

Verified Functional Programming in

Aaron Stump
Computer Science
The University of Iowa
Iowa City, Iowa, USA

August 19, 2015

For Madeliene and Seraphina

Contents

Preface	1
1 Functional Programming with the Booleans	11
1.1 Declaring the Datatype of Booleans	11
1.1.1 Aside: type levels	13
1.1.2 Constructors <code>tt</code> and <code>ff</code>	13
1.1.3 Aside: some compiler directives	13
1.2 First steps interacting with Agda	14
1.3 Syntax declarations	14
1.4 Defining boolean operations by pattern matching: negation	16
1.4.1 Aside: space around operators and in files	17
1.5 Defining boolean operations by pattern matching: and, or	18
1.6 The if-then-else operation	20
1.6.1 Some examples with if-then-else	22
1.7 Conclusion	23
1.8 Exercises	24
2 Theorem Proving with the Booleans	25
2.1 A First Theorem about the Booleans	25
2.1.1 Types as formulas: the Curry-Howard isomorphism	26
2.1.2 Proofs as programs: more Curry-Howard	27
2.1.3 Going deeper: Curry-Howard and constructivity	28
2.2 Universal Theorems	32
2.2.1 Proving the theorem using pattern matching	32
2.2.2 An alternative proof and implicit arguments	33
2.2.3 Using holes in proofs	35
2.3 Another Example, and More On Implicit Arguments	35
2.4 Theorems with Hypotheses	38
2.4.1 The absurd pattern	39
2.4.2 An alternative proof	41
2.4.3 Matching on equality proofs	42
2.4.4 The <code>rewrite</code> directive	44
2.5 Further examples	44
2.6 Conclusion	46

2.7	Exercises	47
3	Natural Numbers	49
3.1	Peano Natural Numbers	50
3.2	Recursive Definition of Addition	51
3.3	Simplest Theorems about Addition	53
3.4	Associativity of Addition and Working with Holes	57
3.5	Commutativity of Addition	60
3.6	Multiplication	62
3.7	Some Basic Theorems about Multiplication	63
3.7.1	Right distributivity of multiplication over addition	63
3.7.2	Commutativity of multiplication	65
3.7.3	Associativity of multiplication	66
3.8	Arithmetic Comparison (Less-Than)	67
3.8.1	Dotted variables	70
3.9	An Equality Test for Natural Numbers	71
3.10	Conclusion	73
3.11	Exercises	73
4	Lists	75
4.1	The List Datatype and Type Parameters	75
4.2	Basic Operations on Lists	77
4.2.1	Length of a list	77
4.2.2	Appending two lists	77
4.2.3	Mapping a function over a list	79
4.2.4	Filtering a list	80
4.2.5	Removing an element from a list, and lambda abstractions	81
4.2.6	Selecting the n 'th element from a list, and the <code>maybe</code> type	82
4.2.7	Reversing a list (easy but inefficient way)	82
4.2.8	Efficiently reversing a list	83
4.3	Reasoning about List Operations	84
4.3.1	Distributing length over append	84
4.3.2	Associativity of list append	86
4.3.3	Length of filtered lists, and the <code>with</code> construct	86
4.3.4	Filter is idempotent, and the <code>keep</code> idiom	90
4.3.5	Reverse preserves list length	93
4.4	Conclusion	95
4.5	Exercises	96
5	Internal Verification	99
5.1	Vectors	99
5.1.1	The Vector Datatype	100
5.1.2	Appending vectors	101
5.1.3	Head and tail operations on vectors	102
5.1.4	Using types to express properties of other vector operations	103
5.2	Braun Trees	106

5.2.1	The <code>braun-tree</code> datatype, and sum types	106
5.2.2	Inserting into a Braun tree	107
5.2.3	Removing the minimum element from a Braun tree	110
5.3	Sigma Types	114
5.3.1	Why Sigma and Pi?	115
5.4	Binary Search Trees	116
5.4.1	The <code>bst</code> module, and <code>bool-relations</code>	116
5.4.2	The <code>bst</code> datatype	120
5.4.3	Searching for an element in a binary search tree	120
5.4.4	Inserting an element into a binary search tree	122
5.5	Discussion: Internal vs. External Verification	123
5.6	Conclusion	125
5.7	Exercises	126
6	Generating Agda Parsers with <code>gratr</code>	129
6.1	A Primer on Grammars	130
6.1.1	Derivations	131
6.1.2	From derivations to syntax trees	132
6.1.3	Regular-expression operators for grammars	136
6.2	Generating Parsers with <code>gratr</code>	136
6.2.1	Compiling the emitted parsers	138
6.2.2	Running the emitted executable	139
6.2.3	Modifying the emitted code to process parse trees	140
6.2.4	Reorganizing rules to resolve ambiguity	141
6.3	Conclusion	144
6.4	Exercises	144
7	Type-level Computation	147
7.1	Integers	147
7.1.1	The \mathbb{Z} datatype	148
7.1.2	Addition on integers	149
7.1.3	Antisymmetric integer comparison	150
7.2	Formatted Printing	151
7.2.1	An unsuccessful attempt	152
7.2.2	A working solution	154
7.3	Proof by Reflection	157
7.3.1	The datatype of representations, and its semantics	158
7.3.2	The list simplifier	159
7.3.3	Proving that the simplifier preserves the semantics	161
7.4	Conclusion	163
7.5	Exercises	164
8	A Case Study: Huffman Encoding and Decoding	165
8.1	The Files	166
8.2	The Input Formats	167
8.3	Encoding Textual Input	167

8.3.1	Computing word frequencies using tries	169
8.3.2	Initializing a priority queue with Huffman leaves	171
8.3.3	Processing the priority queue to compute a Huffman tree	174
8.3.4	Computing the code from the Huffman tree	175
8.3.5	Encoding the input words using the computed code	175
8.4	Decoding Encoded Text	177
8.4.1	Code trees	177
8.4.2	Computing the code tree	177
8.4.3	Decoding the input	179
8.5	Conclusion	180
8.6	Exercises	180
9	Formalizing Deductive Systems	181
9.1	Termination Proofs	181
9.1.1	Defining termination in Agda	182
9.1.2	Termination of greater-than on natural-numbers	184
9.1.3	Proving termination using a measure function	185
9.1.4	Nontermination with negative datatypes	186
9.2	Operational Semantics for SK Combinators	188
9.2.1	Mathematical syntax and semantics for combinators	188
9.2.2	Defining the syntax and semantics in Agda	190
9.2.3	Termination for S-free combinators	192
9.2.4	Defining lambda abstractions using combinators	196
9.3	Conclusion	202
9.4	Exercises	204
10	Intuitionistic Logic and Kripke Semantics	205
10.1	Positive Propositional Intuitionistic Logic (PPIL)	206
10.2	Kripke structures	209
10.3	Kripke semantics for PPIL	214
10.4	Soundness of PPIL	216
10.4.1	Monotonicity proof	217
10.4.2	Soundness proof	219
10.5	Completeness	221
10.5.1	A universal structure	221
10.5.2	Completeness and soundness with respect to \mathcal{U}	223
10.5.3	Concluding completeness and universality	227
10.5.4	Composing soundness and completeness	228
10.6	Conclusion	234
10.7	Exercises	234
	Quick Guide to Symbols	237
	Commonly Used Keyboard Commands	239
	Some Extra Emacs Definitions	241

References**243****Index****245**

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 US, 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 “testing shows the presence, not the absence of bugs.” 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 Languages 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. 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 `Control-c Control-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 these:

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

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):

1008	1009	100A	100B
 10080	 10090	 100A0	 100B0
 10081	 10091	 100A1	 100B1
 10082	 10092	 100A2	 100B2
 10083	 10093	 100A3	 100B3
 10084	 10094	 100A4	 100B4

Ok, 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. 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 \forall (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. And 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 which 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 termination 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.

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 ($\{0, 1, 2, \dots\}$) 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:

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

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 $\forall L \rightarrow \text{rev} (\text{rev } L) \equiv L$ 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 above, 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 $\text{rev} (\text{rev } L)$ 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 – which both rest on Agda’s powerful type system – below.

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. Sure, it sounds at least somewhat intellectually interesting, and in some far-off future, maybe companies will be hiring programmers to write verified software for pacemakers, or power substations controllers, or electronic control units (ECUs) in cars – all systems that have been shown to be vulnerable to computer attacks in the past few years. But it seems a bit impractical for right now. After all, if you search the web for something like “Agda programmer jobs” you will find nothing.

My answer to this is that while admittedly, verified functional programming is restricted to academia and perhaps a few small advanced-development companies, 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 which 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 basic ideas from functional programming appear in many other very mainstream languages. For example, JavaScript and Python both have anonymous functions, which (as mentioned above) 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.

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 US, 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 above, 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 development below has been confirmed to work with the latest version of Agda (2.4.2.2 at the time of this writing).

The Iowa Agda Library

The chapters of the 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 the current version of the IAL via subversion (or browse it with a web browser) here:

<https://svn.divms.uiowa.edu/repos/clc/projects/agda/ial>

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 below 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. 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 which 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.

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 programming 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.

Acknowledgments

I am grateful to the undergraduate students of CS:3820 (22c:111) at The University of Iowa for feedback which 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 6, 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! I am grateful to my wife for her support as I finished this book up Summer 2015. I also wish to acknowledge the support of the Catholic community of St. Wenceslaus, Iowa City. AMDG.

Chapter 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 Introduction 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:

```
module bool where

open import level

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

data  $\mathbb{B}$  : Set where
  tt :  $\mathbb{B}$ 
  ff :  $\mathbb{B}$ 
```

The main thing we want to consider here is the declaration of the boolean datatype \mathbb{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 5.2).

- **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 below 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  
  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 introduction, 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. The section "Some Extra Emacs Definitions" in the appendix 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 [15]. To avoid this, Agda uses a trick: an infinite hierarchy of type levels. `Set` really stands for `Set 0`, and for every natural number $n \in \{0, 1, 2, \dots\}$, 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 `ℕ`, all we really need to know is that `data ℕ : Set` is declaring a datatype `ℕ`, 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 `ℕ`. 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 `ℕ`, you will see some these comments:

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

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

Comments written with the delimiters “`{-#`” and “ `#-}`” are directives to the Agda compiler. In this case, they are telling Agda that `ℕ` and its constructors are the definition to use for a builtin notion of booleans that the compiler is supporting. There is also a directive with `COMPILED_DATA`, telling Agda to compile the `ℕ` 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 Introduction, 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 Control-c Control-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 Control-c Control-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 `ℕ` in response.
- **Evaluating an expression.** To see what value an expression computes to (or *normalizes* to, in the terminology of type theory), type Control-c Control-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 `ℕ` datatype of booleans, you will see declarations like this (and a few more):

```
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 above. 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 above 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 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` using Control-c Control-d (as mentioned in Section 1.2, or you can also ask it for the type of the negation operator `~_` by typing Control-c Control-d and then literally `~_`. You will see that the type is

```
 $\mathbb{B} \rightarrow \mathbb{B}$ 
```

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:

```
~_ :  $\mathbb{B} \rightarrow \mathbb{B}$ 
~ 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

```
~_ :  $\mathbb{B} \rightarrow \mathbb{B}$ 
```

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 $\mathbb{B} \rightarrow \mathbb{B}$ is the type for functions from \mathbb{B} to \mathbb{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 matching a set of function calls that could be made with the operator 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 Control-c Control-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 Control-c Control-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
```

```

_&&_ :  $\mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B}$ 
tt && b = b
ff && b = ff

```

Figure 1.1: The definition of boolean conjunction

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

```
 $\mathbb{B} \rightarrow (\mathbb{B} \rightarrow \mathbb{B})$ 
```

This type says that `_&&_` takes in a boolean (the first conjunct) and returns a function of type $\mathbb{B} \rightarrow \mathbb{B}$. 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 above (Section 1.3) that this operator can be given two arguments in infix notation, like this

```
tt && ff
```

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 $\mathbb{B} \rightarrow (\mathbb{B} \rightarrow \mathbb{B})$, we can also just give the conjunction operation a single argument:

```
_&&_ tt
```

If you ask Agda what type this has by typing Control-c Control-d and then `_&&_ tt`, you will see that it has type $\mathbb{B} \rightarrow \mathbb{B}$. This makes sense, given that conjunction has type $\mathbb{B} \rightarrow (\mathbb{B} \rightarrow \mathbb{B})$. 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 above) 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
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 `Control-c Control-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 above (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
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 recursive 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 above 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 Control-c Control-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 typable.

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 \forall , 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 towards the end of Section 1.1 above. 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:

$$\mathbb{B} \rightarrow A \rightarrow A \rightarrow A$$

This is expressing the basic idea that if-then-else takes in three inputs, which are a boolean (\mathbb{B} , for the guard) and then an A value (for the then-part) and another A value (for the else-part); and return an A values. 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 Control-c Control-d) the expression

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

it will figure out that the type variable A must be \mathbb{B} , since the then- and else-parts are both booleans. And since \mathbb{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 \mathbb{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 $\mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B}$. 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 ($\mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B}$), 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  $\mathbb{B}$  else ( $\mathbb{B} \rightarrow \mathbb{B}$ )
```

This is certainly not something you can write in a mainstream programming language. This expression evaluates to the type \mathbb{B} if its guard is true (which it is) and to the type $\mathbb{B} \rightarrow \mathbb{B}$ otherwise. We know that \mathbb{B} has type `Set 0` (which is just an abbreviation for `see Set 0`, as mentioned in Section 1.1). It turns out that $\mathbb{B} \rightarrow \mathbb{B}$

has type `Set ℓ`, also. In fact, in Agda, if A and B both have type `Set ℓ`, then so does $A \rightarrow B$. 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 `Control-c Control-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 the “Quick Guide to Symbols” at the end of the book).
- 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 `Control-c Control-d`, then the expression, and then return.
 - Normalize (evaluate) an expression by typing `Control-c Control-n`, then the expression, and then return.
- 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.

1.8 Exercises

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`

2. What is the type of each of the following expressions (you can check these in Agda with Control-c Control-d)?

(a) `tt`

(b) `if tt then ff else tt`

(c) `_&&_`

(d) `ℬ`

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 )
```

4. Define a datatype `day` which is similar to the `ℬ` datatype, but has one constructor for each day of the week.
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`).
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.
7. Define a function `is-red` which takes in a `suit` as defined in the previous problem, and returns `tt` iff the suit is a red one (hearts and diamonds).

Chapter 2

Theorem Proving with the Booleans

In the previous chapter, we saw the definition of the booleans, and how to define functions on them using pattern matching and equations. It is pretty interesting to be able to declare our own mixfix operators like if-then-else. And writing code with equations is rather elegant; for example, the definition of negation looks just like we might expect from a mathematical definition:

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

And it is somewhat intriguing that we can form if-then-else expressions with then- and else-parts of any type, including then- and else-parts that are types themselves. That is rather exotic, and not something we find in mainstream programming languages. But you could be forgiven for not being overawed by the features we have considered so far.

I hope this chapter will start to change that a little, as we consider how to state and prove theorems about the boolean operations we defined in the previous chapter. The Agda code in this Chapter can be found in `bool-thms.agda` in the IAL (there is also a `bool-thms2.agda` with yet more theorems about boolean operations). As described in Section 1.2, you can load this file by opening it within `emacs` and then typing `Control-c Control-l`. I will assume below that you have already done this.

2.1 A First Theorem about the Booleans

There are many properties that we might be interested in proving about our boolean operations. For example, in Section 1.4 we saw how to evaluate the expression `~ ~ tt` in Agda, using `Control-c Control-n`. Of course, the answer we get back is `tt`. If our definition of `~_` were incorrect, this property – that double negation on booleans can be eliminated – might not hold. For example, suppose we had this definition of negation instead:

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

Admittedly, we'd have to be having a pretty bad coding day to define $\sim \text{ff}$ to be ff , but it could happen. And certainly if we were writing some more complicated code, we know very well that we could make mistakes and accidentally introduce bugs into the code without meaning to. If we wrote the above definition, evaluating $\sim \sim \text{tt}$ would give us ff , instead of the expected tt . So it would be nice to record and check the fact that $\sim \sim \text{tt}$ is supposed to be tt . We could write some test code and remember to run it from time to time to check that we get the expected answer. But we can use this test case as a very rudimentary form of theorem, and use Agda's theorem-proving capabilities to confirm that the theorem holds (i.e., the test succeeds).

2.1.1 Types as formulas: the Curry-Howard isomorphism

First, we have to express the theorem of interest, namely that $\sim \sim \text{tt}$ equals tt . In Agda, this is done with an **equality type**:

```
 $\sim \sim \text{tt} \equiv \text{tt}$ 
```

An equality like this is called a **propositional** equality, to distinguish it from another form of equality we will consider shortly. To type that triple-equals symbol, you use `\==`, and then you may need to hit space to select that symbol from a few choices that can be typed with a key sequence starting with `"\=="`. If you ask Agda what type this has (using Control-c Control-d as described in Section 1.2), Agda will respond with `Set`. So Agda views a formula like this equation as a type (recall from Section 1.1 that `Set` is the type for types). The idea that formulas can be viewed as types is a central idea in type theory, known as the **Curry-Howard isomorphism**, after Haskell Curry and William Howard, the two people primarily credited with discovering it. It is a powerful idea, because when developed in full as in a language like Agda, it allows us to use one and the same language for writing types and writing formulas. We will see shortly that we can also use the same language for writing programs (which have types) and proofs (which prove formulas). This cuts down on the new languages we need to learn to do verified programming. Instead of needing to know

- a language for programs
- a language for their types
- a language for formulas
- a language for proofs

we just need to grasp a fairly straightforward language for programs, along with an admittedly much richer language for types. But the Curry-Howard isomorphism has positive benefits for both programming and theorem-proving. By giving us a way to write formulas just as certain special types, the Curry-Howard isomorphism has introduced new types into our mental universe, which we can use to write typed programs that are not possible with the usual types we are familiar

with from mainstream programming languages. We will see several interesting examples of this, in Chapters 5 and 7 below.

2.1.2 Proofs as programs: more Curry-Howard

Let us see now how to write a proof of our example formula, as a program. This example is in `bool-test.agda`, in the Iowa Agda Library. We will give our proof a name: `~~tt`. It is not easy coming up with good names for theorems. Naturally we want to pick names which give some indication of what the theorem in question is, without being too verbose. Here, since our theorem is about the value of `~~tt`, I am using `~~tt` as the name. Thanks to Agda's requirement of spaces around operators (see Section 1.4.1), this is a legal symbol we can use to name anything we like (here, a proof) within Agda.

In Agda, writing a proof is just programming, so we have to approach this just the way we did writing boolean operations in Chapter 1. We have to declare the type of our function, which is easy:

```
~~tt : ~ ~ tt ≡ tt
```

and now we have to write a set of equations – in this case we will need just one – which give the definition of `~~tt`.

Ok, so finally: how do we prove that `~ ~ tt` equals `tt`? Well, how would we prove it if we were just reasoning on paper? We would likely just say that `~ ~ tt` evaluates to `tt` by running the program expression `~ ~ tt`. If pressed, we could give some more detail, by showing how individual calls to the `~_` operation evaluate:

$$\sim \sim tt = \sim ff = tt$$

Fortunately, Agda does not need us to do this. Agda adopts the simple but powerful idea of treating any program expressions that appear inside types as equal if their defining equations make them equal. So when Agda's type checker is asked to consider the type

```
~ ~ tt ≡ tt
```

it takes into account the defining equation for the negation operator `~_`. So it treats this type as equivalent to just

```
tt ≡ tt
```

Now this is a formula you would hope is easy to prove – and it is! The proof is just `refl`, which stands for **reflexivity**. We can recall from discrete math that this is a general property one can ascribe to different mathematical relations (like equality, in this case), which says that every element is related to itself.

Finally we have enough information to write our proof (which you can find in `bool-test.agda`):

