `if`, `then`, and `else` symbols. So we can write

```
if x then y else z
```

and Agda will understand that we are applying the `if_then_else_` operator (which we will define in a moment) to the three arguments x, y, and z. In fact, Agda will treat the nice *mixfix* notation (where arguments can be declared to appear in and around various symbols constituting the operator name) displayed just above as convenient shorthand for applying the `if_then_else_` function to the three arguments in succession:

```
if_then_else_ x y z
```

Occasionally it is necessary to refer to a mixfix function like `if_then_else_` separately from its arguments. In this case, we actually have to include the underscores in the name. So for example, you can ask Agda what the type of an expression like `tt && ff` is using Ctrl+c Ctrl+d (as mentioned in Section 1.2), or you can also ask it for the type of the negation operator `~_` by typing Ctrl+c Ctrl+d and then literally `~_`. You will see that the type is

```
𝔹 → 𝔹
```

We will consider next what this type means and how operations like conjunction can be defined in Agda. One final note before we do: syntax declarations can appear anywhere in the file, as long as that file does indeed define the functions in question.

1.4 Defining Boolean Operations by Pattern Matching: Negation

Just below the precedence declarations in `bool.agda`, which we were just examining, we have the definitions of a number of central functions related to the booleans. Let us look first at the simplest, negation `~_`. The definition is this:

bool.agda

```
~_ : 𝔹 → 𝔹
~ tt = ff
~ ff = tt
```

Definitions of functions in Agda, however complex they may be, follow a basic pattern. First we declare the type of the function. This is accomplished by the line

```
~_ : 𝔹 → 𝔹
```

This declares that the `~_` function takes in a boolean and returns a boolean. In Agda, types that look like $A \rightarrow B$ are the types of functions which take in input of type A and produce output of type B. So 𝔹 → 𝔹 is the type for functions from 𝔹 to 𝔹. And negation certainly has this type, since if we give it boolean value `tt`, we expect to get back boolean value `ff`; and vice versa. In fact, this is exactly what is expressed by the rest of the definition of the `~_` function:

```
~ tt = ff
~ ff = tt
```

Functions in Agda are defined by sets of equations. The left side of each equation is required to be a pattern which matches a set of function calls, namely, the ones that could be made with the function we are defining. Patterns can contain constructors (like `tt` and `ff` here for type 𝔹), as well as variables, which are any other symbols that do not already have a definition at this point in the code. In this case, which is very simple, the first equation describes the function call `~ tt`, and the second describes the function call `~ ff`. These are the only two possible function calls, since negation just takes in a single boolean argument, and there are only two possible values for this argument. The patterns do not need to use variables to cover all the possible function calls to `~_`. Agda requires that the patterns used in the equations defining each function cover all possible function calls to that function. We will get an error message if we leave some function calls uncovered by the patterns we have written. For example, if you delete the second equation for `~_` and then reload the file with Ctrl+c Ctrl+l, Agda will complain with an error message like this:

```
Incomplete pattern matching for ~_. Missing cases:
  ~_ ff
when checking the definition of ~_
```

So, we need to include both those equations for our definition of negation. To test out the definition, try normalizing the term `~ ~ tt` (for example) in Agda, by typing Ctrl+c Ctrl+n, then `~ ~ tt`, and then enter. Agda will tell you that this term normalizes (evaluates) to `tt`.

1.4.1 Aside: Space around Operators and in Files

If you ask Agda to normalize `~~tt`, you will get a nasty error message:

```
1,1-5
Not in scope:
  ~~tt at 1,1-5
when scope checking ~~tt
```

What is Agda complaining about? Isn't `~~tt` treated the same as `~ ~ tt`, which we just asked Agda to evaluate a moment ago? The answer, which is obvious from this little experiment with `~~tt`, is no. One quirk of Agda that takes a little getting used to is that user-defined operators need to be separated from most other symbols with at least one space. Otherwise, Agda thinks that you are really using just a single strange operator name. So the error message Agda is giving us for `~~tt` makes sense: it says that `~~tt`, viewed as a single symbol, is not in scope. It thinks that `~~tt` is the name of some defined operation. So it is parsing `~~tt` as a single symbol, rather than two nested function calls of the `~_` operator.

This requirement of whitespace around operators can lead to lots of irritating errors for beginners with Agda. But by imposing this requirement, the language designers have made it feasible to use a very rich set of names for operations. For example, we might actually want to use `~~tt` as the name for something, maybe a theorem about applying negation twice. Also, this requirement is a small price to pay for Agda's flexible and natural system of user-defined notations. So it is worth getting used to.