```

~~tt : ~ ~ tt ≡ tt
~~tt = refl

```

Let us review what is happening with this piece of Agda code. We are writing a definition for the symbol `~~tt`. As usual with definitions in Agda, and as we saw for the definitions of the boolean operations in Chapter 1, we first have to write the type for `~~tt`, and then give equations defining `~~tt`. The type is `~ ~ tt ≡ tt`. Agda’s type checker treats program expressions that appear in types as equal if their defining equations, which explain how to evaluate calls to the functions occurring in those expressions, make them equal. Since `~ ~ tt` evaluates to `tt` (as we can easily confirm with Control-c Control-n), Agda treats the formula we are trying to prove as equivalent to the trivial formula `tt ≡ tt`. The two formulas are said to be **definitionally equal**. The formula `tt ≡ tt` can be proved with `refl`, and so we complete our definition of `~~tt` by saying that `~~tt` is defined to be `refl`. This `refl` is actually the sole constructor of the equality type, as you can see in `eq.agda` if you are interested.

We have proved our first theorem in Agda. Time to celebrate!

2.1.3 Going deeper: Curry-Howard and constructivity

The Curry-Howard isomorphism was originally developed for **constructive logic**, and this is still the setting in which it is most used and best known. Constructive logic is a restriction of standard, or as it is called, **classical** logic. Have you ever asked a friend if they want to go get pizza or burgers and they say “yes”? This humorous answer is (likely unwittingly) playing on the difference between constructive and nonconstructive logic. In constructive logic, if we prove that A or B is true, we always know exactly which one of those facts it is that is true. You expect to be in this situation when you ask about pizza (A) or burgers (B). Similarly, if we prove that there exists an object x satisfying some property, we know what the identity of that object x is. To convince me that this x exists, you need to “show me” (as they say in Missouri). You might think that all proofs of *disjunctions* (“or” statements) and existential statements are like that, and hence that all reasoning must be constructive. Here is a well-known clever proof, first brought to the attention of logicians by Roger Hindley [9], that shows this is not the case.

Theorem 2.1.1. *There exist irrational numbers a and b such that a^b is rational.*

Proof. We have to remember that a rational number is one which can be described as a fraction $\frac{p}{q}$, where p and q are integers, with $q \neq 0$; and an irrational number is one that lacks such a description. The ancient Greeks knew that $\sqrt{2}$ is irrational, and we will use that fact now in the proof. Either $\sqrt{2}^{\sqrt{2}}$ is rational or else it is not. Let us suppose it is rational. In that case, our proof is complete, as we can take a

and b both to be $\sqrt{2}$. So let us suppose that $\sqrt{2}^{\sqrt{2}}$ is irrational. We will take a to be this number, and b to be just $\sqrt{2}$. Let us confirm that a^b is rational (in fact, an integer):

$$\begin{aligned}
 a^b &= (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} && \text{definition of } a \text{ and } b \\
 &= \sqrt{2}^{(\sqrt{2}\sqrt{2})} && \text{basic property of exponentiation} \\
 &= \sqrt{2}^2 && \text{definition of } \sqrt{2} \\
 &= 2 && \text{definition of } \sqrt{2}
 \end{aligned}$$

□

At the end of this proof, we know that a and b are either $\sqrt{2}$ and $\sqrt{2}$ or else $\sqrt{2}^{\sqrt{2}}$ and $\sqrt{2}$. But we do not know which it is. Now it happens that other results in the theory of irrational numbers can confirm that indeed the second possibility is the correct one, and the first possibility is wrong. But the proof we have just seen – which is a perfectly correct *classical* proof of the theorem – does not tell us. We are left in the dark about the exact values of a and b , even though we have established they exist. This is a good example of a nonconstructive proof.

The essential nonconstructive reasoning here is the use of what is called the **law of excluded middle**: for every formula F , $F \vee \neg F$ (either F is true or else F is false). It is a rather philosophical question whether or not this statement is, in fact, true of the real world (at least as we conceptualize it). From a computational viewpoint, under the Curry-Howard isomorphism, the law of excluded middle says that for any fact F , we can compute whether or not F is true. And it is well known from the theory of computation that some properties simply cannot be computationally tested in general. The paradigmatic example is the **Halting Problem**: does a particular program terminate in a finite number of steps, or does it run forever? It is not hard to prove that there is no computer program which can decide the Halting Problem. So constructively, we cannot prove the following formula, expressing that every program P either halts or does not halt:

$$\forall P. (\text{Halts}(P) \vee \neg \text{Halts}(P))$$

Proving this constructively would amount to writing a program that can solve the Halting Problem, which, as just mentioned, is impossible.

The impact of the example proof above is somewhat lessened by the fact that we have a simple constructive proof of the same theorem (see [9] for sources):

Theorem 2.1.2 (Constructive version). *There exist irrational numbers a and b such that a^b is rational.*

Proof. Take a to be $\sqrt{2}$ and b to be $\log_2 9$. We must see that a^b is rational, and also

that $\log_2 9$ is irrational. For the former:

$$\begin{aligned}
 a^b &= \sqrt{2}^{\log_2 9} && \text{definition of } a \text{ and } b \\
 &= \sqrt{2}^{2 \log_2 3} && \text{basic property of logarithms} \\
 &= (\sqrt{2}^2)^{\log_2 3} && \text{basic property of exponentiation} \\
 &= 2^{\log_2 3} && \text{definition of } \sqrt{2} \\
 &= 3 && \text{definition of } \log_2 3
 \end{aligned}$$

To see that $\log_2 9$ is irrational, let us assume that it is rational, and then derive a contradiction. If $\log_2 9$ is rational, then it must equal $\frac{p}{q}$ for some integer p and nonzero integer q . So we have:

$$\log_2 9 = \frac{p}{q}$$

Multiplying by q gives:

$$q \log_2 9 = p$$

A basic property of logarithms gives:

$$\log_2 9^q = p$$

The definition of logarithm now gives:

$$9^q = 2^p$$

This is impossible, since if it were true, 2 would have to divide 9^q evenly, and it does not. \square

Here is another example, whose nonconstructive proof is perhaps not quite as strikingly ingenious as the one above, but for which no constructive proof is currently known. We must recall that an algebraic number is one which is the root of a nonzero polynomial with rational coefficients, and a transcendental number is one which is not algebraic. The numbers π and e are transcendental, though this was only proved in the late 19th century.

Theorem 2.1.3. *Either the sum $\pi + e$ or the product πe (or both) is irrational.*

Proof. Consider the polynomial $(x - \pi)(x - e) = x^2 - (\pi + e)x + \pi e$. The roots of this polynomial are π and e . A number is algebraic, as we just recalled, if it is the root of a nonzero polynomial with rational coefficients. So this polynomial must fail to have rational coefficients, otherwise by the definition of an algebraic number, π and e would both be algebraic; and we know they are both not algebraic. Hence, one of the coefficients of the polynomial must fail to be a rational number. So either $\pi + e$ or πe , or possibly both, are irrational. \square

Notice that the proof does not show that $\pi + e$ is irrational, nor does it show that πe is irrational. Indeed, it is not currently known which is true (or if both are true).

The proof just shows that one of those numbers must be irrational. This is another example of nonconstructivity: we have a proof of a disjunction (“or” statement), but not of either disjunct (the statements being combined with “or”).

Finally, let us consider the Curry-Howard isomorphism and classical logic. It does not appear possible to use the Curry-Howard isomorphism with nonconstructive reasoning. For it seems that a program that proves $\forall F. F \vee \neg F$ would have to be a universal oracle, which given any formula F could compute whether F is true or else report that F is false. So if one wanted to know if the Goldbach Conjecture (which states that every integer greater than two is the sum of two prime numbers) is true or false, one could just ask this oracle.

Quite surprisingly, it turns out that if we expand the kinds of programs we are allowed to write, the Curry-Howard isomorphism can be made to work for classical logic. This amazing insight is due to Timothy Griffin [7], and is still an active subject of research. To get a hint for how this could make sense, here is how a proof of $\forall F. F \vee \neg F$ can be viewed computationally. The proof – let us call it p – takes in a formula F , and is required to tell you whether either F is true, or else $\neg F$ is true. Now, $\neg F$ is very often defined in logic just to be $F \rightarrow \text{False}$. That is, we say $\neg F$ is true iff the truth of F would imply a contradiction. So here is what p does. It tells you that yes, $\neg F$ is true, by returning a “proof” q (we will see how q works in a moment). The proof p always returns this proof q saying that $\neg F$ is true, regardless of what F is. How can we make use of this (dubious) information that p has given us? Well, the only thing you can do with a proof of $A \rightarrow B$, in general, is call it with an argument of type A . Here, we have a proof of $F \rightarrow \text{False}$. So, imagine that at some point, we want to use this proof. That means we are going to call it with an argument of type F . Let us call this argument r . As soon as q sees r , it, so to speak, travels back in time to the moment when p was being asked to return either a proof of F or a proof of $\neg F$. And it changes the past, so that now, p returns r showing that F is true. So the proof q which p returns initially is a sort of placeholder. It just says that sure, $\neg F$ is true, but if anyone ever tries to use that information by giving a proof r of F , then it jumps back and tells p to return this proof r instead of q .

This backtracking sounds a bit like exceptions in programming languages, and indeed, the computational interpretation of such proofs is as **control operators**, which are like more powerful forms of exceptions. With this approach, a proof of $\forall F. F \vee \neg F$ is not a universal oracle that can tell you whether any formula F is true or false. For you can never take irrevocable action based on the truth or falsity of F . You must always be prepared to backtrack, if it turns out that the answer $\neg F$ which the proof gave you is not definitive. So you could never send a network message or initiate some control sequence based on the answer you get from this proof of $\forall F. F \vee \neg F$. And that is where the Curry-Howard isomorphism for classical logic is less useful for programming than the Curry-Howard isomorphism for constructive logic.

2.2 Universal Theorems

Despite our careful, not to say painstaking, consideration of the proof in Section 2.1.2 above, I hope you will agree that this theorem is easy, and surely theorem proving cannot be so trivial in general. Yes, mathematicians are in business for a reason: theorem proving can be extremely difficult, with a single conjecture stumping the brightest minds humankind has to offer for centuries. Most proofs about everyday programs, however, are not so difficult as that. They lie somewhere in between the totally trivial proof we just considered, and more complicated developments. Let us take some further steps into the world of theorem proving by considering universal theorems.

We proved above that $\sim \sim tt$ equals tt . And we could just as well prove, in essentially the same way, that $\sim \sim ff$ equals ff (see `bool-test.agda`):

```
 $\sim \sim ff : \sim \sim ff \equiv ff$ 
 $\sim \sim ff = refl$ 
```

Now it is a natural step of human reasoning to generalize from the concrete to the abstract. We notice that $\sim \sim tt$ equals tt and that $\sim \sim ff$ equals ff , and part of our basic intellectual equipment as humans cries out for us to abstract from the concrete data tt and ff , to observe that for any boolean b , we have $\sim \sim b \equiv b$. In mathematical terms, we would like to write a universally quantified formula, to express this idea for any boolean b :

$$\forall (b : \mathbb{B}) \rightarrow \sim \sim b \equiv b$$

The upside-down capital “A” is called the **universal quantifier**. Here, it allows us to make a statement about any b of type \mathbb{B} .

But we have seen this \forall notation before, in the type of if-then-else (Section 1.6). It is the operator for type polymorphism. Well, in Agda, because types can contain program expressions, this operator can also be used to express program polymorphism. Just as the polymorphic type for if-then-else can be instantiated with any type, for the then- and else-parts, that we wish; similarly, some types can be instantiated with any program that we wish. We can view \forall as the operator for universal quantification or for polymorphism. In Agda, these concepts are unified, by the Curry-Howard isomorphism.

2.2.1 Proving the theorem using pattern matching

So let us see how to prove our universal theorem, which you can find a short way into the file `bool-thms.agda` in our library. We should begin just as we have for other definitions in Agda, by choosing a symbol and then writing the type we want that symbol to have:

$$\sim\sim\text{-elim} : \forall (b : \mathbb{B}) \rightarrow \sim \sim b \equiv b$$

This says that we are starting a definition of the $\sim\sim\text{-elim}$ symbol, which will have the given universal type. Now we have to write equations defining $\sim\sim\text{-elim}$. Here we need to combine ideas we saw from our definitions of boolean operations like negation and conjunction, with the Curry-Howard isomorphism. This theorem that we are trying to prove can be shown by case analysis on the variable b . That is, since we know that $\sim \sim \text{tt}$ equals tt and $\sim \sim \text{ff}$ equals ff , to prove that $\sim \sim b \equiv b$, we just have to consider explicitly the two possibilities for b . This is done in Agda by pattern matching, just as when we were defining functions. In each case, the proof is by `refl`. So we get this final definition proving our theorem:

```

 $\sim\sim\text{-elim} : \forall (b : \mathbb{B}) \rightarrow \sim \sim b \equiv b$ 
 $\sim\sim\text{-elim tt} = \text{refl}$ 
 $\sim\sim\text{-elim ff} = \text{refl}$ 

```

We have written two equations defining the behavior of $\sim\sim\text{-elim}$ for the two possibilities for boolean variable b . So we are actually defining $\sim\sim\text{-elim}$ as if it were a function taking in one argument, namely the boolean value for b . So the universal type $\forall (b : \mathbb{B}) \rightarrow \sim \sim b \equiv b$ can be seen as a function type: given input b , the $\sim\sim\text{-elim}$ function will return an output of type $\sim \sim b \equiv b$. This is also part of the Curry-Howard isomorphism.

We should note something important about how Agda type-checks a definition like this. Our two equations are for the two cases for boolean variable b . Each equation deals with one instantiation of this variable. Once the left-hand side of the equation has specified an instantiation for a variable, then the rest of the equation – including any additional parts of the left-hand side, and then the whole right-hand side – are type-checked under that instantiation. So in the first equation, after it sees `tt` as the instantiation for b , Agda is checking that the right hand side has type $\sim \sim b \equiv b$, under the instantiation of b to `tt`. In other words, Agda is checking that the right-hand side has type $\sim \sim \text{tt} \equiv \text{tt}$, since we have instantiated b to `tt`. This process of proceeding with type-checking using an instantiated version of the type we started with is sometimes called **type refinement**. We are refining the type $\sim \sim b \equiv b$ which we have for the right-hand sides of our equations, based on instantiations encountered on their left-hand sides.

2.2.2 An alternative proof and implicit arguments

The great thing about a proof in Agda is that if the type checker accepts it, then the formula in question is valid (assuming no bugs in Agda, no cosmic rays flipping bits of the computer's memory system while running Agda, etc.). We do not need to know how a proof works, if all we care to know is that the formula it proves is valid. But still, you may be curious: can we prove our $\sim\sim\text{-elim}$ theorem another way? Could we use our $\sim\sim\text{tt}$ and $\sim\sim\text{ff}$ theorems to prove the two cases? We

certainly can do that. Agda also accepts the following as a proof of our example theorem (see `bool-test.agda`):

```
~~-elim2 : ∀ (b : ℬ) → ~ ~ b ≡ b
~~-elim2 tt = ~~tt
~~-elim2 ff = ~~ff
```

This makes sense, since `~~tt` and `~~ff` have exactly the types we need for their respective cases. Is this really a different proof? Not really, because `~~tt` and `~~ff` are both defined to equal `refl`, so this proof is equal to the first proof we wrote.

While we are thinking about proofs being equal, we know that `~~tt` equals `refl` and `~~ff` equals `refl`. So shouldn't that imply that `~~tt` equals `~~ff`? If that is true, then even though it seems a bit wrong, wouldn't the following proof type-check in Agda?

```
~~-elim3 : ∀ (b : ℬ) → ~ ~ b ≡ b
~~-elim3 tt = ~~ff
~~-elim3 ff = ~~tt
```

If you try this out, you will see that the answer is no. The reason is more subtle than you might think, and it has to do with a very handy feature of Agda called **implicit arguments**. Implicit arguments to a function are arguments that you do not have to write when you call the function. Agda will try to infer them. We saw these already, briefly, in discussing the if-then-else operator in Chapter 1:

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

The curly braces around `ℓ` and `A` in the type tell Agda to try to infer the values for those inputs. Agda uses an inference algorithm, which unfortunately – and for deep mathematical reasons – cannot always succeed, to try to figure out what the instantiations for `ℓ` and `A` have to be, based on the other arguments to if-then-else, or on how the if-then-else expression is being used in a bigger surrounding context.

The `refl` function we have been using also has implicit arguments. One of these implicit arguments is the program expression `x` for which we are proving `x ≡ x`. In the case of `~~tt` this implicit argument is `tt`, while in the case of `~~ff` it is `ff`. So `~~tt` does not equal `~~ff`, contrary to what we were thinking might be the case above. This is because `~~tt` equals `refl` but with `tt` for the implicit argument for `x` (the `x` as in `x ≡ x`), while `~~ff` equals `refl` but with `ff` for that implicit argument. Implicit arguments are one of the great features of Agda, and we will see more examples of their use shortly.

2.2.3 Using holes in proofs

When trying to prove something complicated, it is very helpful to be able to put some holes in your proof, to be filled in later. In fact, you can even just put a hole for your entire proof! Of course, then you have not proved anything, but there is still some benefit: you can make sure that the formula you are trying to prove can be type-checked by Agda. The syntax for a hole in Agda is a question mark ("?"). So we could write this proof with holes, for the example we have been considering:

```
~~-elim : ∀ (b : ℤ) → ~ ~ b ≡ b
~~-elim = ?
```

When you ask Agda to check the file (by typing Control-c Control-l), this hole will turn into a green region in between curly braces, followed by a distinct number for each hole. We will see some other commands later, for getting information about holes. Holes can go anywhere on the right-hand side of an equation. So we could also take a first step on the proof, and then put some more holes, like this:

```
~~-elim : ∀ (b : ℤ) → ~ ~ b ≡ b
~~-elim tt = ?
~~-elim ff = ?
```

Loading the file with Control-c Control-l will change this text so it looks like

```
~~-elim : ∀ (b : ℤ) → ~ ~ b ≡ b
~~-elim tt = { }0
~~-elim ff = { }1
```

where { }0 and { }1 are highlighted in green. These are the holes. To remove a hole, you cannot use backspace or delete in *emacs*. Instead, you have to cut the hole out. A simple way to do this is to position your cursor at the start of the hole (at the "{" symbol), and type Control-k. This is a standard *emacs* command to cut all the text to the end of the line. You can paste it back in with Control-y. For other basic *emacs* commands, see Figure 10.27 at the end of the book.

2.3 Another Example, and More On Implicit Arguments

Let us prove an obvious theorem about conjunction. You can find this coming next in `bool-thms.agda`, but we are going to build up to the exact form of proof that is used there. To start with, the theorem we will prove is that for any boolean *b*, we have that *b* && *b* equals *b*. In English: conjoining *b* with itself gives you the same thing as just *b*. First, let us make sure we know how to state this theorem in Agda. It is a universal theorem, so we should use our universal quantifier \forall with a variable *b* of type \mathbb{B} :

```
∀ (b : ℤ) → b && b ≡ b
```

We can prove this theorem very similarly to the previous example, because each instance, for each concrete boolean value for b , follows by just running the code for conjunction. So the proof looks like this:

```
&&-same : ∀ (b : ℬ) → b && b ≡ b
&&-same tt = refl
&&-same ff = refl
```

Now, one thing about Agda, and functional programming in general, is that a great emphasis is placed on being as concise as possible. This is taken almost to the point of fanaticism, and certainly less code is not always more useful for programmers trying to come back and understand how something they or someone else wrote actually works. Still, the ability to omit unnecessary parts of code is powerful, and can help craft very elegant and more easily understandable programs – if used wisely. In the case of the theorems we have considered so far, there is not a lot of fat to trim, except that Agda’s inference mechanism is clever enough to deduce the type of the variable b from the fact that b is used as an argument to a boolean operation. So we can write this alternative and minutely shorter proof for the theorem:

```
&&-same : ∀ b → b && b ≡ b
&&-same tt = refl
&&-same ff = refl
```

The equations for the proof are unchanged, but we have shortened the type from

$$\forall (b : \mathbb{B}) \rightarrow b \ \&\& \ b \equiv b$$

to

$$\forall b \rightarrow b \ \&\& \ b \equiv b$$

This is somewhat nice: there was no need to tell Agda that b has type \mathbb{B} , because Agda can figure this out from the way b is used later in the type.

One other possible improvement we can make to the theorem is to make this boolean b an implicit argument. To do this, as we saw for if-then-else, we just have to put curly braces around the variable in the type:

$$\forall \{b\} \rightarrow b \ \&\& \ b \equiv b$$

One new issue that arises, though, is that unlike the case for the variables ℓ and A in the code for if-then-else, here we must pattern-match on the input argument given for variable b , to consider the cases where b is tt and where it is ff . Agda has a very sensible notation for pattern-matching on implicit arguments. The arguments have to be given in curly braces in the patterns on the left-hand sides of the defining equations. So the final code we have for `&&-same` is:

```
&&-same : ∀ {b} → b && b ≡ b
&&-same {tt} = refl
&&-same {ff} = refl
```

Making this argument implicit is rather neat, because sometimes when we use the theorem later on, we will not have to supply an argument for b , just the way we do not have to supply arguments for ℓ and A when we call `if-then-else`. For example, `bool-thms.agda` includes the following, just to demonstrate this point:

```
test-∧∧-same : tt ∧∧ tt ≡ tt
test-∧∧-same = ∧∧-same
```

Notice that we do not have to give any implicit argument to `∧∧-same` where we are using it in the definition of `test-∧∧-same`. Agda can infer that the implicit b must be `tt`, since we are using `∧∧-same` here to prove `tt ∧∧ tt ≡ tt`.

Using implicit arguments can lead to much more concise code in Agda, but it can also cause confusion, because if Agda cannot infer an implicit argument, you may get a somewhat unpleasant error message including variables whose names start with underscores (these are meta-variables Agda has introduced, for expressions it needs to deduce). If this happens, you can supply implicit arguments explicitly using curly braces. For example, we could just as well write

```
test-∧∧-same : tt ∧∧ tt ≡ tt
test-∧∧-same = ∧∧-same{tt}
```

One small note about this: you must supply the implicit arguments in the order they are declared in the type for the function. So if we wanted to supply an argument for the type variable A of the `if-then-else` operator, we would have to supply an argument for ℓ first. This can be a bit of a nuisance (for example, we have not even seen how to construct type levels, though you can find the constructors in the file `level.agda`). To get around this problem, Agda lets you name the variable you wish to instantiate, as part of the curly-braces notation. So for `if_then_else_`, we know we have this type:

$$\forall \{ \ell \} \{ A : \text{Set } \ell \} \rightarrow \mathbb{B} \rightarrow A \rightarrow A \rightarrow A$$

To instantiate A to \mathbb{B} , without explicitly instantiating ℓ , we can write

```
if_then_else_{A = \mathbb{B}}
```

Then we will see the type $\mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B}$, as expected. This feature of Agda is a little dangerous, because we have to name the variable (A , in this case) where we are doing the instantiation. If we decided to change the name of the variable where we are defining the `if-then-else` function, we would break every call to `if-then-else` that instantiates A by name. But probably such renaming of variables in types for function definitions is relatively rare, so this should not trouble us too often. Note that the spaces around the equality symbol are required; you will get an error if you write `if_then_else_{A=\mathbb{B}}`.

Figure 2.1 has a proof of a very similar example about disjunction (“or”). We have an implicit boolean argument b , which we pattern match on by writing the patterns in curly braces, on the left-hand sides of the defining equations. In each case, the equality type refines to something which Agda can see is trivial. In the

```

||-same : ∀ {b} → b || b ≡ b
||-same{tt} = refl
||-same{ff} = refl

```

Figure 2.1: A proof about disjunction (“or”)

first equation, where `b` is instantiated to `tt`, the equality we are trying to prove is refined by Agda to

```
tt || tt ≡ tt
```

This is definitionally equal to

```
tt ≡ tt
```

and so we can trivially prove it with `refl`. As we were seeing above, two expressions in Agda are *definitionally equal* iff they can be proved equal using the defining equations for the function symbols involved (and one more equation which is built in to Agda, that is not needed yet for our examples so far).

2.4 Theorems with Hypotheses

Continuing our tour of `bool-thms.agda`, let us consider this theorem:

```

||≡ff1 : ∀ {b1 b2} → b1 || b2 ≡ ff → b1 ≡ ff
||≡ff1 {ff} p = refl
||≡ff1 {tt} ()

```

Note that the subscripted 1 can be written in Agda mode in `emacs` with `_1`. Let us make sure we can read the theorem that is being proved:

```
∀ {b1 b2} → b1 || b2 ≡ ff → b1 ≡ ff
```

Let us insert some parentheses (always legal in Agda) to make the syntax of this formula a little clearer:

```
∀ {b1 b2} → ((b1 || b2) ≡ ff) → (b1 ≡ ff)
```

This says that for any booleans `b1` and `b2`, if the disjunction (“or”) of `b1` and `b2` is `ff` (false), then `b1` must be `ff`. This makes sense: since a disjunction is true if either disjunct is true, if we have a false disjunction, then both disjuncts must be false, in particular the first disjunct. If `b1` were true, that would make the whole disjunction true, but this theorem is assuming it is false.

This is our first theorem with an assumption. Under the Curry-Howard isomorphism, to express an assumption, we write a function that takes in a proof of that

assumption, and then produces the proof of the desired result. Here, our assumption is

```
b1 || b2 == ff
```

and the final desired result is

```
b1 == ff
```

So in the type, we write

```
b1 || b2 == ff → b1 == ff
```

to indicate that the function we are writing takes in a proof of

```
b1 || b2 == ff
```

and then produces a proof of the desired result. Notice this fits the basic pattern for function types: a function type $A \rightarrow B$ is the type for functions with inputs of type A and outputs of type B .

Assumptions allow us to express more interesting theorems than we have so far, since many interesting properties of our programs are true only under some assumptions. Writing proofs when we have assumptions to take into account also becomes more complex. Let us consider the first defining equation for the proof:

```
||==ff1 {ff} p = refl
```

This is defining the function in question, namely

```
||==ff1
```

by giving patterns for several arguments. We are here considering the case where the first implicit argument `b1` is `ff`. So we have one pattern in curly braces:

```
{ff}
```

After this, we have a variable `p` corresponding to the sole explicit argument of this proof, namely the proof of our assumption. We actually do not need to use this variable on the right-hand side in this case, because if we are in the case where `b1` is `ff`, then our desired result is definitionally equal to just `ff == ff`. So this equation has just `refl` on the right-hand side, to prove that trivial equation `ff == ff`.

2.4.1 The absurd pattern

Now let us consider the second defining equation. Actually, it is not an equation at all, just a left-hand side of an equation containing the **absurd pattern** `()`. This is an interesting Agda feature we have not used up to now. Let us look at this left-hand side again:

```
||==ff1 {tt} ()
```

Where is the equals sign and the right-hand side? They are omitted because we are using the absurd pattern `()` to point out to Agda that the case which is being considered here is impossible.

Let us puzzle out what is happening. We have written our theorem with two implicit input arguments, `b1` and `b2`. The pattern we are looking at is instantiating `b1` with `tt`, using the curly braces for implicit arguments. If we instantiate `b1` with `tt` in the type of the theorem (which Agda is using to check this clause of the definition), we will get this type:

```
∀ {b2} → tt || b2 ≡ ff → tt ≡ ff
```

This type is obtained from the starting type for the theorem by just replacing `b1` with `tt` everywhere. Now look closely at what the desired final result has become:

```
tt ≡ ff
```

That doesn't look good! We cannot prove `tt` equal to `ff`, because they are distinct constructors. In general, Agda's type theory will never let us prove that two terms are equal, if those two terms begin with distinct datatype constructors. What are we going to do? We know we cannot prove this formula, so are we stuck? Is this really not a theorem?

Theorem proving can induce hand wringing, but in this case there is no need: if you are asked to prove something impossible, there is a way out, if some assumption you are using is also impossible. If you have to prove the impossible, but only if some impossible assumption is true, then your job is done, since that impossible assumption can never be satisfied by someone who wants to use your theorem. And in this case, our sole assumption is actually impossible to satisfy. Because the assumption is

```
tt || b2 ≡ ff
```

And this type is definitionally equal to

```
tt ≡ ff
```

That is because the definition of disjunction (from `bool.agda`) is this:

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

This code looks at the first argument, and if that first argument is `tt`, it just returns `tt`, without even looking at the second argument.

Agda's absurd pattern `()` is special notation which allows us to "quit early" when we have an obviously impossible assumption. An assumption is obviously impossible if it is definitionally equal to an equation between two terms which start with different constructors. Here our assumption is definitionally equal to `tt ≡ ff`, and the two terms in question (`tt` and `ff`) do indeed start with different constructors. So we can use the absurd pattern to tell Agda to take notice of the fact that

we have an impossible assumption, and let us off the hook for proving the desired result in this case. You simply write `()` in place of the pattern corresponding to the assumption. In this case, our theorem only has one explicit argument, which is the proof of the assumption. So we just write `()` there. Agda accepts this definition as a correct proof of our theorem.

2.4.2 An alternative proof

We could also have proved the theorem this way:

```
||≡ff₁ : ∀ {b1 b2} → b1 || b2 ≡ ff → b1 ≡ ff
||≡ff₁ {ff} p = refl
||≡ff₁ {tt} p = p
```

This is the same as the proof above, but for the second clause of the definition, we have an actual defining equation which just returns the proof `p` of the impossible assumption. The type of `p` is definitionally equal to `tt ≡ ff`, which is what we have for the desired result with `b1` instantiated to `tt`.

Is there any reason to prefer our first proof using the absurd pattern over this one? Yes, because the first proof is more robust to some minor changes we might conceivably make to the statement of the theorem. For example, suppose we changed our theorem to this one:

```
∀ {b1 b2} → b1 || b2 ≡ ff → ff ≡ b1
```

Here, I have just changed the desired result from `b1 ≡ ff` to `ff ≡ b1`. You would expect this makes little or no difference, and that is correct, for the first proof we wrote. The absurd pattern still can be used for the second clause of the definition, and the exact same proof we wrote is accepted by Agda:

```
||≡ff₁ : ∀ {b1 b2} → b1 || b2 ≡ ff → ff ≡ b1
||≡ff₁ {ff} p = refl
||≡ff₁ {tt} ()
```

But things do not adapt so smoothly for the second proof:

```
||≡ff₁ : ∀ {b1 b2} → b1 || b2 ≡ ff → ff ≡ b1
||≡ff₁ {ff} p = refl
||≡ff₁ {tt} p = p
```

Why? Because in this case, the type of `p` is definitionally equal to `tt ≡ ff`, but the desired result formula is `ff ≡ tt`. These are very similar, and one obviously follows from the other, but they are not definitionally equal. And so Agda will complain that on the right-hand side of that second defining equation, `p` has the wrong type. We can rectify this problem if we want, using an explicit proof of symmetry of equality:

```
||≡ff₁ : ∀ {b1 b2} → b1 || b2 ≡ ff → ff ≡ b1
```

```

||≡ff1 {ff} p = refl
||≡ff1 {tt} p = sym p

```

The `sym` operator is defined in `eq.agda`. It takes a proof of $x \equiv y$ and returns a proof of $y \equiv x$. So we can use it here to go from $tt \equiv ff$ (the type of `p`) to the desired result $ff \equiv tt$. But this is not necessary: the first proof using the absurd pattern is easier and shorter.

2.4.3 Matching on equality proofs

A little further on in `bool-thms.agda`, you will find this example theorem:

```

||-cong1 : ∀ {b1 b1' b2} → b1 ≡ b1' →
              b1 || b2 ≡ b1' || b2
||-cong1 refl = refl

```

As usual, let us make sure we understand what this theorem is saying, and then take a detailed look at the (very short) proof. The theorem is

$$\forall \{b1 \ b1' \ b2\} \rightarrow b1 \equiv b1' \rightarrow b1 \ || \ b2 \equiv b1' \ || \ b2$$

We can see that we have three universally quantified variables $b1$, $b1'$, and $b2$. Let us first insert some parentheses to make sure the structure a little clearer:

$$\forall \{b1 \ b1' \ b2\} \rightarrow (b1 \equiv b1') \rightarrow ((b1 \ || \ b2) \equiv (b1' \ || \ b2))$$

So this is saying: if $b1$ equals $b1'$, then the following two expressions are also equal:

```

b1 || b2
b1' || b2

```

This is expressing a special case of the general property, sometimes called **congruence**, that if x equals y , then any type containing x is equal to that same type with some of the x 's replaced by y . This property holds for definitional equality automatically. For propositional equalities $x \equiv y$, the general congruence property – or special cases as here – must be proved.

Now let us look carefully at the proof. The variables $b1$, $b1'$, and $b2$ are all implicit, since they are declared with curly braces in the type. There is only one explicit argument, and it is the assumption that $b1 \equiv b1'$. As noted earlier in this chapter, there is actually only one way to construct an equality proof, and that is with the `refl` constructor. When we write `refl` as the pattern for this case, Agda deduces that since `refl` can only prove something of the form $t \equiv t$, then $b1$ must be exactly $b1'$. Actually, because Agda always takes definitional equality into account, it knows that $b1$ must be definitionally equal to $b1'$. Indeed, Agda refines the desired type using this definitional equality. So we now really only need to prove

```
((b1 || b2) == (b1 || b2))
```

And this formula can obviously be proved by `refl`.

This trick of using `refl` as a pattern for an equality assumption seems really great. Can we just use it everywhere? What about the proof we considered at the start of Section 2.4? Would the following proof work?

```
||==ff1 : ∀ {b1 b2} → b1 || b2 == ff → ff == b1
||==ff1 refl = refl
```

The answer is no. There is an important rule about using type refinements in Agda. Agda will not allow us to refine types in such a way that non-variable terms are discovered to be definitionally equal to other such terms. Here, refinement would force Agda to conclude that the following two non-variable terms are definitionally equal:

```
b1 || b2
ff
```

Agda cannot reason internally about assumed definitional equalities between non-variable terms. So it will not allow us to use `refl` as a pattern in this case.

One final note about pattern matching on equality proofs: a detail has been hidden here because the booleans `b1`, `b1'`, and `b2` are implicit arguments to this proof. Suppose we decide to include them in the pattern on the left-hand side of the equation, using the curly-braces notation we saw in Section 2.3 above:

```
||-cong1 : ∀ {b1 b1' b2} → b1 == b1' → b1 || b2 == b1' || b2
||-cong1{b1}{b1'}{b2} refl = refl
```

If we do this, Agda will complain with a strange error:

```
b1 != b1' of type ℬ
when checking that the pattern refl has type b1 == b1'
```

The problem is that to type check this pattern, we cannot have possibly distinct values `b1` and `b1'`. Agda is deducing that `b1` and `b1'` must be definitionally equal, and for this reason does not allow them to be distinct variables. The solution to this is to use a **dot pattern**:

```
||-cong1 : ∀ {b1 b1' b2} → b1 == b1' → b1 || b2 == b1' || b2
||-cong1{b1}{.b1}{b2} refl = refl
```

Writing a dot (“.”) in front of a term in a pattern tells Agda that the term written there is not a subpattern to match, but rather has a form which is dictated by the type of the whole pattern. We did not need to do this in the first proof we considered at the start of this section, because we did not match on the implicit arguments.

2.4.4 The `rewrite` directive

One further important theorem-proving feature of Agda is the `rewrite` directive which can be used in equations. We will see this in a similar example to the one we just solved by pattern matching on an equality proof:

```
||-cong2 : ∀ {b1 b2 b2'} → b2 ≡ b2' → b1 || b2 ≡ b1 || b2'
||-cong2 p rewrite p = refl
```

This theorem is quite similar to the one in the previous section. We could prove it the same way, but let us see an interesting alternative. We just introduce a pattern variable `p`, which has type `b2 ≡ b2'`. Now, whenever you have such a pattern variable, you can tell Agda you wish to rewrite every occurrence of the left-hand side, as it appears in the current goal, with the right-hand side. The goal is just the formula which we have to prove on the right-hand side of the current equation. The `rewrite p` directive tells Agda to transform the `b2` in

```
b1 || b2 ≡ b1 || b2'
```

to `b2'`. So the goal (the type Agda is checking against the right-hand side of the equation) becomes

```
b1 || b2' ≡ b1 || b2'
```

And of course, this does follow by `refl`.

In general, `rewrite` works as follows in Agda. Suppose you have a proof `p` of some equation `X ≡ Y`. The proof `p` is simply any expression which has `X ≡ Y` as its type; it does not have to be a variable, but could be a call to a lemma that has already been proved, for example. Then `rewrite p` instructs the Agda type checker to look in the goal for any occurrences of `X`, and transform those into `Y`. `X` and `Y` could be complex expressions; they do not have to be just variables. The `rewrite` directive will come in very useful for later proofs, and we will see it in action more below.

2.5 Further examples

Let us look at a couple more examples of theorem proving with boolean operations, from `bool-thms.agda`. First, let us see an example of proving a “polymorphic theorem”. This sounds fancier than it needs to. We are really just going to prove a theorem that includes a universal quantification over types (and type levels), so that we can reason about type-polymorphic functions like `if-then-else`. Here is our example theorem:

```
ite-same : ∀ {a} {A : Set a} →
           ∀ (b : ℤ) (x : A) →
```

```

      (if b then x else x) ≡ x
ite-same tt x = refl
ite-same ff x = refl

```

As usual, let us first read through the type (i.e., the formula being proved) carefully:

$$\forall \{\ell\} \{A : \text{Set } \ell\} \rightarrow \forall (b : \mathbb{B}) \ (x : A) \rightarrow (if\ b\ then\ x\ else\ x) \equiv x$$

This says that for every type level ℓ and every type A at that level, and for all booleans b and values x of type A , the following expression equals x :

```
(if b then x else x)
```

This if-then-else expression returns x if b is true, and also if it is false. So of course it is equal to x !

How does the proof of this work in Agda? Well, since the type level ℓ and the type A are implicit arguments (since they are introduced with curly braces in the type), we do not need to write any arguments for them in the patterns on the left-hand sides of our defining equations. Then we are reasoning by cases on b . If b is `tt`, the rest of the desired type becomes

$$\forall (x : A) \rightarrow (if\ tt\ then\ x\ else\ x) \equiv x$$

This type is definitionally equal to

$$\forall (x : A) \rightarrow x \equiv x$$

And that formula is easily proved by just taking in the argument x (without any pattern matching, since the type A could be anything, and so we do not have any information about what x could look like); and then returning `refl`. The same works for the second case, where b is `ff`. So the fact that this theorem includes a quantification over the type level ℓ and the type A at that level does not lead to any extra complexity in the proof.

Let us see one more example proof from `bool-thms.agda`, to close this chapter. We have seen above that if we have an assumption which is obviously impossible – because it is an equality between two terms which start with different constructors, like `tt` and `ff` – then we can use the absurd pattern `()` to point out to Agda that the assumption is impossible, and stop early in our code to handle that case. Here we use this same approach to prove that if `ff ≡ tt` holds, then we can prove any formula P we want:

```

B-contr : ff ≡ tt → ∀ {P : Set} → P
B-contr ()

```

The name of this theorem is `B-contr` for “boolean contradiction”. The formula we are proving can be written with some more parentheses this way:

$$(ff \equiv tt) \rightarrow (\forall \{P : \text{Set}\} \rightarrow P)$$

We are saying that if $ff \equiv tt$ is true, then the universal formula

$$\forall \{P : \text{Set}\} \rightarrow P$$

is also true. That formula says that for every formula P , this P is true. Not every formula is true, in fact, so this is another way of expressing a falsehood in Agda. We are effectively saying that if $ff \equiv tt$ is true, then something false is true. This can be handy in a case where our assumptions entail the obviously impossible equality $ff \equiv tt$, but it is not convenient to use the absurd pattern. The absurd pattern is, of course, how we show Agda that there is nothing to prove in this case; the defining clause for this theorem is just