One other note about spacing in Agda files. Agda resembles Haskell and some other languages like Python in attaching semantic significance to the whitespace used in your programs. Generally, indenting a further or lesser amount in Agda code can change the meaning of what you are writing. The basic rule is that parts of your Agda code that are linguistically parallel should be at the same indentation level. For example, consider again our declaration of the boolean datatype:

```
data 𝔹 : Set where
  tt : 𝔹
  ff : 𝔹
```

For example, you will get a parse error from Agda if you change the indentation like this:

```
data 𝔹 : Set where
  tt : 𝔹
   ff : 𝔹
```

This is because constructor declarations are linguistically parallel, and so they should be at the same indentation level. How far you indent does not matter, but you should indent at least one space in this case (and generally for nested subexpressions), and you must indent with the same number of spaces. For another example, if you change the definition of negation like this

```
~_ : 𝔹 → 𝔹
~ tt = ff
 ~ ff = tt
```

so that the second equation is at a different indentation level than the first, you will get a parse error.

1.5 Defining Boolean Operations by Pattern Matching: And, Or

Let us look next, in `bool.agda` in the IAL, at the definition of conjunction (boolean "and") `_&&_`, shown in Figure 1.1. We see from the first line that the type of conjunction is

```
𝔹 → 𝔹 → 𝔹
```

The arrow operator is grouped to the right, so this type is actually parsed by Agda as

```
𝔹 → (𝔹 → 𝔹)
```

This type says that `_&&_` takes in a boolean (the first conjunct) and returns a function of type 𝔹 → 𝔹. This function is waiting for a boolean (the second conjunct), and then it will return the "and" of those two booleans. We know from our discussion of syntax declarations in Section 1.3 that this operator can be given two arguments in infix notation, like this:

```
tt && ff
```

```
_&&_ : 𝔹 → 𝔹 → 𝔹
tt && b = b
ff && b = ff
```

`bool.agda`

Figure 1.1 The definition of boolean conjunction.

or in prefix notation, if we include the underscores as part of the operator name:

```
_&&_ tt ff
```

Because the type for conjunction is parsed by Agda as $\mathbb{B} \to$ ($\mathbb{B} \to \mathbb{B}$), we can also just give the conjunction operation a single argument:

```
_&&_ tt
```

If you ask Agda what type this has by typing Ctrl+c Ctrl+d and then `_&&_ tt`, you will see that it has type $\mathbb{B} \to \mathbb{B}$. This makes sense, given that conjunction has type $\mathbb{B} \to$ ($\mathbb{B} \to \mathbb{B}$). We have given it a boolean, and now we get back a boolean-to-boolean function. So `_&&_` takes in two booleans and returns a boolean, but it can take its booleans one at a time.

Now let us look at the code (repeated from Figure 1.1) defining the conjunction operation:

```
tt && b = b
ff && b = ff
```

We are defining the behavior of conjunction using equations as we did for negation (Section 1.4). Conjunction takes in two boolean arguments, and since there are two independent possibilities for each of these, you might expect that we would have four defining equations for conjunction. Indeed, we could just as well have defined it this way:

```
tt && tt = tt
tt && ff = ff
ff && tt = ff
ff && ff = ff
```

But this is unnecessary. As the definition from `bool.agda` demonstrates, we can define conjunction just by looking at the first argument. If the first argument is true (`tt`), then the truth of the conjunction is completely determined by the truth of the second argument. So we use the first equation

```
tt && b = b
```

to express that. Here the symbol b is treated by Agda as a variable, since it is not defined previously. Variable names can be any symbols, with very few restrictions. They cannot be constructor names, naturally. The symbols used can involve almost any Unicode characters, though a few pieces of built-in punctuation cannot be used: dot, semicolon, and parentheses.

If the first argument is false (`ff`), on the other hand, then there is no hope for this poor conjunction: it will turn out false independent of the value of the second argument. That is expressed by the second equation:

```
ff && b = ff
```

Again, we are using a variable to range over the two possibilities for the second argument. Notice that the fact that we used the same variable `b` in both equations is not semantically significant: pattern variables like these have scope only on the right-hand side of their equation. So we could have used `b1` in the first equation and `b2` in the second, for example.