```
ℬ-contra ()
```

where the absurd pattern is written for the first argument to `ℬ-contra`, namely the assumption of $ff \equiv tt$.

2.6 Conclusion

In this chapter, we have gotten our first taste of theorem proving in Agda. The theorems we proved are very simple, and their proofs are easy. The point was, of course, to see how to write these proofs in Agda. We saw a number of important features:

- The equality type $t \equiv t'$ expresses equality of expressions t and t' , which must have the same type.
- The basic way to prove an equality is with `refl`, which will prove $t \equiv t'$ for any *definitionally equal* t and t' .
- Two expressions are definitionally equal in Agda if they can be proved equal using their defining equations.
- A proof in Agda is just a program, which is defined like all other Agda programs, using pattern matching.
- Pattern matching can *instantiate* universally quantified variables which appear in the type. This has the effect of specializing the desired formula to the specific instances of those variables.
- Arguments can be declared implicit with curly braces in the type. We can still pattern match on implicit arguments, by writing curly braces around the pattern, on the left-hand side of an equation.
- Assumptions in theorems are just treated as extra arguments to the programs which are serving as proofs of those theorems.
- The absurd pattern `()` can be used when an assumption is definitionally equal to an obviously impossible equality, where the equated terms start

with different datatype constructors (like `tt` and `ff`).

- If you use `refl` as the pattern for an equality assumption $t \equiv t'$, Agda will try to refine the type by making t and t' definitionally equal. This will fail unless one of these expressions is definitionally equal to a variable.

In the next chapter, we will take a big step forward in programming and theorem proving complexity, by considering recursive functions on the natural numbers.

2.7 Exercises

1. Pick a theorem from `bool-thms.agda` or `bool-thms2.agda`, and reprove it yourself.
2. Think of a theorem about the boolean operations that is missing from the IAL (i.e., from `bool-thms.agda` and `bool-thms2.agda`), and prove it.
3. Which of the following formulas could be proved by `refl` (there may be several correct answers)?
 - (a) $tt \equiv tt$
 - (b) $ff \equiv ff$
 - (c) $ff \equiv tt$
 - (d) $ff \ \&\& \ ff \equiv \sim \ tt$
 - (e) $tt \ \&\& \ x \equiv x$, where x is a variable of type \mathbb{B}
 - (f) $x \ \&\& \ tt \equiv x$, where x is a variable of type \mathbb{B}

Chapter 3

Natural Numbers

The natural numbers are the nonnegative integers

$$0, 1, 2, 3, \dots$$

The notation \mathbb{N} is used for the set of natural numbers. There are quite a few differences between the mathematical natural numbers and numeric types in programming languages like Java and many others. In Java, for example, the `int` type, in addition to including negative numbers, only allows you to represent a finite set of numbers. There are (infinitely!) many numbers which are too big to fit into the 32 bits which the Java language specification says is the size of an `int`. Any number which needs more than that many bits cannot be represented as an `int`. Such restrictions are imposed by the arithmetic hardware in a modern microprocessor, which operates on numbers of some maximum binary representation. There are many ways to get around this while retaining efficient arithmetic operations on large numbers.

Agda does not provide any built-in datatypes, so no built-in numeric datatypes, either. We could certainly define a datatype of binary numbers. Using such a datatype for heavily numeric computation would be very slow, because we would not be taking advantage of the machine's arithmetic hardware if we wrote our own addition and multiplication functions on binary numbers represented with a user-defined datatype. Efficient numeric computation would really require some kind of axiomatization of the machine integers, and this is not something we will undertake. So efficient numeric computation is not something that will be possible with Agda as we are going to use it.

Instead, we will work with so-called **Peano** numbers. These are numbers represented not in binary, but in unary. Unary could be thought of as caveman counting. To write 1, we make one scratch along the cave wall. To write 2, we make two scratches. This representation is exponentially inefficient compared to binary or decimal notation, since a number like 1000 can be represented with just 4 symbols in decimal (a 1 and then three 0's), while it requires a hand-cramping 1000 marks on the cave wall. Your savvy caveman will quit in disgust – maybe to go invent positional notation like decimal (or sexagesimal if his cave is in Babylon)!

Operations on Peano numbers are much easier to reason about than operations on binary or decimal numbers, because the representation of numbers is so simple. For mathematical reasoning, as opposed to numeric programming, Peano numbers are the go-to representation scheme. They also make for an excellent

testbed for learning some more mathematically interesting theorem proving in Agda. Even if we tackle numeric computation by axiomatizing hardware arithmetic, we will likely want to use Peano numbers as part of the description of what the hardware is doing.

3.1 Peano Natural Numbers

The code in this section can be found in `nat.agda` in the Iowa Agda Library. Near the top of the file, you will find the inductive definition of the type `ℕ` for the Peano natural numbers:

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

Since this is only the second datatype definition we have considered (the first was for the booleans, Section 1.1), let us look at it carefully. We have the `data` keyword telling Agda that we are declaring a datatype. Then we have the name of the type, which is `ℕ`. Then we are giving the type of this datatype, which is `Set` (see Section 1.1.1). Then we have the declarations of the **constructors** of the `ℕ` type. There are two: `zero` is for zero (obviously), and `suc` is for the successor of a number. So here is how we would write the first few numbers as elements of the `ℕ` datatype:

```
0 = zero
1 = suc zero
2 = suc (suc zero)
3 = suc (suc (suc zero))
...
```

Notice that to call the constructor `suc` on an argument like `zero`, we write `suc zero`, and for nested calls, we need to use some parentheses:

```
suc (suc (suc zero))
```

In general in Agda (and in other functional languages like Haskell), we have to use parentheses around nested function calls. Otherwise, without parentheses, Agda would misinterpret an expression like

```
suc suc suc zero
```

as an application of `suc` to three arguments: `suc`, `suc`, and then `zero`. Actually, we could use syntax declarations (discussed in Section 1.3) to tell Agda to treat the above term as a nested application – but we will not usually write out such terms, and so we will not bother. Agda has built-in support to turn decimal notation like `3` into a term like the one above, built using `suc` and `zero`. In fact, to enable this notation, `nat.agda` has the following pragmas a little below the definition of `ℕ`:

```
{-# BUILTIN NATURAL ℕ    #-}
```

If you ask Agda to normalize `suc 3` (using Control-c Control-n), it will report 4 as the result. But this is just cosmetic: Agda views 4 as definitionally equal to `suc 3` (and to `suc (suc 2)`, etc.). And if you ask Agda what the type of 3 is (using Control-c Control-d), you will see \mathbb{N} , as expected.

Around the same place in `nat.agda`, there are a few syntax declarations, for smoother parsing of expressions built using the arithmetic operators, which we will consider next.

3.2 Recursive Definition of Addition

In mainstream programming languages, we mostly take the definition of arithmetic operations like addition and multiplication for granted (unless we get bitten by a bug due to arithmetic overflow, for example, when the result of a computation is too big to fit into a `int` or other finite built-in datatype). In Agda and other theorem-proving languages, the natural numbers and their operations are usually defined. Here is the definition of addition, from a little further in `nat.agda`:

```
_+_ : ℕ → ℕ → ℕ
zero + n = n
suc m + n = suc (m + n)
```

This definition is very similar to the ones we saw above, for boolean operations like conjunction (`_&&_`). We first have the name of our function, which is `_+_`. We are using the underscores to tell Agda that this will be an infix operator: the arguments go to the left and right of the symbol when it is used in an expression. If it is used without two arguments, we have to write `_+_`. Then we have the type of the function, which is

$$\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$$

As discussed in Section 1.5, the arrows are really grouped like this:

$$\mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$$

So addition is a function which takes in the first number to add, and then returns a function which is waiting for the second number and will then return their sum. So if you ask Agda for the type of a *partial application* like

```
_+_ 2
```

using Control-c Control-d, you will get

$$\mathbb{N} \rightarrow \mathbb{N}$$

And of course, if you ask for the type of $2 + 4$ you will get \mathbb{N} .

Now let us look carefully at the defining equations for addition:

```
zero  + n = n
suc m + n = suc (m + n)
```

These follow the same pattern as for the boolean operations. We have some equations, whose left-hand sides are patterns matching the possible forms for calls to the addition function. The first pattern covers all calls where the first summand (the first argument to addition) is `zero`. The second pattern covers all calls where the first summand looks like `suc m`, where `m` is a pattern variable matching any natural number. Since every value of type \mathbb{N} is either `zero` or `suc m` for some other value `m` of type \mathbb{N} , this indeed covers all the cases for what the first summand could be. And since each of the two left-hand sides just lists a pattern variable (`n`) for the second summand, our two patterns have covered all the possible calls to the addition function.

Phew! We are not used to thinking this hard about the definition of addition, right? We usually just take it for granted. But when we are building everything up from first principles as Agda does, these details must be considered. Now that we know we are correctly covering all the possible inputs, we can consider what the right-hand sides present as the outputs. In the first defining equation, the output is just `n`. So we are returning `n` if the first input is `zero` and the second is `n`. In the second defining equation, the first input is `suc m`, and the second is `n`. The right-hand side says that to add these numbers, what we should do is recursively add `m` and `n`, and then take the successor of that.

Notice that we are making a recursive call on a **structurally smaller** expression: we are defining addition for the case where the first input is `suc m`, and we are making a recursive call to addition on `m`. Since `m` is subdata of `suc m`, it is structurally smaller. If our functions only make recursive calls on structurally smaller inputs, for one chosen input argument, then they are guaranteed to terminate. Agda requires all functions to terminate, to ensure that we cannot prove a theorem like $A \rightarrow B$ by just writing a function which takes in an input of type A and then immediately goes into an infinite loop, like:

```
f a = f a
```

An infinite loop like this allows us to prove any formula we want, which would destroy the usefulness of Agda as a theorem-proving language. Such a language is only interesting if it distinguishes some formulas as true and others as false. If we can prove every formula by writing an infinite loop, then we are saying all formulas are true.

Here is how the above code for addition will execute on an example, adding 3 and 2. I am underlining each expression as it is transformed using one of the defining

equations for addition above.

$$\begin{aligned}
 3 + 2 &= \\
 &\frac{(\text{suc } (\text{suc } (\text{suc } \text{zero}))) + (\text{suc } (\text{suc } \text{zero}))}{\text{suc } ((\text{suc } (\text{suc } \text{zero})) + (\text{suc } (\text{suc } \text{zero})))} = \\
 &\text{suc } (\text{suc } ((\text{suc } \text{zero}) + (\text{suc } (\text{suc } \text{zero})))) = \\
 &\text{suc } (\text{suc } (\text{suc } (\text{zero} + (\text{suc } (\text{suc } \text{zero})))))) = \\
 &\text{suc } (\text{suc } (\text{suc } (\text{suc } (\text{suc } (\text{suc } \text{zero})))))) = \\
 &5
 \end{aligned}$$

The definition of addition is pulling uses of `suc` from the first summand out, until they have all been removed and the first summand is just `zero`. We are so used to performing addition with decimal numbers that this algorithm may seem strange, even to be thinking about. But it is simpler and more basic than decimal addition.

3.3 Simplest Theorems about Addition

The simplest theorems about addition concern adding 0 to a number. You can find the examples we are about to consider in `nat-thms.agda`. The first one is very easy:

```

0+ : ∀ (x : ℕ) → 0 + x ≡ x
0+ x = refl

```

The name of the theorem is `0+`, and the theorem itself (the type) is

```

∀ (x : ℕ) → 0 + x ≡ x

```

This says: “for any natural number x , 0 plus x equals x .” This is very obvious from the definition we saw above for addition:

```

_+_ : ℕ → ℕ → ℕ
zero  + n = n
suc m + n = suc (m + n)

```

We just have to remember that Agda automatically treats 0 as the constructor `zero` of the `ℕ` datatype, and then we can see that the `0+` theorem should just follow from definitional equality. And that is exactly how the proof works. We take in an argument x , and then just return `refl`, because Agda can see, from the first defining equation for addition, that `zero + x` is definitionally equal to x .

That was so easy, it feels like we are still living on the Easy Street of theorems about boolean operations we were considering in the previous chapter. Let’s try to prove the same theorem, but with the order of the summands reversed:

```

+0a : ∀ (x : ℕ) → x + 0 ≡ x
+0a x = refl

```

If you try this (by copying it into `nat-thms.agda`, say, and doing `Control-C Control-L`), you will see the following error message:

```
x + 0 != x of type ℕ
when checking that the expression refl has type x + 0 ≡ x
```

Agda is saying that it was trying to confirm that `refl` has type $x + 0 \equiv x$, which it does not have. And if we look again at the definition of addition, we can see why:

```
_+_ : ℕ → ℕ → ℕ
zero  + n = n
suc m + n = suc (m + n)
```

Our code here is doing pattern matching on the first argument. But in the theorem we are trying to prove, we have a variable for the first argument to addition:

```
∀ (x : ℕ) → x + 0 ≡ x
```

So Agda's type checker has no chance to apply the first (or the second) defining equation for addition to simplify this expression. It can only simplify $0 + x$, not $x + 0$.

So how do we solve this? The theorem is true, isn't it? (This is a very important thing to ask yourself when you get stuck trying to prove something, because if the theorem is false, you could think as long and as hard as you like, and you will still never prove it – assuming Agda is consistent!) Let's try to understand how we would prove this theorem, by considering some small examples (another good method to use when you are stuck proving something). Suppose we have $3 + 0$. This will be normalized this way:

$$\begin{aligned}
 3 + 0 &= \\
 &= \text{succ} (\text{succ} (\text{succ zero})) + \text{zero} = \\
 &= \text{succ} ((\text{succ} (\text{succ zero})) + \text{zero}) = \\
 &= \text{succ} (\text{succ} ((\text{succ zero}) + \text{zero})) = \\
 &= \text{succ} (\text{succ} (\text{succ} (\text{zero} + \text{zero}))) = \\
 &= \text{succ} (\text{succ} (\text{succ zero})) = \\
 &= 3
 \end{aligned}$$

We are just repeatedly peeling off the `succ` from the first argument, until we reach `zero`, our base case. Then we turn `zero + zero` into `zero`, and we have the same number of `succ` constructors nested above that as we had `succ` constructors in our first argument originally (namely, 3). Does this help us see how to prove this?

Well, if we are trying to prove $3 + 0 \equiv 3$, this is definitionally equal to proving

```
succ (2 + 0) ≡ succ 2
```

And that would follow easily, if only we knew that the argument to `succ` on the left-hand side – namely $2 + 0$ – were equal to 2. But that just exactly fits the

pattern of this theorem we are trying to prove! If only our proof could just call itself recursively, we would be able to get a proof at this point that $2 + 0 \equiv 2$, and then use that to complete the proof for $3 + 0$. Fortunately, our proofs are allowed to call themselves recursively, since proofs are just programs. Let us see the code for `+0`, which does this:

```
+0 : ∀ (x : ℕ) → x + 0 ≡ x
+0 zero = refl
+0 (suc x) rewrite +0 x = refl
```

We are case-splitting on the form of the input `x` of type `ℕ`. Either it is `zero` (the first equation), or else it looks like `suc x` for some other `x` of type `ℕ`. In the first case, the proof is easy. Just think about what the Agda type checker will do in this case. It is checking the type

$$\forall (x : \mathbb{N}) \rightarrow x + 0 \equiv x$$

against the equation

```
+0 zero = refl
```

So it instantiates the variable `x` in the type to `zero`. Then Agda will check that the right-hand side of the equation has type $x + 0 \equiv x$, with the instantiation of `x` to `zero`. In other words, it is checking that `refl` has type

$$0 + 0 \equiv 0$$

which is definitionally equal (that is, equal if you just run code, in this case the code for addition) to $0 \equiv 0$. And `refl` indeed has this type.

Now let us look at the second defining equation for the `+0` proof:

```
+0 (suc x) rewrite +0 x = refl
```

We are instantiating the variable `x` in the type with `suc x`. As an aside: it is important to realize that the `x` in the type is not the same as the `x` pattern variable in `suc x`. The scope of a variable in a dependent function type like

$$\forall (x : \mathbb{N}) \rightarrow x + 0 \equiv x$$

is the rest of the type following the declaration $(x : \mathbb{N}) \rightarrow$, namely $x + 0 \equiv x$. The scope of a pattern variable like the `x` on the left-hand side of the equation is just that equation.

Returning to the second defining equation for `+0`: since the `x` in the type has been instantiated to `suc x` (for the pattern variable `x`, which we were just noting is different), Agda is left trying to check that `refl` has type

$$(\text{suc } x) + 0 \equiv \text{suc } x$$

This is the **goal formula** (the thing we are trying to prove). Now what happens next? Well, the goal formula can be simplified using the definition of addition. Let us look at that definition again:

```

_+_ : ℕ → ℕ → ℕ
zero  + n = n
suc m + n = suc (m + n)

```

The second defining equation makes our goal formula definitionally equal to

```
suc (x + 0) ≡ suc x
```

How? Well, if you instantiate `m` in the second defining equation for addition with `x`, and `n` with `0`, then the second defining equation becomes:

```
suc x + 0 = suc (x + 0)
```

So the left-hand side of the goal formula as it was written above can be simplified by Agda, and the goal becomes

```
suc (x + 0) ≡ suc x
```

As noted in our informal discussion above, if only we could make a recursive call to `+0` to obtain the fact that

```
x + 0 ≡ x
```

then we could rewrite with that fact to simplify our goal to a trivial statement

```
suc x ≡ suc x
```

And this is exactly what the second defining equation for `+0` does:

```
+0 (suc x) rewrite +0 x = refl
```

Recall from Section 2.4.4 that `rewrite` can be used with any proof of an equation $L \equiv R$ (for some left-hand side `L` and right-hand side `R`) to transform `L` into `R` anywhere in the instantiated type we are checking for this clause. Here, we are using `rewrite` with the proof

```
+0 x
```

What is the type of this proof? It is $x + 0 \equiv x$, based on the declared type for `+0`. So following this `rewrite` directive, Agda will replace any occurrences of `x + 0` with just `x` in the instantiated type against which we are checking the second defining equation. So it will rewrite `suc (x + 0)` to just `suc x` in the goal formula, and then `refl` is all we need to conclude the proof.

This is really quite amazing, so let us take another look at the complete proof:

```

+0 : ∀ (x : ℕ) → x + 0 ≡ x
+0 zero = refl
+0 (suc x) rewrite +0 x = refl

```

Even though the explanation was a bit detailed, the final resulting proof is a marvel of elegance and concision. This is largely due to the use of recursive equations with pattern matching, and the `rewrite` directive.

Recursive proofs are called proofs by **induction** in mathematics. We must prove our property for the base case of 0, and then show the step (or inductive) case: if the property holds for x , then it also holds for $x + 1$. Here, the property in question is $x + 0 \equiv x$. The base case is the first defining equation, where we must prove this property with x instantiated to `zero`. The second defining equation corresponds to the step case of the induction. We must prove the property with x instantiated to `suc x` (for a different x). This corresponds to proving the property for $x + 1$. But we are allowed to assume that the property is true for x . This corresponds to making recursive calls like `+0 x`. We are only allowed to use our property for x in the step case of the mathematical induction, where we have to prove it for $x + 1$. Similarly, we can only make a recursive call to `+0` on x , which is structurally smaller than `suc x`, for which we have to prove the property in the second defining equation. So proofs by mathematical induction become (terminating) recursive programs, like `+0`. This is another important example of the Curry-Howard isomorphism.

3.4 Associativity of Addition and Working with Holes

Let us prove now a more interesting theorem about addition, namely associativity. The proof is `+assoc` in `nat-thms.agda`. Associativity says that if we have three numbers to add, it does not matter if we add the first two and then the third, or else add the first to the sum of the second two: we will get the same answer. This is simpler to express with a formula:

$$\forall (x \ y \ z : \mathbb{N}) \rightarrow x + (y + z) \equiv (x + y) + z$$

A constructive proof of this formula is a function which takes in arguments x , y , and z , and returns a proof of $x + (y + z) \equiv (x + y) + z$. This is certainly a more impressive property than the fact that $x + 0 \equiv x$ which we proved in the last section. Amazingly, the proof is almost exactly the same! The big question we have to answer when we have multiple variables of type \mathbb{N} like this in our theorem is, which variable should we do induction on? The best rule of thumb is that you should do induction on a variable which is used where a recursive function will perform pattern matching on it; and if there are several such variables, prefer the one where such pattern matching will happen more. Since addition as we have defined it does pattern matching on the first argument, it is a good idea to consider doing induction on a variable used as the first argument to addition. Here, we have two options, x and y , which are both used as first arguments to addition. But x is used twice this way, while y is used only once. So we will try doing induction on x . This rule of thumb does not always work, but it is a good place to start.

In this case, the rule of thumb works, resulting in the proof in Figure 3.1. Let us walk through this in detail. We are case-splitting on x , and just using variables y and z for the other two arguments to `+assoc`, in both defining equations. In the first defining equation, since we have instantiated x to `zero`, the goal formula

```
+assoc : ∀ (x y z : ℕ) → x + (y + z) ≡ (x + y) + z
+assoc zero y z = refl
+assoc (suc x) y z rewrite +assoc x y z = refl
```

Figure 3.1: Proof of associativity of addition

becomes:

```
zero + (y + z) ≡ (zero + y) + z
```

Now, this should simplify, based on the definition of addition. We can actually get Agda to show us the simplified goal. Try this out yourself: first replace the right-hand side of the first defining equation with a question mark:

```
+assoc zero y z = ?
```

This tells Agda that we have a hole in our proof, which we still need to fill in. Agda has some very nice features for working with holes. If we load the file with Control-c Control-l, we will see the following (where the `{ } 0` is highlighted in green):

```
+assoc zero y z = { } 0
```

Now if you position your cursor in the hole (i.e., between the curly braces), there are some new Agda commands you can use:

- **Viewing the normalized goal and context.** If you add the code in the appendix section “Some Extra Emacs Definitions” to your `.emacs` file, you can type `Control-c Control-,` (control followed by the comma character) to see the normalized goal. Agda normalizes the goal by running all code in it, if possible. You will also see the assumptions (if any) that you have to work with. The set of these is called the context. If you do not add that extra emacs code, you will see what Agda gives as the default, which is a “simplified” goal. Unfortunately, there is no documentation about what this means exactly, and I find the normalized goal more helpful generally.
- **Viewing the normalized goal, context, and the type of an expression.** If you type a proof – or any expression – into a hole (again, between the curly braces) and type Control-c Control-. (that is Control and then a period), you will see the same things as with `Control-c Control-,` as well as the type of the expression you entered in the hole. This can be very useful if you think you have part of the proof worked out to fill the hole, but you want to check the type of what you have so far. (Again, without the additions to your `.emacs` file, you will see “simplified” rather than normalized expressions.)
- **Solving the hole.** If you have a proof entered in the hole and want Agda to try to use that proof in place of the hole, type Control-c Control-r. This will fail if the expression in the hole cannot be used to fill in the hole because the expression’s type does not match the type of the hole.

If we type Control-c Control-, in the hole above for `+assoc`, we will see this:

```
Goal: y + z ≡ y + z
```

```
-----
```

```
z : ℕ
```

```
y : ℕ
```

Agda is printing the simplified goal, which is just

```
y + z ≡ y + z
```

It also prints the types of the pattern variables `y` and `z`. We can prove this goal with `refl`, as usual.

Now let us look at the second defining equation for `+assoc`:

```
+assoc (suc x) y z rewrite +assoc x y z = refl
```

We are instantiating the variable `x` in the type with `suc x`, where as explained in detail in Section 3.3, the `x` in `suc x` is different from the `x` in the type. If we put a hole on the right-hand side of this second defining equation, we can see what the goal is with the instantiation of `x` to `suc x`. Let us also remove the `rewrite` directive for a moment, just to understand how this case of the proof works:

```
+assoc (suc x) y z = ?
```

If we load the file with Control-c Control-l, then this will turn into

```
+assoc (suc x) y z = { }0
```

Also, Agda will print out for us the type of hole `0`, in a small buffer at the bottom of the `emacs` window:

```
?0 : suc x + (y + z) ≡ suc x + y + z
```

This is exactly what we would expect, with `x` instantiated to `suc x`. To see better how to proceed with the proof, we should do Control-c Control-, with our cursor in hole `0`. Agda will show us this:

```
Goal: suc (x + (y + z)) ≡ suc (x + y + z)
```

```
-----
```

```
z : ℕ
```

```
y : ℕ
```

```
x : ℕ
```

Our context lists the variables `x`, `y`, and `z` of type `ℕ`, with the simplified goal. Simplification has pulled `suc` out of the addition expressions. We can clearly see here that if only we had a proof of

```
(x + (y + z)) ≡ (x + y + z)
```

we could rewrite the left-hand side of the goal formula to be identical to the right-hand side. And of course, by making a recursive call to `+assoc` (i.e., by applying

our induction hypothesis), we can get such a proof. This is exactly what the second defining equation for `+assoc` does. Here is the complete proof once again:

```
+assoc : ∀ (x y z : ℕ) → x + (y + z) ≡ (x + y) + z
+assoc zero y z = refl
+assoc (suc x) y z rewrite +assoc x y z = refl
```

3.5 Commutativity of Addition

In this section we will see the proof of the following theorem:

$$\forall (x \ y : \mathbb{N}) \rightarrow x + y \equiv y + x$$

This says that for any natural numbers x and y , you get the same answer if you add x and y or if you add y and x . This is called commutativity of addition.

Let us see what happens if we try to prove this from scratch. We have the usual question of which of our two natural-number variables x and y to do induction on. Here, the rule of thumb mentioned in the previous section is inconclusive: each variable occurs as the first argument to a call to addition. So let us just try pattern-matching on the first one, and then let us write some holes to get some more information:

```
+comm : ∀ (x y : ℕ) → x + y ≡ y + x
+comm zero y = ?
+comm (suc x) y = ?
```

To try this out, just comment out the code for the existing `+comm` proof in `nat-thms.agda`, and use the above code instead (recall that you can comment out code by placing it between `{-` and `-}`). If we load the file with Control-c Control-l, Agda will change the code to this:

```
+comm : ∀ (x y : ℕ) → x + y ≡ y + x
+comm zero y = { }0
+comm (suc x) y = { }1
```

It will also tell us the types of these two holes:

```
?0 : zero + y ≡ y + zero
?1 : suc x + y ≡ y + suc x
```

Let us work on hole 0 first. If you put your cursor in the hole and type Control-c Control-, you will see this:

```
Goal: y ≡ y + 0
```

```
-----
y : ℕ
```

This looks promising: we have simplified the left-hand side to just y , and we can rewrite the right-hand side using `+0`, which we proved in Section 3.3. So we can change our code so far for `+comm` to this (the number of the remaining hole will be updated when you load the file again with Control-c Control-l):

```
+comm : ∀ (x y : ℕ) → x + y ≡ y + x
+comm zero y rewrite +0 y = refl
+comm (suc x) y = { }0
```

We are using the `rewrite` directive in the first defining equation. As we have seen before, to use this directive, you must put a proof of an equation immediately after the `rewrite` keyword. Here, the equation in question is

$$y + 0 \equiv y$$

We do not write this equation itself after `rewrite`. Rather, we have to write a proof of this equation. And `+0 y` is a proof of $y + 0 \equiv y$ for any y we give to `+0` as an argument. Then the `rewrite` directive will simplify our goal to just $y \equiv y$, which is proved by writing `refl` on the right-hand side of the equation.

Let us look now at the remaining hole (in the second defining equation) for our definition of `+comm`. If we do Control-c Control-, in this hole, Agda will tell us:

```
Goal: suc (x + y) ≡ y + suc x
```

```
-----
y : ℕ
x : ℕ
```

Now, this also looks pretty promising. We can certainly apply our induction hypothesis to rewrite $x + y$ on the left-hand side of the goal to $y + x$. To do this, we use a `rewrite` directive. The proof is now:

```
+comm : ∀ (x y : ℕ) → x + y ≡ y + x
+comm zero y rewrite +0 y = refl
+comm (suc x) y rewrite +comm x y = { }0
```

If we reload this and ask Agda about hole 0, it will tell us:

```
Goal: suc (y + x) ≡ y + suc x
```

```
-----
y : ℕ
x : ℕ
```

Ok, this is looking pretty good. We would be done if we could just rewrite the right-hand side using this equation:

$$y + \text{suc } x \equiv \text{suc } (y + x)$$

How can we get a proof of this equation? Well, we actually have to prove this by a separate inductive argument. The lemma `+suc` defined a little above `+comm` in `nat-thms.agda` does this:

```
+suc : ∀ (x y : ℕ) → x + (suc y) ≡ suc (x + y)
+suc zero y = refl
+suc (suc x) y rewrite +suc x y = refl
```

The proof is very similar to the proof of `+0` which we considered above (Section 3.3). Both these lemmas, `+suc` and `+0`, show what happens when you have a constructor term (either `zero` or `suc x`) as the second argument of addition. We already know what happens when such a term is the first argument of addition, because we defined addition (in `nat.agda`) to use pattern matching on the first argument:

```
+_ : ℕ → ℕ → ℕ
zero + n = n
suc m + n = suc (m + n)
```

The final proof for `+comm` looks like this:

```
+comm : ∀ (x y : ℕ) → x + y ≡ y + x
+comm zero y rewrite +0 y = refl
+comm (suc x) y rewrite +suc y x | +comm x y = refl
```

This demonstrates one feature of `+comm` which we did not use yet, namely rewriting with multiple equations. We just need to separate the proofs of those equations, following the `rewrite` keyword, with vertical bars:

```
rewrite +suc y x | +comm x y
```

3.6 Multiplication

Let us return to `nat.agda` to see the definition of multiplication:

```
_*_ : ℕ → ℕ → ℕ
zero * n = zero
suc m * n = n + (m * n)
```

This code follows the familiar pattern for defining functions in Agda. First we choose the name of our function, which in this case is `_*_`. The underscores indicate that the arguments to this function will be written on its left and right sides. Then we have the type, which is $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$, and then defining equations. This definition is case splitting on the first argument to multiplication. If it is `zero`, we return `zero` no matter what the value of the second argument is. That is what the first defining equation says. If it is `suc m` for some `m`, then we have this second defining equation:

```
suc m * n = n + (m * n)
```


We just have to add n to the result of making a recursive call to multiply m and n . Notice that the recursive call is on the argument m , which is structurally smaller than the argument $\text{suc } m$ on the left-hand side of the equation. As noted above, this guarantees that all calls to multiplication will terminate.

Our definition is defining multiplication to be the same as iterated addition. To multiply 3 and 4, you just add 4 to 0 three times:

$$\begin{aligned}
 3 * 4 &= \\
 &= (\text{suc } (\text{suc } (\text{suc zero}))) * 4 = \\
 &= 4 + ((\text{suc } (\text{suc zero})) * 4) = \\
 &= 4 + 4 + ((\text{suc zero}) * 4) = \\
 &= 4 + 4 + 4 + (\text{zero} * 4) = \\
 &= 4 + 4 + 4 + \text{zero} = \\
 &= 12
 \end{aligned}$$

3.7 Some Basic Theorems about Multiplication

Like addition, multiplication is associative and commutative, and `nat-thms.agda` has proofs `*assoc` and `*comm` for these properties. Before we look at those proofs, let us first see proofs of some distributivity properties of multiplication over addition.

3.7.1 Right distributivity of multiplication over addition

Let us prove the following theorem relating multiplication and addition:

$$\forall (x \ y \ z : \mathbb{N}) \rightarrow (x + y) * z \equiv x * z + y * z$$

This is `*distribr` in `nat-thms.agda` in the IAL. Which of the three variables in question should we try to induct on? Our rule of thumb asks us to see how many times a variable is used as an argument on which pattern matching will be performed. The tallies are:

<i>variable</i>	<i>count</i>
x	2
y	1
z	0

This makes it pretty clear that the best choice for the variable on which to induct is x . One rather nice feature of Agda we can introduce at this point is the `emacs` command to split pattern variables automatically. Say we start out our definition of `*distribr` this way:

$$*distribr : \forall (x \ y \ z : \mathbb{N}) \rightarrow (x + y) * z \equiv x * z + y * z$$

```
*distribr x y z = ?
```

We could again comment out the current definition of `distribr` in `nat-thms.agda` to try these steps. If we load the file with Control-c Control-l and then do Control-c Control-, we will see

```
*distribr x y z = {! }0
```

Now here is the new `emacs` command: if you put your cursor in the hole (so in between the curly braces) and type Control-c Control-c, Agda will ask for “pattern variables to case”. We can enter one or more pattern variables, separated by spaces, and Agda will automatically construct equations with right-hand sides covering all possibilities for those variables. In this case, if we enter `x` as the sole pattern variable to split on, Agda will automatically turn the above equation into the following two:

```
*distribr zero y z = {! }0
*distribr (suc x) y z = {! }1
```

The old equation with pattern variable `x` has been replaced by two new ones with the two possible constructors for `x`, namely `zero` and `suc`.

The goal in the first equation simplifies to

```
y * z ≡ y * z
```

which we prove with `refl` as usual. If we type Control-c Control-comma with our cursor in the second hole, we see this remaining goal:

```
z + (x + y) * z ≡ z + x * z + y * z
```

Inserting some extra parentheses to make the syntax more explicit, this is

```
z + ((x + y) * z) ≡ (z + x * z) + y * z
```

We see that the induction hypothesis can be applied here, to transform $(x + y) * z$ into $(x * z) + (y * z)$. This is legal because we are using our induction hypothesis with `x` to prove the case with `suc x`. So let us use a `rewrite` with a recursive call to `*distribr`. The proof becomes:

```
*distribr : ∀ (x y z : ℕ) → (x + y) * z ≡ x * z + y * z
*distribr zero y z = {! }0
*distribr (suc x) y z rewrite *distribr x y z = {! }1
```

If we use Control-c Control-comma in the hole labeled 1, we will see this goal to prove:

```
z + (x * z + y * z) ≡ z + x * z + y * z
```

If we read through this carefully, on each side of the \equiv sign (for propositional equality), we have a sum of z , $x * z$, and $y * z$. The sums are just associated differently: on the left-hand side, the sum is right-associated, while on the right-hand side it is left-associated (since `_+_` is declared left-associative in `nat.agda`, Agda omits the parentheses which would make that explicit). We can reassociate

```

*distribr : ∀ (x y z : ℕ) → (x + y) * z ≡ x * z + y * z
*distribr zero y z = refl
*distribr (suc x) y z rewrite *distribr x y z =
  +assoc z (x * z) (y * z)

```

Figure 3.2: Proof of right-distributivity of multiplication over addition

the sum on the left-hand side by applying the `+assoc` proof we proved above. We just call this proof as we would call any function. In fact, we can simply use a call to `+assoc` to complete the proof, because the goal we have to prove is nothing more than an instance of associativity of addition. In other words, we have to prove

$$A + (B + C) \equiv A + B + C$$

where A is z , B is $x * z$, and C is $y * z$. And this is exactly what is proved by calling `+assoc` with those three expressions as inputs. So the completed proof of `distribr` is the one shown in Figure 3.2.

3.7.2 Commutativity of multiplication

This proof is very similar to the proof of commutativity of addition. As for addition, our rule of thumb for induction determines no choice of which of the two variables to induct on when proving

$$\forall (x y : \mathbb{N}) \rightarrow x * y \equiv y * x$$

So let us try x . The proof becomes:

```

*comm : ∀ (x y : ℕ) → x * y ≡ y * x
*comm zero y = { } 0
*comm (suc x) y = { } 1

```

Looking at hole 0 (by placing our cursor in the hole and typing `Control-c Control-comma`), we see that the goal to prove is:

```
Goal: 0 ≡ y * 0
```

```
-----
y : ℕ
```

This is similar to what happened when we proved commutativity for addition, and the solution is similarly easy. We need a helper lemma with the following typing, proved earlier in `nat-thms.agda`:

```
*0 : ∀ (x : ℕ) → x * 0 ≡ 0
```

We can just rewrite with a call to this lemma, and then use `refl` to finish that case. So the proof becomes:

```

*comm : ∀ (x y : ℕ) → x * y ≡ y * x
*comm zero y rewrite *0 y = refl
*comm (suc x) y rewrite *suc y x | *comm x y = refl

```

Figure 3.3: Proof of commutativity of multiplication

```

*comm : ∀ (x y : ℕ) → x * y ≡ y * x
*comm zero y rewrite *0 y = refl
*comm (suc x) y = { }0

```

If we look at the goal for the new hole 0 (since Agda rennumbers the holes when we reload the file), we will see it is:

```
Goal: y + x * y ≡ y * suc x
```

```
-----
y : ℕ
x : ℕ

```

Just as for commutativity of addition, we need a helper lemma to shift the `suc` off the second argument to multiplication on the right-hand side of this goal equation. The typing for this lemma, which can be found earlier in `nat-thms.agda`, is

```
*suc : ∀ (x y : ℕ) → x * (suc y) ≡ x + x * y
```

If we rewrite with a call to this, our new goal becomes

```
y + x * y ≡ y + y * x
```

And now we just need to make a recursive call to `*comm`, to turn `x * y` on the left-hand side of the equation into `y * x`. The complete proof of commutativity of multiplication is then in Figure 3.3.

3.7.3 Associativity of multiplication

Let us prove one more theorem about multiplication, namely associativity:

```
∀ (x y z : ℕ) → x * (y * z) ≡ (x * y) * z
```

Just as for associativity of addition, our rule of thumb provides us with a clear suggestion for which variable might be most helpful to induct on. For `x` is used twice as the first argument of multiplication (and multiplication as we have defined it does pattern matching on that first argument), while `y` is used once like that, and `z` zero times. So let us recurse on `x`. The proof starts out as:

```

*assoc : ∀ (x y z : ℕ) → x * (y * z) ≡ (x * y) * z
*assoc zero y z = ?
*assoc (suc x) y z = ?

```

```

*assoc : ∀ (x y z : ℕ) → x * (y * z) ≡ (x * y) * z
*assoc zero y z = refl
*assoc (suc x) y z
  rewrite *assoc x y z | *distribr y (x * y) z = refl

```

Figure 3.4: Proof of associativity of multiplication

The first case is trivial, because when x is instantiated to 0 , the goal simplifies to just $0 \equiv 0$, which we prove with `refl` as usual. So let us look (using `Control-c` `Control-comma` with our cursor in the hole) at the second goal we have to prove:

```
Goal: y * z + x * (y * z) ≡ (y + x * y) * z
```

```
-----
z : ℕ
y : ℕ
x : ℕ
```

This one has gotten a bit more complicated. We can see a place where we could apply our induction hypothesis (i.e., make a recursive call), on the left-hand side of the goal equation. We might as well just do that now, since we will surely want to do it, and it is not too clear yet what else we should do to this goal. If we do a `rewrite` with a recursive call to `*assoc`, our proof becomes

```

*assoc : ∀ (x y z : ℕ) → x * (y * z) ≡ (x * y) * z
*assoc zero y z = refl
*assoc (suc x) y z rewrite *assoc x y z = {! } 0

```

The goal for hole `0` is now:

```
y * z + x * y * z ≡ (y + x * y) * z
```

It is still not too clear what to do, but if we think about it a bit, we will hopefully notice that on the right-hand side, there is an expression we could simplify by applying the right-distributivity lemma we proved above (Section 3.7.1). In fact, if we do this, the two sides of our goal equation will become identical. This completes the proof, shown in Figure 3.4.

3.8 Arithmetic Comparison (Less-Than)

In `nat.agda` in the IAL, we have a definition of the usual less-than operation for arithmetic comparison. This is shown in Figure 3.5. The code has four equations, corresponding to the four basic possibilities for two natural-number inputs: either both are zero, both are nonzero, or we have one of each. In three of these cases, the boolean answer is determined with no further computation. Only if both numbers are nonzero is something more required: for `suc x` to be less than `suc y`, we

```

_<_ : ℕ → ℕ → ℬ
0 < 0 = ff
0 < (suc y) = tt
(suc x) < (suc y) = x < y
(suc x) < 0 = ff

```

Figure 3.5: A definition of the less-than function on natural numbers

must have x less than y .

Let us prove that $_<_$ is transitive: if x is less than y and y is less than z , then we must have x less than z . This is a basic property of the less-than function. We will see other examples involving transitive relations later in the book (for example, in Section 5.4.1). The proof, from `nat-thms.agda`, is shown in Figure 3.6. This proof is not difficult, but it does combine a lot of the proof techniques we have seen so far for Agda. The theorem, called `<-trans`, takes in implicit natural-number inputs x , y , and z . It then takes in two explicit arguments, which are proofs that $x < y$ is equal to `tt` and $y < z$ is also equal to `tt`. It then returns a proof that $x < z$ is equal to `tt`. You might wonder why we do not just write the following for this theorem:

$$\forall \{x \ y \ z : \mathbb{N}\} \rightarrow x < y \rightarrow y < z \rightarrow x < z$$

This looks more elegant, because we do not have the “ $\equiv \text{tt}$ ” everywhere we are using $_<_$. But this expression will not type check, because as defined in the IAL, $_<_$ returns a boolean (\mathbb{B}), and a boolean cannot be used as a type. Types have type `Set` in Agda, and \mathbb{B} is not the same as `Set`. One could alternatively define $_<_$ as a *relation*: it would take in two natural numbers and return a `Set`. But then less-than could not be used computationally in code. Expressions which have type `Set` describe code, but they are not code themselves (one cannot pattern-match, for example, on expressions of type `Set`, nor do anything else computational with them). The transitivity theorem we are considering is making a statement about the behavior of a particular program we have written, namely $_<_$. If that program returns `tt` when given x and y as inputs, and if it also returns `tt` when given y and z as inputs, then it will yet again return `tt` when given x and z as inputs.

Turning to the actual proof in Figure 3.6: we are pattern-matching on all three implicit inputs (x , y , and z). There are several impossible cases, which are handled with the absurd pattern `()`. For example, in the third defining clause, we the absurd pattern is used because the type of the second explicit input, namely $y < z \equiv \text{tt}$, has been instantiated with `suc y` for y (remember that these are two distinct variables named y) and `0` for z . So that type is

$$\text{suc } y < 0 \equiv \text{tt}$$

This type is equal to `ff` $\equiv \text{tt}$, by the definition of $_<_$ (the fourth equation in Figure 3.5 applies in this case, to simplify `suc y < 0` to `ff`). And as we saw previously (Section 2.4.1), we can use the absurd pattern `()` in such cases so that

```

<-trans : ∀ {x y z : ℕ} →
           x < y ≡ tt → y < z ≡ tt → x < z ≡ tt
<-trans {x} {0} p1 p2 rewrite <-0 x = B-contra p1
<-trans {0} {suc y} {0} p1 ()
<-trans {0} {suc y} {suc z} p1 p2 = refl
<-trans {suc x} {suc y} {0} p1 ()
<-trans {suc x} {suc y} {suc z} p1 p2 =
  <-trans {x} {y} {z} p1 p2

```

Figure 3.6: Proof of transitivity of less-than

Agda knows we will not define the value of the function in this case. The function could never be called with a proof of an impossible formula like $ff \equiv tt$, so no definition is needed.

The first equation in Figure 3.6 is a little more interesting. Here we are again in an impossible situation: we have a proof of $x < y \equiv tt$, where y has been instantiated to 0. So our first explicit argument is a proof of $x < 0 \equiv tt$, which is impossible. But in this case, the impossibility is not obvious enough for us to be able to use the absurd pattern. To use the absurd pattern for an equation $A \equiv B$, Agda requires that A and B are definitionally equal to values starting with distinct constructors, like tt and ff or $zero$ and suc . What the proof in Figure 3.6 does in this case is to do a `rewrite` with an application of the following easily proved lemma from `nat-thms.agda`:

```
<-0 : ∀ (x : ℕ) → x < 0 ≡ ff
```

This changes the type of `p1` from $x < 0 \equiv tt$ to $ff \equiv tt$. But if we change the type using `rewrite`, we cannot use the absurd pattern instead of `p1`. Agda does not allow this. Instead, we are giving `p1` as an input to the theorem `B-contra`, which we saw in Section 2.5:

```
B-contra : ff ≡ tt → ∀ {P : Set} → P
```

In this case, Agda is able to figure out what the implicit type P is which we want to inhabit. So the proof closes this case with the call to `B-contra`.

The most interesting case is when x , y , and z are all nonzero numbers $suc\ x$, $suc\ y$, and $suc\ z$ (for different values for x , y , and z). In that case we have

- $suc\ x < suc\ y \equiv tt$
- $suc\ y < suc\ z \equiv tt$

By the third equation from the definition of `_<_` (Figure 3.5), these types are definitionally equal to

- $x < y \equiv tt$
- $y < z \equiv tt$

This is great, because we can then just use our induction hypothesis (i.e., make a recursive call to `<-trans`). This is legal because each input is structurally less for this recursive call than the corresponding input to the call we are covering. And that concludes this case.

There remains one trivial case, when x is 0 and y and z are both nonzero. The goal equation becomes definitionally equal to $tt \equiv tt$, and can thus be proved by `refl`. That completes the proof of transitivity of `_<_`. In `nat-thms.agda`, you will find several variations of this theorem, using a less-than-or-equal-to function `_≤_`, defined in terms of `_<_` and an *equality test* `_=N_`, which we will consider next.