The definition of disjunction (boolean "or"), which comes next in `bool.agda`, is quite similar:

```
_||_ : 𝔹 → 𝔹 → 𝔹                                    bool.agda
tt || b = tt
ff || b = b
```

Here the situation is dual to that of conjunction. We again just consider the first argument. If it is `tt`, then this disjunction wins the jackpot: it will come out `tt` regardless of the value of the second argument. And if the first argument is `ff`, then the only hope is that the second argument is `tt`; the disjunction's value is now determined completely by that second argument.

You can test out these definitions by evaluating something like

```
tt && (ff || tt)
```

using Ctrl+c Ctrl+n in Agda (you should get `tt` in this case).

1.6 The if-then-else Operation

As mentioned in our discussion of syntax declarations in Section 1.3, we can define our own if-then-else operation in Agda. In many languages, this operation is built in, but we will define it ourselves. Here is the code:

```
if_then_else_ : ∀ {ℓ} {A : Set ℓ} → 𝔹 → A → A → A      bool.agda
if tt then t else f = t
if ff then t else f = f
```

As we have seen already, every function definition in Agda starts by stating the type of the function (using a colon ':'), and then gives the actual code for the function. The complicated part of this definition is the type, so let's come back to that in a moment. The equations defining the functions are very easy to understand:

```
if tt then y else z = y
if ff then y else z = z
```

An if-then-else expression like `if` x `then` y `else` z consists of a guard x, a then-part y, and an else-part z. The equations are pattern-matching on the guard. They say that if the guard of the if-then-else expression is true (`tt`), then we should return the then-part y (whatever value that has). And if the guard is false (`ff`), then we should return the else-part z. So if we ask Agda to evaluate a term like

```
if tt then ff else tt
```

using Ctrl+c Ctrl+n, we will get the then-part, which is `ff` in this case.

Now the interesting thing about this if-then-else operator is that it is **polymorphic**: it will work just fine no matter what the type of the then-part and the else-part are, as long as those parts have the same type. So when we define the natural numbers in Chapter 3, we will be able to ask Agda to evaluate terms like

```
if tt then 21 else 32
```

(and get the value 21 back). We cannot do that at the moment while inspecting the file `bool.agda`, because the natural numbers are defined in `nat.agda`, which is not imported by `bool.agda`. So Agda does not yet know about the natural numbers in the file `bool.agda`. But the main point is that as long as x has type $\mathbb{B}$ and y and z have the same type, the expression `if` x `then` y `else` z is typeable.

And this is what the type for the if-then-else operator says:

```
if_then_else_ : ∀ {ℓ} {A : Set ℓ} → 𝔹 → A → A → A
```

Let us read through this type slowly. First, we have the symbol ∀, which can be used to introduce some variables which may be used later in the type expression itself. To type this symbol into EMACS, you type `\all`. Here, we are introducing ℓ (which is typed with `\ell` in EMACS) and A. Agda always requires a type for each variable that we introduce, but a lot of times it can figure those types out. Here, Agda sees that we are using ℓ as an argument to `Set`, and it is able to deduce that ℓ is a type level. We mentioned those toward the end of Section 1.1. The point of these variables is to express polymorphism. The if-then-else operation works for any type A, from any type level ℓ. Now, we won't use type levels too much in this book, but it is a good idea to make our basic operations like if-then-else as polymorphic as we reasonably can in Agda. So we should make them both type and level polymorphic. By putting those type variables in curly braces, we are instructing Agda to try to infer their values. We will see more about this feature later. So when we call the function, we do not actually have to tell Agda what the values for ℓ and A are. It will try to figure them out from the types of the other arguments.

After these type variables expressing polymorphism of if-then-else, we have a part of the type expression that hopefully is less exotic:

```
𝔹 → A → A → A
```

This is expressing the basic idea that if-then-else takes in three inputs, which are a boolean (𝔹, for the guard) and then an *A* value (for the then-part) and another *A* value (for the else-part); and returns an *A* value. This works for whatever type *A* Agda has inferred for the then- and else-parts.

1.6.1 Some Examples with if-then-else

If we ask Agda to type-check (with Ctrl+c Ctrl+d) the expression

```
if tt then ff else ff
```

it will figure out that the type variable *A* must be 𝔹, since the then- and else-parts are both booleans. And since 𝔹 is at type level 0, the variable ℓ can be inferred to be the value for level zero. The type for the whole term is 𝔹, of course.