```

_≤_ : ℕ → ℕ → ℬ
x ≤ y = (x < y) || x =N y

```

3.8.1 Dotted variables

This is not a bad place to mention one feature of implicit arguments which we have not yet seen. For our proof of `<-trans`, we matched on the implicit arguments in the equations defining `<-trans`. But suppose we started out our proof without doing that:

```

<-trans : ∀ {x y z : ℕ} →
           x < y ≡ tt → y < z ≡ tt → x < z ≡ tt
<-trans p q = { } 0

```

If we look at the context in the hole, we will see this:

```
Goal: .x < .z ≡ tt
```

```

-----
q  : .y < .z ≡ tt
p  : .x < .y ≡ tt
.z : ℕ
.y : ℕ
.x : ℕ

```

Agda has introduced variables whose names start with dots, for the implicit arguments which we have not matched. Those variables cannot be used on the right-hand side of the equation. They are just there to allow expression of the types for the explicit variables, `p` and `q` in this case. If we find we do need to refer to those variables on the right-hand side of the equation (as we did in the proof of `<-trans` above), then the solution is simply to match on the implicit arguments in the pattern on the left-hand side. So if we instead have

```

<-trans : ∀ {x y z : ℕ} →
           x < y ≡ tt → y < z ≡ tt → x < z ≡ tt
<-trans{x}{y}{z} p q = { } 0

```



```

_=N_ : ℕ → ℕ → ℬ
0 =N 0 = tt
suc x =N suc y = x =N y
_ =N _ = ff

```

Figure 3.7: A function testing natural numbers for equality

then we will see the following context, which does not have dotted variables.

```
Goal: x < z ≡ tt
```

```

-----
q : y < z ≡ tt
p : x < y ≡ tt
z : ℕ
y : ℕ
x : ℕ

```

3.9 An Equality Test for Natural Numbers

So far in this book, we have considered Agda’s built-in definitional equality (see Section 2.1.2), and also the defined propositional equality \equiv . The former is used automatically by Agda during type checking to determine if expressions are equivalent, while the latter is used to express, as a type, the proposition that two values are equal. There are many more kinds of equality that we can consider in type theory. One which is important for verified programming is a function for testing whether or not two values are equal. We could call this **computational equality**, or an equality test. For natural numbers, the code for this equality test is given in Figure ???. There are three situations covered by the equations: either both numbers are zero, both are nonzero, or we have one of each. If both numbers are zero, then they are of course equal and we return tt . If we have $\text{suc } x$ and $\text{suc } y$, we recursively test x and y for equality. And in the other situation we return ff . Note that there we use the pattern $_$, which is just like a variable except that its value cannot be referenced on the right-hand side of the equation.

So, as you can confirm using Control-c Control-n, $3 =N 3$ will return tt , and $3 =N 4$ will return ff .

Now let us look at two theorems which show that this equality test we have defined returns tt for its natural-number inputs iff those inputs are propositionally equal. We could take this property as the definition of an equality test for a type A : a function f of type $A \rightarrow A \rightarrow \mathbb{B}$ is an equality test iff $f \ x \ y$ returns tt iff $x \equiv y$ is provable. Not every type A can have an equality test. For example, it seems there is no way to define an equality test for the type $\mathbb{N} \rightarrow \mathbb{N}$, because such a test would have to be able to compare functions for definitional equality,

```

=N-refl : ∀ (x : ℕ) → (x =ℕ x) ≡ tt
=N-refl 0 = refl
=N-refl (suc x) = (=N-refl x)

=N-to-≡ : ∀ {x y : ℕ} → x =ℕ y ≡ tt → x ≡ y
=N-to-≡ {0} {0} u = refl
=N-to-≡ {suc x} {0} ()
=N-to-≡ {0} {suc y} ()
=N-to-≡ {suc x} {suc y} u
  rewrite =N-to-≡ {x} {y} u = refl

=N-from-≡ : ∀ {x y : ℕ} → x ≡ y → x =ℕ y ≡ tt
=N-from-≡ {x} refl = =N-refl x

```

Figure 3.8: Proofs showing that the function we have defined is truly an equality test

and this seems impossible within the language. There is no way to consider different cases for what the functions could look like, for example. But for many inductive datatypes, it is possible to define equality tests.

Figure 3.8 shows three proofs about this function `=ℕ`. The first is an easy one showing that `=ℕ` is reflexive: for any `x` of type `ℕ` calling `=ℕ` on `x` and `x` will always return `tt`. This theorem will be used shortly. The proof `=N-to-≡` shows one direction of the property for `=ℕ` to be an equality test: if `=ℕ` says two numbers are equal, then they really are. This is a **soundness** property: the things indicated to be true really are true. The proof just considers the four basic cases for two natural number inputs. There is one trivial case when both are zero. There are two absurd cases when one is zero and the other is nonzero (these are absurd cases because the instantiated type `x =ℕ y ≡ tt` becomes definitionally equal to `ff ≡ tt`). And in the last equation, where both inputs are nonzero, we do a `rewrite` with a recursive call, to turn the goal into the trivial equation `suc y ≡ suc y`.

Finally, the third proof, `=N-from-≡`, is very easy. We just match on the equality proof to force Agda to realize that `x` and `y` must be definitionally equal. Then our goal becomes just

```
x =ℕ x ≡ tt
```

which we prove using `=N-refl`.

3.10 Conclusion

We have seen how to define the basic arithmetic functions addition and multiplication on unary (Peano) natural numbers, where a number like 2 is represented as `suc (suc zero)`. We also saw how to prove several basic theorems about addition, using induction. In Agda, a proof by induction is just a terminating recursive function, so writing proofs of somewhat interesting properties like commutativity of addition ($x + y \equiv y + x$) is just programming, albeit with fancier types. The files `nat.agda` and `nat-thms.agda` define several more basic arithmetic functions, with various lemmas and theorems about them.

3.11 Exercises

1. As usual, pick some theorems in `nat-thms.agda` whose proofs do not look too complicated, and reprove them yourself.
2. In `nat.agda`, greater-than and greater-than-or-equal-to orderings are defined in terms of less-than and less-than-or-equal-to. Prove versions of theorems like `<-trans` and `<+`, but modified to use `_>_` instead of `_<_`. This is mostly for practice in writing out formulas in Agda, since the proofs can be written just to invoke the theorems dealing with `_<_`.
3. For each of the following functions, which one statement describes best what the function `f` does? There is exactly one correct answer for each.

(a) $f : (n : \mathbb{N}) \rightarrow \mathbb{N}$
 $f \ 0 = 1$
 $f \ (\text{suc } x) = (\text{suc } x) * (f \ x)$

- i. Reverses a list
- ii. Multiplies input `n` by itself
- iii. Computes n^n
- iv. Computes factorial

v. Factors `n`

(b) $f' : \mathbb{N} \rightarrow \mathbb{B}$
 $f : (n : \mathbb{N}) \rightarrow \mathbb{B}$
 $f \ 0 = \text{tt}$
 $f \ (\text{suc } x) = f' \ x$
 $f' \ 0 = \text{ff}$
 $f' \ (\text{suc } x) = f \ x$

- i. Returns tt (boolean true) iff n is even.
- ii. Returns tt iff n is 0.
- iii. Returns tt iff n is odd.
- iv. Returns the length of n
- v. Computes primality of n

Chapter 4

Lists

So far, we have studied programming and proving with the booleans and the natural numbers. These are important datatypes, certainly, but they are not data structures: data designed for holding and organizing other data. Data structures are, of course, fundamental to programming and Computer Science, so it is high time we looked at programming and proving with some sort of data structure. In this chapter, we will study programming and proving with lists in Agda. Lists are a basic data structure in any programming language, but for functional programming languages – from LISP to ML to Haskell – they are really a programming cornerstone. This may be because they provide a simple way to aggregate a collection of elements (usually of the same type) into a single structure. Lists as we will study them in Agda are essentially the same as lists in OCaml, Haskell, and other functional programming languages.

4.1 The List Datatype and Type Parameters

The list datatype is declared in the file `list.agda` in the Iowa Agda Library:

```
data  $\mathbb{L}$  { $\ell$ } (A : Set  $\ell$ ) : Set  $\ell$  where
  [] :  $\mathbb{L}$  A
  _::_ : (x : A) (xs :  $\mathbb{L}$  A)  $\rightarrow$   $\mathbb{L}$  A
```

This is our first example of a parametrized datatype, so let us look at it carefully. The declaration is similar to that of \mathbb{B} (booleans, Section 1.1) and \mathbb{N} (natural numbers, Section 3.1). We have the `data` keyword, indicating that this is a datatype declaration, and then we have the name \mathbb{L} of the type we are declaring. Next comes something we did not have in our previous datatype declarations: parameters. This datatype \mathbb{L} is for lists of elements, where all elements have the same type. For example, we could have a list of booleans, or a list of naturals. Whatever the type of the elements is, we need to mention it in the type of the lists of such elements. So for a list of booleans, we will use the type $\mathbb{L} \ \mathbb{B}$, and for a list of naturals, we will use $\mathbb{L} \ \mathbb{N}$. To do this, we will view \mathbb{L} as a function which takes in a type and returns a type. It is a *type-level* function. Actually, we will allow \mathbb{L} to take in a type at any type level ℓ . It will return a type at that same level (this is required by Agda).

To express the idea that \mathbb{L} is a function which takes a type A at level ℓ as input and then produces a type at that same level, we are writing some **type parameters** following the name of the datatype:

```
data  $\mathbb{L}$  { $\ell$ } (A : Set  $\ell$ )
```

We are using curly braces for the type level ℓ since Agda will be able to infer the level when we write a type like $\mathbb{L} \ \mathbb{B}$. Then we use parentheses for type parameter A , since this argument to \mathbb{L} needs to be explicit (Agda will not generally be able to infer it). Following these parameters, we list the type for \mathbb{L} itself (assuming its parameters are given), and then the `where` keyword:

```
data  $\mathbb{L}$  { $\ell$ } (A : Set  $\ell$ ) : Set  $\ell$  where
```

One nice thing about using parameters in a datatype definition is that we can use those parameters in the types for the constructors. So for the first constructor, `[]`, we just write this declaration:

```
[] :  $\mathbb{L} \ A$ 
```

This constructor `[]` is for the empty list, which contains no elements. All lists get built from the empty list by applying the second constructor, which is called “cons” from its ancestor operation in LISP:

```
_::_ : (x : A) (xs :  $\mathbb{L} \ A$ ) →  $\mathbb{L} \ A$ 
```

This infix operator takes in x of type A and then xs of type $\mathbb{L} \ A$ and it returns a $\mathbb{L} \ A$. The idea is that we construct a bigger list $x :: xs$ from an element x of type A and then a sublist xs which contains the other elements (if any) of the list. The element x is often called the **head** of the new list, while xs is its **tail**. A little later in the file there is a syntax declaration making the `_::_` operation right-associative. So with these definitions, we can construct lists like:

```
1 :: 2 :: 3 :: []
tt :: tt :: ff :: ff :: []
```

The first list just conses the natural numbers 1, 2, and 3, in that order, onto the empty list. It has type $\mathbb{L} \ \mathbb{N}$. The second list contains two copies of `tt` and then two of `ff`. It has type $\mathbb{L} \ \mathbb{B}$. Again, the complete definition of the list datatype is:

```
data  $\mathbb{L}$  { $\ell$ } (A : Set  $\ell$ ) : Set  $\ell$  where
  [] :  $\mathbb{L} \ A$ 
  _::_ : (x : A) (xs :  $\mathbb{L} \ A$ ) →  $\mathbb{L} \ A$ 
```

One small note: if you ever have occasion to look at the standard library for Agda (not the Iowa Agda Library), be warned that the `cons` operator there is a single character that looks like a double colon. You enter this in `emacs` with `\ ::`.

4.2 Basic Operations on Lists

Let us look at how to implement some basic operations on lists. Remember that in Agda, all datatypes are immutable, so we are here working with lists that cannot be changed. If we want to insert an element into a list, for example, we will not do so by destructively modifying that list. Rather, we will build a new (immutable!) list which is just like the original one except that it has the inserted element at the desired position. The code examples we are considering below are all from `list.agda` in the Iowa Agda Library.

4.2.1 Length of a list

To compute the length of a list, we have to recurse through the list, adding one for each use of `cons (: :)` we see:

```
length : ∀{ℓ}{A : Set ℓ} → ℒ A → ℕ
length [] = 0
length (x :: xs) = suc (length xs)
```

If we examine the type given for `length`, we see that `length` takes in an implicit type-level ℓ and a type A , and then takes in one explicit argument, of type $\mathbb{L} A$, which is the list for which we should calculate the length. It will then return a natural number (\mathbb{N}) for the length of the list. The code is straightforward. We have one case for the empty list, and one case for a nonempty list $x :: xs$ with head x and tail xs . These two cases are expressed as two equations. In the first case, we just return 0. In the second, we are returning the successor of the length of the tail (xs). The length of the tail is obtained by making a recursive call (`length xs`). In effect, this code is going to translate a sequence of `cons`'s into a sequence of successors. For example, it will turn

```
10 :: 20 :: 9 :: 19 :: 8 :: []
into
suc (suc (suc (suc (suc zero))))
which Agda knows can also be written 5.
```

4.2.2 Appending two lists

One basic operation is appending two lists. We want to take two lists, like

```
1 :: 2 :: 3 :: []
and
```

```
3 :: 4 :: 5 :: 6 :: []
```

and append them to get

```
1 :: 2 :: 3 :: 3 :: 4 :: 5 :: 6 :: []
```

We just create a new list with all the elements in order from the first and then the second list. This is pure functional programming, as we were recalling above, so we do not destroy or modify either of the two input lists in the course of generating the output list. We will use the notation `l1 ++ l2` to append lists `l1` and `l2`. The `++` operator is also used in Haskell for appending two lists. Here is the Agda code for this append operator:

```
_++_ : ∀ {ℓ} {A : Set ℓ} → ℒ A → ℒ A → ℒ A
[]      ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

Append takes in an implicit type-level ℓ and type A , and then takes in two explicit arguments, which are the lists to append. Those lists have to hold the same type of elements, and so the input and output lists all have type $\mathbb{L} A$. The code then proceeds by recursion on the first explicit argument. So we are pattern matching on the first list being appended. Let us look at the first equation:

```
[] ++ ys = ys
```

This says that if the first list is empty (i.e., `[]`), and the second list is `ys`, then we just return `ys`. Here `ys` is just a pattern variable. Often in functional programming you will see code use a variable name like `xs` or `ys` to refer to a list (with the idea that it is a list “of x ’s”). Hopefully this equation makes sense: if the first list we are appending is empty, then we just return the second list, because there is nothing in the first list to add to it.

Now let us look at the second equation:

```
(x :: xs) ++ ys = x :: (xs ++ ys)
```

This says that if we are appending a list `x :: xs`, with head `x` and tail `xs`, to some other list `ys`, then the new list we create will have head `x`, and its tail will be whatever we get from the recursive call `xs ++ ys`. For an example of this code in action, let us step through an example, where I am underlining the expression which is simplified next by applying one of the two defining equations for `_++_`:

$$\begin{aligned}
 (1 :: 2 :: 3 :: []) ++ (3 :: 4 :: 5 :: 6 :: []) &= \\
 1 :: ((2 :: 3 :: []) ++ (3 :: 4 :: 5 :: 6 :: [])) &= \\
 1 :: 2 :: (((3 :: []) ++ (3 :: 4 :: 5 :: 6 :: []))) &= \\
 1 :: 2 :: 3 :: ([] ++ (3 :: 4 :: 5 :: 6 :: [])) &= \\
 1 :: 2 :: 3 :: 3 :: 4 :: 5 :: 6 :: [] &
 \end{aligned}$$

4.2.3 Mapping a function over a list

A very commonly used function for immutable lists is the `map` function. The idea with `map` is that we have some list of elements, and we want to create a new list by applying some function `f` to those elements in order. Suppose our function is `_+_ 10`. Recall from Section 3.2 in Chapter 3 that this is a partial application of the addition function to the number 10. Its type is $\mathbb{N} \rightarrow \mathbb{N}$; it is a function that is waiting for a second number, and then will return the sum of 10 and that number. If we map this `_+_ 10` function over the list `1 :: 2 :: 3 :: []`, we will get `11 :: 12 :: 13 :: []`; in other words, the list where 10 has been added to each of the elements of `1 :: 2 :: 3 :: []`. This can be tested in Agda by typing `Control-c Control-n` (to request normalization of a term), and then the call to `map`, which is

```
map (_+_ 10) (1 :: 2 :: 3 :: [])
```

Here is the Agda code for `map`:

```
map : ∀ {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'} → (A → B) → ℒ A → ℒ B
map f []           = []
map f (x :: xs) = f x :: map f xs
```

This definition is following the practice (which we will not have much occasion to capitalize on) of being generic in the type levels of `A` and `B`. So we first have implicit arguments `ℓ` and `ℓ'` for those levels. The type `A` is the type for the elements of the first list, over which we want to map our function. The type `B` is the return type of the function, and hence of the elements of the second list. In the introductory example above, `A` and `B` were the same, namely \mathbb{N} . But in general they could be different. These types `A` and `B` are also implicit arguments. The first type of an explicit argument is `A → B`. This corresponds to the first explicit argument to `map`, which we are having be the function that is being mapped over the list. Next is the type `ℒ A`, for the input list. And the return type of the function is `ℒ B`, for the result of mapping the function.

The code itself for `map` is then quite straightforward. There are two defining equations for `map`. The first says that if we are calling `map` with a function `f` and the empty list `[]`, then the result is just the empty list.

```
map f []           = []
```

This makes sense: we want to apply the function `f` to every element of the list, but there are no elements, so we just return the empty list.

The second defining equation is where the action is:

```
map f (x :: xs) = f x :: map f xs
```

The left-hand side applies when we are calling `map` with a function `f` and a non-empty list with head `x` and tail `xs`. Intuitively, we know that we should apply

f to x , since x is one of the elements of the list. Then we should recursively keep mapping f over the tail xs . This is exactly what the right-hand side of the equation does. Thanks to the parsing precedence for $_::_$, the right-hand side parses as

$$(f\ x) :: (map\ f\ xs)$$

So this is saying to cons $f\ x$ as the new head onto $map\ f\ xs$ as the new tail of the list which is being returned as the output list in this case.

For an example of `map` in action, let us step through the computation which maps $\sim_$ (boolean negation) over the list `tt :: ff :: tt :: []`:

$$\begin{aligned} & \text{map } \sim_ (tt :: ff :: tt :: []) = \\ & \frac{(\sim tt) :: (\text{map } \sim_ (ff :: tt :: []))}{ff :: (\text{map } \sim_ (ff :: tt :: []))} = \\ & \frac{ff :: (\sim ff) :: (\text{map } \sim_ (tt :: []))}{ff :: tt :: (\text{map } \sim_ (tt :: []))} = \\ & \frac{ff :: tt :: (\sim tt) :: (\text{map } \sim_ [])}{ff :: tt :: ff :: (\text{map } \sim_ [])} = \\ & ff :: tt :: ff :: [] \end{aligned}$$

4.2.4 Filtering a list

Suppose we want to filter out some elements of a list which do not pass a given test p . If the elements of the list have type A , then this test p can be given as a function of type $A \rightarrow \mathbb{B}$. Such a function is sometimes also called a **predicate**. If p returns `tt` for an element x , that means we want to keep x (not filter it out), and if it returns `ff`, then we do not want to keep x . Here is the definition of the `filter` function from `list.agda` in the IAL:

```
filter : ∀{ℓ}{A : Set ℓ} → (A → ℬ) → ℒ A → ℒ A
filter p [] = []
filter p (x :: xs) = let r = filter p xs in
                     if p x then x :: r else r
```

There are two defining equations here. For the first, just as for `map` above, if the input list is empty (`[]`), then the filtered list has to be empty, too. The second defining equation is more interesting. The input list is $x :: xs$. So the head (an element) is x , and the tail (rest of the list) is xs . We want to check if the test p returns `tt` or `ff` for x . In the former case, we are keeping x , and in the latter we are not. We do this check using the `if_then_else_` operator we defined in Section 1.6. There is one more feature to note. The code is using a `let`-term to make a local definition, so that we can avoid repeating the recursive call `filter p xs` in the two branches of the `if_then_else_` expression. Agda supports `let`-terms like this, to introduce a local variable (in this case r) with a particular definition, (`filter p xs`) in a particular local scope following the `in` keyword. Here, the

benefit is that we can write $x :: r$ and r in the `if_then_else_` term, instead of $x :: (\text{filter } p \text{ } xs)$ and `filter p xs`.

Suppose we have a function `is-even` of type $\mathbb{N} \rightarrow \mathbb{B}$, which returns `tt` if the input number is even, and `ff` otherwise. (Such a function is defined in `nat.agda` in the Iowa Agda Library.) Then we can filter a list like

```
1 :: 2 :: 3 :: 4 :: 5 :: 6 :: 7 :: []
```

using the term

```
filter is-even (1 :: 2 :: 3 :: 4 :: 5 :: 6 :: 7 :: [])
```

The result is `2 :: 4 :: 6 :: []`.

4.2.5 Removing an element from a list, and lambda abstractions

Using the `filter` function we just defined, we can define a `remove` function which removes all copies of an element a from a list. This is done just by filtering the list to retain only those elements which are different from a . The `remove` function must be given another function `eq` to tell whether an element in the list is equal to a or not. So the typing for the `remove` function is

```
remove : ∀{ℓ}{A : Set ℓ}(eq : A → A → ℬ)(a : A)(l : ℒ A) → ℒ A
```

How can we call `filter` with a function which takes in x and returns `tt` iff x does not equal a ? A nice solution is to use an anonymous function. ^{no need to type him} This is a function which can be created right in the middle of other code. It does not have to be declared at the top level of a file as functions in Agda usually must be. Since they are anonymous, they cannot be recursive, and hence are less powerful than top-level functions. Pattern-matching is allowed, but it is probably best to use them just for simple glue functions that do not need pattern matching. Anonymous functions are introduced with λ (“lambda”, Greek lowercase “l”). This notation and the associated theory of lambda calculus was introduced by the logician and proto computer scientist Alonzo Church [4]. To declare an anonymous function for our needs here with `remove`, we can write

```
λ x → ~ (eq a x)
```

This function is equivalent to `anon a`, if we have this top-level definition of a function `anon`:

```
anon : ∀{ℓ}{A : Set ℓ} → A → A → ℬ
anon a x = ~ (eq a x)
```

So the code for `remove` is then just:

```
remove : ∀{ℓ}{A : Set ℓ}(eq : A → A → ℬ)(a : A)(l : ℒ A) → ℒ A
remove eq a l = filter (λ x → ~ (eq a x)) l
```

```
data maybe {ℓ} (A : Set ℓ) : Set ℓ where
  just  : A → maybe A
  nothing : maybe A
```

Figure 4.1: The definition of the `maybe` type

```
nth : ∀{ℓ}{A : Set ℓ} → ℕ → ℒ A → maybe A
nth _ [] = nothing
nth 0 (x :: xs) = just x
nth (suc n) (x :: xs) = nth n xs
```

Figure 4.2: Code for selecting the n 'th element of a list, if there is one

4.2.6 Selecting the n 'th element from a list, and the `maybe` type

Suppose we wish to select the n 'th element of a list, where the head of the list is considered element 0. There is the issue that the list might not have n elements for the particular value of n given. In other languages, we might raise an exception in this case, or fail in some other way. In Agda, since every function must terminate on all inputs and the language does not have exceptions, we have to handle this situation differently. The standard approach for this in pure functional programming is to use a `maybe` type (in OCaml it is called `option` instead of `maybe`). Instead of just returning a value of type `A` for the n 'th element of a list of `As`, we will return a value of type `maybe A`. The definition of the `maybe` type, from `maybe.agda` in the IAL, is shown in Figure 4.1.

An element of type `maybe A` is either `just a`, where `a` has type `A`; or else it is `nothing`. If the value of n given to our selection function, which we will call `nth`, is too large, then we will return `nothing`. Otherwise, if there is indeed an n 'th element `a` of the given list, then we will return `just a`. The code for this is in Figure 4.2. The first equation says that if `nth` is called on the empty list `[]`, then the value `nothing` is returned. If the list is nonempty, then if the index given is 0, we return the head (this is the second equation); or else if the index is `suc n`, we recurse with `n` on the tail of the list.

4.2.7 Reversing a list (easy but inefficient way)

Let us look at two ways to implement reversing a list. The first way is straightforward to code but has worse than necessary asymptotic complexity (and for this reason, I am not including it in the Iowa Agda Library). The idea is to pull the head `h` off the front of a nonempty list `h :: t` and append it to the end of the reversed tail `t`. This can be written as follows, where I am using a function `[_]`, defined in `list.agda` in the IAL, to create a list with just one element, when given that

element as input:

```
sreverse : ∀{ℓ}{A : Set ℓ} → ℒ A → ℒ A
sreverse [] = []
sreverse (h :: t) = (sreverse t) ++ [ h ]
```

This implementation is simple, and it does correctly reverse input lists. Unfortunately, it takes time quadratic in the size of the input list. The reason is that each call to `_++_` takes time linear in the length of the first list. So if we are reversing a list like

```
1 :: 2 :: 3 :: 4 :: 5 :: []
```

then using `sreverse` (“slow reverse”), we will need to compute

```
((([ ] ++ [ 5 ]) ++ [ 4 ]) ++ [ 3 ]) ++ [ 2 ]) ++ [ 1 ]
```

The left-associated calls to `_++_` will take time $\sum_{i=1}^n i$, since each left-grouped call to `_++_` is computing a list of length i , where i ranges from 1 to n . And the above summation is well known to be $O(n^2)$. So while this version of reverse is easy to write (and also relatively easy to reason about), it is inefficient.

4.2.8 Efficiently reversing a list

There is a somewhat trickier way to reverse a list, that only requires time linear in the size of the input list. The basic idea is to use a helper function that will accumulate the reversed result as it recurses through the input list. One way to think about how this works is to imagine that the input list is a tower of blocks, with the head at the top. For example, the list `1 :: 2 :: 3 :: 4 :: []` can be viewed this way:

```
1
2
3
4
```

Now to reverse this list, just move blocks from the top (the head of the list) to the top of a second list. So we move the head off the above tower and obtain:

```
2
3
4  1
```

Continuing, we get:

```
3  2
4  1
```

In the next step:

```

      3
      2
4    1

```

And finally:

```

      4
      3
      2
      1

```

This can be done with the following Agda function (in `list.agda`):

```

reverse-helper : ∀ {ℓ}{A : Set ℓ} → (ℕ A) → (ℕ A) → ℕ A
reverse-helper h [] = h
reverse-helper h (x :: xs) = reverse-helper (x :: h) xs

```

Here, the list which is accumulating the reversal is the first explicit argument `h` to `reverse-helper`. In the first defining equation we just return `h`. In the second, we remove the head `x` from the nonempty input list `x :: xs` and place it at the head of `h`. So in our recursive call we have `reverse-helper (x :: h) xs`; we have moved `x` from being on `xs` to being on `h`. To reverse a list, we must just call this helper function with the accumulator initialized to the empty list:

```

reverse : ∀ {ℓ}{A : Set ℓ} → ℕ A → ℕ A
reverse l = reverse-helper [] l

```

4.3 Reasoning about List Operations

The following theorems about the list operations we defined above can be found in `list-thms.agda` in the Iowa Agda Library.

4.3.1 Distributing length over append

Here is a simple theorem we can prove involving both the `length` function and list `append` (`_++_`):

```

length-++ : ∀{ℓ}{A : Set ℓ}(l1 l2 : ℕ A) →
  length (l1 ++ l2) ≡ (length l1) + (length l2)
length-++ [] l2 = refl
length-++ (h :: t) l2 rewrite length-++ t l2 = refl

```

This theorem says that if you append two lists `l1` and `l2` and then get the length of the result (`l1 ++ l2`), the number you obtain is the sum of the lengths of the

two lists. We do have a choice of which list to split on. How do we know to case split on `l1` instead of `l2`? We are using the basic rule of thumb (introduced in Section 3.4) for choosing variables to induct on in proofs: choose the variable that will trigger the most simplification using the defining equations for the operators involved, by choosing the variable that is going to get matched on the most in the function calls in the theorem. Here, both `l1` and `l2` are used in function calls that will pattern match on them: `l2` is used as an argument to `length`, while `l1` is used as an argument to `length` and also the first argument to `_++_`. Since `append (_++_)` matches on its first argument, that gives the edge to `l1` over `l2` for matching, since `l1` is used in two such positions, while `l2` is used in only one.

In the first equation defining `length-++`, if you replace `refl` by a hole (`?`), you can use `Control-c Control-`, (comma) in the hole (after loading the file with `Control-c Control-l`) to see that the goal has normalized to just

```
length l2 ≡ length l2
```

This is because when `l1` is `[]` (the empty list), the two sides of the equation we are trying to prove both simplify to `length l2`. So we can fill in that hole with just `refl`.

If you replace the second defining equation with just

```
length-++ (h :: t) l2 = ?
```

and use `Control-c Control-`, (again, after loading the file with `Control-c Control-l`), you will see that the normalized goal is

```
suc (length (t ++ l2)) ≡ suc (length t + length l2)
```

This looks promising. We just need to use our inductive hypothesis to transform `length (t ++ l2)` into `length t + length l2`, and then the two sides of the equation will be identical and we can conclude the case with `refl`. This is what the `rewrite` directive in the defining equation as listed at the start of this subsection does:

```
length-++ (h :: t) l2 rewrite length-++ t l2 = refl
```

Recursively calling `length-++` with arguments `t` and `l2` gives us a term

```
length-++ t l2
```

of type

```
length (t ++ l2) ≡ length t + length l2
```

And then the `rewrite` directive tells Agda to replace any occurrences of

```
length (t ++ l2)
```

in the goal with `length t + length l2`. This turns the goal into

```
suc (length t + length l2) ≡ suc (length t + length l2)
```

which can now be proved with `refl`.

4.3.2 Associativity of list append

Let us prove a basic property of the `++` operation: associativity. We proved associativity of addition in Section 3.4, and associativity of list append is quite similar. Here is the proof, from `list-thms.agda`:

```

++-assoc : ∀ {ℓ}{A : Set ℓ}(l1 : ℒ A)(l2 : ℒ A)(l3 : ℒ A) →
            (l1 ++ l2) ++ l3 ≡ l1 ++ (l2 ++ l3)
++-assoc [] l2 l3 = refl
++-assoc (x :: xs) l2 l3 rewrite +-assoc xs l2 l3 = refl

```

Let us first look carefully at the statement of the theorem

```

∀ {ℓ}{A : Set ℓ}(l1 : ℒ A)(l2 : ℒ A)(l3 : ℒ A) →
  (l1 ++ l2) ++ l3 ≡ l1 ++ (l2 ++ l3)

```

This is a bit of a mouthful. We are quantifying over three lists, `l1`, `l2`, and `l3`. Each list has the same type, namely $\mathbb{L} \ A$. Our type cannot contain any undeclared variables, so we also have to quantify over `A` of type `Set ℓ`, and then also the `ℓ`. This covers the first line of the type. The second line is the meat of what we are trying to prove:

```
(l1 ++ l2) ++ l3 ≡ l1 ++ (l2 ++ l3)
```

The left-hand side describes the list you get if you first append `l1` and `l2`, and then append `l3` to the result. The right-hand side describes the list you get if you append `l1` to what you get by appending `l2` and `l3`. In other words, just like associativity of addition, the order in which you apply the binary (two-argument) operator to combine three arguments does not matter.

The proof of this theorem is very short and easy, consisting of these two defining equations:

```

++-assoc [] l2 l3 = refl
++-assoc (x :: xs) l2 l3 rewrite +-assoc xs l2 l3 = refl

```

The key point to note is that we are pattern-matching on the first list `l1`, and considering two cases. Either this list is empty, in which case the pattern is `[]`, the constructor for the empty list. Or else the list is built with the `::` (“cons”) operator. In this case, it has a head `x` and a tail `xs`. In the first case, simplification is enough to reduce both sides of the equation to just `l2 ++ l3`. In the second case, simplification gets us to a point where just rewriting with the induction hypothesis is enough to finish the proof.

4.3.3 Length of filtered lists, and the `with` construct

Here is another theorem from `list-thms.agda`, which will allow us to demonstrate a useful feature of Agda that we have not encountered up until now: the


```

length-filter : ∀{ℓ}{A : Set ℓ}(p : A → ℬ)(l : ℒ A) →
    length (filter p l) ≤ length l ≡ tt
length-filter p [] = refl
length-filter p (x :: l) with p x
length-filter p (x :: l) | tt = length-filter p l
length-filter p (x :: l) | ff =
    ≤-trans{length (filter p l)}
    (length-filter p l) (≤-suc (length l))

```

Figure 4.3: The proof that filtering results in a list no longer than the original one

with `construct`. The theorem we will prove is

```

∀{ℓ}{A : Set ℓ}(p : A → ℬ)(l : ℒ A) →
    length (filter p l) ≤ length l ≡ tt

```

Reading this through in English, it says that for any type A (of any type level ℓ), and any predicate p on A , and any list l of elements of type A ; the length of the list you get by filtering l with p is less than or equal to the length of l . Intuitively, this is true, because filtering with p may discard some elements from the list, but it will not add any new elements to the list. So the length of the list could stay the same (if the predicate p returns `tt` for all elements of the list), but it might decrease (if there is even one element of the list where p returns `ff`). The \leq function is defined in `nat.agda`, and a number of theorems about it can be found in `nat-thms.agda`. We will need a couple of those theorems for our proof.

Let us look now at the proof of this theorem, which is called `length-filter`, from `list-thms.agda`. The proof is in Figure 4.3. The proof is case-splitting on the form of the input list l . If l is the empty list (`[]`), then the goal simplifies quite a bit. Let us see this step by step:

<code>length (filter p l) ≤ length l ≡ tt</code>	is instantiated to
<code>length (filter p []) ≤ length [] ≡ tt</code>	is definitionally equal to
<code>length [] ≤ length [] ≡ tt</code>	is definitionally equal to
<code>0 ≤ 0 ≡ tt</code>	is definitionally equal to
<code>tt ≡ tt</code>	

The definitional equalities all follow from the definitions of the functions involved (`filter`, `length`, and `≤`). So the simplified goal is `tt ≡ tt` (you can see this if you replace the right-hand side of the first defining equation for `length-filter` with a hole `?`, load the file with `Control-c Control-l`, and then enter `Control-c Control-comma` with your cursor in the hole). And `refl` proves this.

Now let us look at the simplified goal in the other case, by asking Agda to process this modified definition of `length-filter`:

```

length-filter : ∀{ℓ}{A : Set ℓ}(p : A → ℬ)(l : ℒ A) →
    length (filter p l) ≤ length l ≡ tt

```

```
length-filter p [] = refl
length-filter p (x :: l) = ?
```

If you load the file containing this modified definition with Control-c Control-l, and then do Control-c Control-comma with your cursor in the hole on the right-hand side of the second equation, you will see the following rather unwieldy goal from Agda:

```
length (if p x then x :: filter p l else filter p l) <
suc (length l)
||
length (if p x then x :: filter p l else filter p l) =N
suc (length l)
≡ tt
```

This actually looks worse than it needs to, because Agda has normalized the call to `_≤_`, using this definition from `nat.agda`, which defines less than or equals (`≤`) in terms of less than (`<`) and our computational equality on natural numbers (`=N`):

```
_≤_ : ℕ → ℕ → ℬ
x ≤ y = (x < y) || x =N y
```

If we (mentally) reverse that normalization step, we will see that the goal is really:

```
length (if p x then x :: filter p l else filter p l)
  ≤ suc (length l) ≡ tt
```

This is still a little intimidating, but not as bad as what Agda printed out for us.

How did this goal arise? Well, our starting goal for this case, when `l` is a nonempty list `x :: l` (for some other list `l`) is

```
length (filter p (x :: l)) ≤ length (x :: l) ≡ tt
```

Using definitional equality, Agda has normalized this goal to

```
length (if p x then x :: filter p l else filter p l)
  ≤ suc (length l) ≡ tt
```

This is because `filter` uses an `if_then_else_` expression to decide whether or not to include `x` in the list it is returning, based on whether `p x` returns `tt` or `ff`. And `length (x :: l)` is defined to be `suc (length l)`.

So how do we make progress on this goal? We would like to apply our induction hypothesis (as this is usually the key to getting one of these proofs to go through), but there is no obvious place to use that fact, which is

```
length (filter p l) ≤ length l ≡ tt
```

What we need to do is consider different cases, based on whether `p x` evaluates to `tt` or to `ff`. This is because the expression

```
length (if p x then x :: filter p l else filter p l)
```

is going to simplify differently, depending on the value of $p\ x$.

How do we do a case split like this, on something which is not an input to the function? That is where the ~~with-construct~~ of Agda comes in. Agda provides a way for us to extend the pattern we are using in the left-hand side of an equation, by pattern matching on an additional term. For `length-filter` (Figure 4.3), this is done like this (where let us just put holes for the resulting two right-hand sides):

```
length-filter p (x :: l) with p x
length-filter p (x :: l) | tt = ?
length-filter p (x :: l) | ff = ?
```

We have extended the left-hand side of our equation by writing `with p x`, to tell Agda that we wish to pattern match on the value of $p\ x$. Then we write two new equations, where we are separating our original pattern, which is

```
length-filter p (x :: l)
```

from the new patterns `tt` (in the first equation) and `ff` (in the second) with a bar.