Slightly more interestingly, we can use then- and else-parts which have functional type. If we ask Agda to type-check

```
if tt then _&&_ else _||_
```

then it will reply that the type is 𝔹 → 𝔹 → 𝔹. We are using the conjunction and disjunction operators without their two arguments, so we have to put the underscores as part of their names. And we are using conjunction as the then-part, and disjunction as the else-part, of this expression. Since conjunction and disjunction both have the same type (𝔹 → 𝔹 → 𝔹), this is a legal use of if-then-else.

Finally, to demonstrate a little bit of just how polymorphic Agda's type system is, we can ask Agda to type-check the following:

```
if tt then 𝔹 else (𝔹 → 𝔹)
```

This is certainly not something you can write in a mainstream programming language. This expression evaluates to the type 𝔹 if its guard is true (which it is) and to the type 𝔹 → 𝔹 otherwise. We know that 𝔹 has type `Set` (which is just an abbreviation for `Set 0`, as mentioned in Section 1.1). It turns out that 𝔹 → 𝔹 has type `Set`, also. In fact, in Agda, if *A* and *B* both have type `Set` ℓ, then so does `A → B`. So both the then-part (𝔹) and else-part (𝔹 → 𝔹) of this if-then-else expression have type `Set`, and so the whole expression has that type, too. And that is what Agda will tell you if you check the type of the expression with Ctrl+c Ctrl+d.

1.7 Conclusion

In this chapter, we have seen some basics of programming in Agda in EMACS.

- We saw the declaration of the boolean datatype 𝔹, with its constructors `tt` and `ff`. To type mathematical symbols like 𝔹 in EMACS, you use **key sequences** that begin with a backslash, like `\bb` for 𝔹 (see Appendix A).
- Agda's flexible mixfix notation system allows you to specify where arguments go in and around parts of an operator name. This lets us declare infix operators (where the arguments are on either side, and the operator is in between them), as well as fancier things like if-then-else. When the function is used by itself, we have to write the underscores as part of its name, which indicate where the arguments go.
- We saw how to define functions on the booleans using pattern matching and equations. The examples we saw were negation `~_`, conjunction `_&&_`, and disjunction `_||_`.
- We can ask Agda to do a couple interesting things for us, from EMACS.
  - Check the type of an expression by typing Ctrl+c Ctrl+d, then the expression, and then Enter.
  - Normalize (evaluate) an expression by typing Ctrl+c Ctrl+n, then the expression, and then Enter.
- Function types like 𝔹 → 𝔹 describe functions. The type 𝔹 → 𝔹 → 𝔹 describes functions which take in two booleans as input, one at a time, and produce a boolean output.
- We also saw an example of a polymorphic function, namely, if-then-else. Its type quantifies over a type level ℓ and a type A at that type level, and then takes in the expected arguments:

```
if_then_else_ : ∀ {ℓ} {A : Set ℓ} → 𝔹 → A → A → A
```

In the next chapter, we will start to see the feature that really distinguishes Agda from programming languages as most of us have known them until now: theorem proving.

Exercises

1.1. Evaluate the following expressions in Agda within EMACS. For this, the easiest thing to do is to open and load the `bool.agda` file first:

(a) `tt && (ff xor ~ ff)`

(b) `~ tt && (ff imp ff)`

(c) `if tt xor tt then ff else ff`

1.2. What is the type of each of the following expressions? (You can check these in Agda with Ctrl+c Ctrl+d?)

(a) `tt`

(b) `if tt then ff else tt`

(c) `_&&_`

(d) $\mathbb{B}$

1.3. Pick a function defined in `bool.agda` like `_xor_`, `_imp_`, or another, to redefine yourself. You can do this in a new file called `my-bool.agda` that begins this way, where *X* should be replaced by the name of the function you will redefine (e.g., `_xor_`, with the underscores):

```
module my-bool where

open import bool hiding ( X )
```

1.4. Define a datatype `day`, which is similar to the $\mathbb{B}$ datatype but has one constructor for each day of the week.

1.5. Using the `day` datatype from the previous problem, define a function `nextday` of type `day` → `day`, which given a day of the week will return the next day of the week (so `nextday Sunday` should return `Monday`).

1.6. Define a datatype `suit` for suits from a standard deck of cards. You should have one constructor for each of the four suits: hearts, spades, diamonds, and clubs.

1.7. Define a function `is-red`, which takes in a `suit` as defined in the previous problem and returns `tt` if and only if the suit is a red one (`hearts` and `diamonds`).