If we ask Agda what the type is for the first hole, it will show us that the goal has now normalized to

```
length (filter p l) ≤ length l ≡ tt
```

This is because

```
length (if tt then x :: filter p l else filter p l)
  ≤ suc (length l) ≡ tt
```

is definitionally equal to

```
length (x :: filter p l) ≤ suc (length l) ≡ tt
```

which is in turn definitionally equal to

```
suc (length (filter p l)) ≤ suc (length l) ≡ tt
```

by the definition of `length`. The definition of `≤` actually strips off the two `suc`'s, leaving us with

```
length (filter p l) ≤ length l ≡ tt
```

This is great, because that is exactly the induction hypothesis, which we obtain by making a recursive call to `length-filter`. This is why the proof in Figure 4.3 has the following for this equation:

```
length-filter p (x :: l) | tt = length-filter p l
```

Now let us look at the goal for the second equation, where $p\ x$ is `ff`.

```
length (filter p l) ≤ suc (length l) ≡ tt
```

We can again use our induction hypothesis, to obtain

```
length (filter p l) ≤ length l ≡ tt
```

And in `nat-thms.agda`, we can find (with some looking!) that there is a theorem called `≤-suc` proving that

```
length l ≤ suc (length l) ≡ tt
```

So these two facts really look like $x \leq y$ and $y \leq z$, where

- x is `length (filter p l)`,
- y is `length l`, and
- z is `suc (length l)`.

Similar to the theorem `<-trans` we saw in Section 3.8, there is a theorem proving transitivity of less-than-or-equal-to, in `nat-thms.agda`:

```
≤-trans : ∀ {x y z : ℕ} →
  x ≤ y ≡ tt →
  y ≤ z ≡ tt →
  x ≤ z ≡ tt
```

The x , y , and z are implicit arguments, but unfortunately in this case, Agda has trouble inferring the value of x . If we just give the two explicit arguments, Agda will highlight the call to `≤-trans` in yellow, to tell us it cannot figure out one of the implicit arguments. As often seems to work in Agda, if we explicitly tell Agda the value of the first implicit argument (x), the values of the others can be inferred automatically. This is what we are doing in the last equation for `length-filter`, from Figure 4.3:

```
length-filter p (x :: l) | ff =
  ≤-trans{length (filter p l)}
    (length-filter p l)
    (≤-suc (length l))
```

We are specifying the value of x with `{length (filter p l)}`, and then passing the two proofs of our two inequalities. This completes the proof `length-filter`.

4.3.4 Filter is idempotent, and the keep idiom

Let us look at another theorem about the `filter` function:

```
∀{ℓ}{A : Set ℓ} (p : A → ℬ) (l : ℒ A) →
  (filter p (filter p l)) ≡ (filter p l)
```

This says that filtering a list twice using a predicate p gives the same result as filtering it once. In other words, the second call to `filter` has no effect after the first call. Intuitively, this is because the first call has already completed the work of dropping elements from the list which do not satisfy the predicate p . In a pure functional language like Agda, where there are no implicit side effects, if an element of the list satisfied p in the first call to `filter`, it must still do so in the

```

filter-idem : ∀{ℓ}{A : Set ℓ}(p : A → ℬ)(l : ℒ A) →
              (filter p (filter p l)) ≡ (filter p l)
filter-idem p [] = refl
filter-idem p (x :: l) with keep (p x)
filter-idem p (x :: l) | tt , p'
  rewrite p' | p' | filter-idem p l = refl
filter-idem p (x :: l) | ff , p' rewrite p' = filter-idem p l

```

Figure 4.4: Proof that filtering is idempotent

second call, because calling `p` on the same input must always produce the same output (this would not necessarily be true if `p` could use some hidden state).

The proof of this theorem is in Figure 4.4, and it is worth looking at for its use of a particular theorem-proving idiom in Agda which I call `keep`. In the Agda standard library, this is called `inspect`. To see why this is needed, let us step through the case when the input list is `x :: l`. If we look at the goal in that case (by doing Control-c Control-comma in a hole on the right-hand side of an equation with pattern `filter-idem p (x :: l)`), we will see this:

```

filter p (if p x then x :: filter p l else filter p l) ≡
(if p x then x :: filter p l else filter p l)

```

This resembles what we saw in the proof about `filter` from the previous section. We have some rather bulky expressions, with embedded `if_then_else_` expressions, where the `if` part is `p x`. So, following the approach of our previous proof about `filter`, we should case-split on `p x` using a `with` expression:

```

filter-idem p (x :: l) with p x
filter-idem p (x :: l) | tt = ?
filter-idem p (x :: l) | ff = ?

```

Now let us look at the goal for the equation where `p x` is `tt`. That goal is:

```

(if p x then x :: filter p (filter p l) else
  filter p (filter p l)) ≡
x :: filter p l

```

This does not look good. We got rid of some of the `if_then_else_` expressions scrutinizing `p x`, but there is still one there! What happened? Let's look at how Agda has simplified the starting goal to get to this point. Instantiating `p x` with `tt`, the goal is:

```

filter p (if tt then x :: filter p l else filter p l) ≡
(if tt then x :: filter p l else filter p l)

```

By the definition of `if_then_else_`, this can be simplified to

```

filter p (x :: filter p l) ≡
x :: filter p l

```

But now we may apply the definition of `filter` to simplify the left-hand side of this equation further, to

```
(if p x then x :: filter p (filter p l) else
  filter p (filter p l)) ≡
x :: filter p l
```

This is really irritating! We are in a case where `p x` is `tt`, but Agda only applied that fact to the goal the first time. It is not applying it again. This is actually due to the way that Agda implements `with`. All with gets to do is instantiate an expression once. If subsequent normalization steps (using definitional equality) produce that expression again, Agda will not instantiate it again.

The `keep` (or as more standardly known, `inspect`) idiom is a cute way around this. Instead of doing a `with` on `p x`, we just do a `with` on `keep (p x)`. Then in the subsequent equations, we will have available an extra variable (here `p'`) saying that the expression we are doing the `keep` on – here, `p x` – is equal to the pattern we have written for it. In more detail, let us look at the type of `p'` in the contexts Agda shows us for each of the following two holes:

```
filter-idem p (x :: l) with keep (p x)
filter-idem p (x :: l) | tt , p' = ?
filter-idem p (x :: l) | ff , p' = ?
```

In the context of the first hole, we have:

```
p' : p x ≡ tt
```

In the second, it is

```
p' : p x ≡ ff
```

Agda will not actually instantiate `p x` in the goal when we do a `keep`. We have to apply an explicit `rewrite p'` to change `p x` into `tt` for the first hole, and into `ff` for the second:

```
filter-idem p (x :: l) with keep (p x)
filter-idem p (x :: l) | tt , p' rewrite p' = ?
filter-idem p (x :: l) | ff , p' rewrite p' = ?
```

If we look at the goal for the first hole now, we will see it is

```
(if p x then x :: filter p (filter p l) else
  filter p (filter p l))
≡ x :: filter p l
```

This is the same normalized goal we had gotten above. So it looks like we have not made any progress over just using `with` without `keep`. But we have made progress, because we still have `p' : p x ≡ tt` in our context. We can do a second `rewrite p'`, to eliminate this second occurrence of `p x` which has shown up after normalization. To do multiple rewrites in Agda, one just separates them with a bar. The rewrites are then performed sequentially. So we have:

```

filter-idem p (x :: l) with keep (p x)
filter-idem p (x :: l) | tt , p' rewrite p' | p' = ?
filter-idem p (x :: l) | ff , p' rewrite p' = ?

```

And now Agda shows us that the first goal is:

```

x :: filter p (filter p l) ≡ x :: filter p l

```

That is fantastic for us, because now we can just do a rewrite with the induction hypothesis to get a trivial equation:

```

filter-idem p (x :: l) with keep (p x)
filter-idem p (x :: l) | tt , p'
  rewrite p' | p' | filter-idem p l = refl
filter-idem p (x :: l) | ff , p' rewrite p' = ?

```

The second case, where $p\ x$ is `ff`, is handled more simply. In fact, it is just an application of the induction hypothesis:

```

filter-idem p (x :: l) with keep (p x)
filter-idem p (x :: l) | tt , p'
  rewrite p' | p' | filter-idem p l = refl
filter-idem p (x :: l) | ff , p' rewrite p' = filter-idem p l

```

This completes the proof of `filter-idem`. A last note about the `keep` idiom: in the Iowa Agda Library, this is implemented using dependent pairs, which are explained in Section 5.3.

4.3.5 Reverse preserves list length

Let us prove a theorem about our efficient list-reversal function (Section 4.2.8) above: the length of the reverse of a list is the same as the length of the original list. In Agda, this theorem is stated:

$$\forall \{\ell\} \{A : \text{Set } \ell\} (l : \mathbb{L} A) \rightarrow \text{length } (\text{reverse } l) \equiv \text{length } l$$

The theorem uses an implicit quantification over ℓ and A , as we have seen above, and then an explicit quantification for the list l that is being reversed.

Now, to prove anything interesting about `reverse`, we are going to have to reason about `reverse-helper`, which `reverse` calls. As discussed above, `reverse-helper` takes two arguments: the first is the reverse of the list which `reverse-helper` has already processed, and the second is the rest of the list still to be reversed. It can be rather tricky to figure out what general property of `reverse-helper` we should prove in order to help us prove a particular theorem about `reverse`. Fortunately, here it is not too hard to realize that the length of `reverse-helper h l` is going to be the sum of the lengths of h and l . So we first will prove this lemma:

$$\forall \{\ell\} \{A : \text{Set } \ell\} (h\ l : \mathbb{L} A) \rightarrow \text{length } (\text{reverse-helper } h\ l) \equiv \text{length } h + \text{length } l$$

There are two arguments here which we might try pattern matching on: `h` and `l`. Which should we choose? Using our basic rule of thumb (see Section 4.3.1 above), we should choose `l`, because it is the argument on which `reverse-helper` pattern matches (`h` is not analyzed using pattern matching by `reverse-helper`). So we start our proof this way:

```
length-reverse-helper :
  ∀{ℓ}{A : Set ℓ} (h l : ℒ A) →
    length (reverse-helper h l) ≡ length h + length l
length-reverse-helper h [] = ?
length-reverse-helper h (x :: xs) = ?
```

Let us try to fill in the first hole, for the case when `l` is `[]`, first. In that case, the normalized goal (which we see by typing `Control-c Control-`, with our cursor in the hole, after first loading the file with `Control-c Control-l`) is

```
length h ≡ length h + 0
```

That looks pretty promising, and indeed, we can finish this case off if we rewrite using the `+0` lemma from `nat-thms.agda`, which says:

```
+0 : ∀ (x : ℕ) → x + 0 ≡ x
```

So our proof of `length-reverse-helper` becomes:

```
length-reverse-helper :
  ∀{ℓ}{A : Set ℓ} (h l : ℒ A) →
    length (reverse-helper h l) ≡ length h + length l
length-reverse-helper h [] rewrite +0 (length h) = refl
length-reverse-helper h (x :: xs) = ?
```

Let us fill in the second hole, which we can see (with `Control-c Control-`, again) has type

```
length (reverse-helper (x :: h) xs) ≡ length h + suc (length xs)
```

We can apply our induction hypothesis to transform

```
length (reverse-helper (x :: h) xs)
```

into

```
length (x :: h) + length xs
```

This would clearly be progress, so let us do it. The proof becomes

```
length-reverse-helper :
  ∀{ℓ}{A : Set ℓ} (h l : ℒ A) →
    length (reverse-helper h l) ≡ length h + length l
length-reverse-helper h [] rewrite +0 (length h) = refl
length-reverse-helper h (x :: xs)
  rewrite length-reverse-helper (x :: h) xs = ?
```

The remaining hole has the following normalized type:


```
suc (length h + length xs) ≡ length h + suc (length xs)
```

This is looking very close, and indeed we have a lemma in `nat-thms.agda` that will help us complete this case. The lemma is

```
+suc : ∀ (x y : ℕ) → x + (suc y) ≡ suc(x + y)
```

This lets us pull out a `suc` from the right argument of a plus-term. In fact, if we flip a suitable instantiation of this equation around using `sym`, we will get exactly what we need to prove our goal. So the final proof is

```
length-reverse-helper :
  ∀{ℓ}{A : Set ℓ}(h l : ℒ A) →
    length (reverse-helper h l) ≡ length h + length l
length-reverse-helper h [] rewrite +0 (length h) = refl
length-reverse-helper h (x :: xs)
  rewrite length-reverse-helper (x :: h) xs =
    sym (+suc (length h) (length xs))
```

And using this helper lemma, we can then easily prove the theorem we desired about the length of the reverse of a list:

```
length-reverse : ∀{ℓ}{A : Set ℓ}(l : ℒ A) →
  length (reverse l) ≡ length l
length-reverse l = length-reverse-helper [] l
```

4.4 Conclusion

We have seen the list datatype \mathbb{L} , which is a central data structure in much functional programming. We looked at basic operations like appending two lists (`_++_`); computing the length of a list; mapping a function over a list to create a new list consisting of the outputs of the function on the list elements; filtering a list to drop all elements which are not accepted by some predicate (a function which returns a boolean when given an element of the type stored in the list); selecting the n 'th element out of a list; and efficiently reversing a list. We then saw a number of interesting basic theorems about these operations. Some of these proofs were more involved than the ones we considered in the previous chapter, for natural numbers. We saw several useful theorem-proving features: the `with` construct, which allows us to case-split on the value of some term which does not have to be an input to the function (unlike in pattern matching without `with`); and the `keep` idiom (also known as `inspect`), where not only can we perform a case-split on some intermediate term, but we get to keep an equation in the context which reflects that case-split. Such equations can be necessary, in order to perform subsequent reasoning based on the fact that the scrutinized term is equal to the particular value for that case alternative.

4.5 Exercises

1. Which of the following formulas are theorems (i.e., they can be proved) about the list operations we have considered in this chapter?

$$(a) \forall \{\ell\} \{A : \text{Set } \ell\} (\ell_1 \ell_2 : \mathbb{L} A) \rightarrow \\ \ell_1 ++ \ell_2 \equiv \ell_2 ++ \ell_1$$

$$(b) \forall \{\ell \ell'\} \{A : \text{Set } \ell\} \{B : \text{Set } \ell'\} \\ (f : A \rightarrow B) (\ell : \mathbb{L} A) \rightarrow \\ \text{length } (\text{map } f \ell) \equiv \text{succ } (\text{length } \ell)$$

- (c) The `repeat` function takes a number `n` and an element `a`, and constructs a list of length `n` where all elements are just `a`. So `repeat 3 tt` evaluates to `tt :: tt :: tt :: []`.

$$\forall \{\ell\} \{A : \text{Set } \ell\} \{p : A \rightarrow \mathbb{B}\} \{a : A\} (n : \mathbb{N}) \rightarrow \\ p \ a \equiv \text{ff} \rightarrow \\ \text{filter } p (\text{repeat } n \ a) \equiv []$$

- (d) The `is-empty` function returns `tt` if the list it is given is `[]` (empty list), and `ff` otherwise.

$$\forall \{\ell\} \{A : \text{Set } \ell\} (\ell : \mathbb{L} A) \rightarrow \\ \text{is-empty } \ell \equiv \text{tt} \rightarrow \\ \text{is-empty } (\text{reverse } \ell) \equiv \text{ff}$$

$$(e) \forall \{\ell\} \{A : \text{Set } \ell\} (p : A \rightarrow \mathbb{B}) (\ell_1 \ell_2 : \mathbb{L} A) \rightarrow \\ \text{filter } p (\ell_1 ++ \ell_2) \equiv \text{filter } p \ell_1 ++ \text{filter } p \ell_2$$

2. Indicate which of the typings listed for each test term would be accepted by Agda (more than one typing could be accepted for the same term, due to implicit arguments).

$$(a) \text{test} = []$$

$$\text{i. test} : \mathbb{L} \mathbb{N}$$

$$\text{ii. test} : \mathbb{L} \mathbb{L}$$

$$\text{iii. test} : \mathbb{L} \mathbb{B}$$

$$\text{iv. test} : \mathbb{L} (\mathbb{L} \mathbb{B})$$

$$\text{v. test} : \mathbb{L} \text{Set}$$

$$(b) \text{test } [] = 0 \\ \text{test } (x :: xs) = \text{succ } (\text{test } xs)$$

i. $\text{test} : \forall \{\ell\} \{A : \text{Set } \ell\} \rightarrow \mathbb{L} A \rightarrow \mathbb{N}$

ii. $\text{test} : \mathbb{L} A \rightarrow \mathbb{N}$

iii. $\text{test} : \mathbb{L} B \rightarrow \mathbb{N}$

iv. $\text{test} : \forall \{\ell\} \{A : \text{Set } \ell\} \rightarrow (xs : \mathbb{L} A) \rightarrow \text{length } xs$

(c) $\text{test } f \ g \ x = \text{map } g \ (\text{map } f \ x)$

i. $\text{test} : \forall \{A \ B \ C : \text{Set}\} \rightarrow (A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow \mathbb{L} A \rightarrow \mathbb{L} C$

ii. $\text{test} : \forall \{\ell\} \{A : \text{Set } \ell\} \rightarrow (A \rightarrow A) \rightarrow (A \rightarrow A) \rightarrow \mathbb{L} A \rightarrow \mathbb{L} A$

iii. $\text{test} : \forall \{A \ B \ C : \text{Set}\} \rightarrow (B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow \mathbb{L} A \rightarrow \mathbb{L} C$

iv. $\text{test} : \forall \{\ell\} \{A \ B \ C : \text{Set } \ell\} \rightarrow (A \rightarrow B \rightarrow C) \rightarrow \mathbb{L} A \rightarrow \mathbb{L} (B \rightarrow C)$

v. $\text{test} : \forall \{B : \text{Set}\} \{A : \mathbb{L} B\} \rightarrow (A \rightarrow B) \rightarrow (B \rightarrow B) \rightarrow \mathbb{L} A \rightarrow \mathbb{L} B$

3. Define a polymorphic function `takeWhile` which takes in a predicate on type `A` (i.e., a function of type `A → B`), and a list of `As`, and returns the longest prefix of the list which satisfies the predicate. (The name of this function, and several in subsequent problems, comes from the Haskell Prelude.)
4. Prove that if value `a` satisfies predicate `p`, then `takeWhile p (repeat n a)` is equal to `repeat n a`, where `takeWhile` is the function you defined in the previous problem.
5. Define a function `take` which takes in a natural number `n` and a list `l`, and returns a list of the first `n` elements of `l` (in order), or all the elements if `n` exceeds the length of the list.
6. `list.agda` in the IAL already defines a similar function `nthTail` that returns the part of the list after the first `n` elements, or the empty list if the list has fewer than `n` elements. Prove that appending the result of `take` with the result of `nthTail` for any `n` (the same value given to both functions) results in the list again.

Chapter 5

Internal Verification

Up to now, we have written programs and proved properties about them. We wrote operations on booleans (Chapter 1), natural numbers (Chapter 3), and lists (Chapter 4), and after our code was written, we wrote some proofs of properties of those operations. This style of verification in type theory is called **external verification**: proofs are external to programs. That is, proofs are distinct artifacts one writes about some pre-existing programs.

In this chapter, we will consider a few examples of an alternative style of type-theoretic verification, called **internal verification**. Here, the idea is to write functions with more semantically expressive types. Often, these arise from datatypes which impose more restrictions on data than usual. A basic and commonly used example of this is the type of vectors. A vector is like a list, but the vector type explicitly records the length of the list. The type $\mathbb{V} \ A \ n$ is the type for vectors of A s of length n . Including the length of the list in the type allows us to express relationships between the lengths of input and output vectors, in functions we write. The type of such a function tells us an interesting semantic property about that function. This is different from first writing a more simply typed function, and then writing a proof about it. Instead, we are writing a richly typed function. Doing so may require us to embed proofs inside our code, to show Agda that the constraints imposed by these rich types are satisfied.

We will see several examples of internal verification in this chapter. First is the vector datatype just mentioned. Next we will look at **Braun trees**, a balanced tree data structure, where we can statically enforce the balancing property. The third example is a datatype for binary search trees (BSTs), a familiar and basic data structure in Computer Science for storing ordered data. We will be able to enforce the binary tree property statically. We will also look at sigma types, a simple generic way to add some internal verification with existing data structures.

5.1 Vectors

Lists are ubiquitous in mainstream functional programming. In dependently typed functional programming as in Agda, some variations on lists are used which enable more verification. One of these is the vector datatype. A vector is just like a list, except that the type tells us how long the vector is. So instead of the type $\mathbb{L} \ A$ for lists of elements of type A , we have a type $\mathbb{V} \ A \ n$ for vectors of length

n of elements of type A . Here, n is just a natural number. The vector type is said to be **indexed** by its length. By making the length of the vector explicit in the type, we can define vector-manipulating functions where the type of the function shows how the lengths of the vectors in question are related. The most commonly used example (it is almost the “hello world” of dependently typed programming) is vector append, which has type:

$$\forall \{\ell\} \{A : \text{Set } \ell\} \{n \ m : \mathbb{N}\} \rightarrow \mathbb{V} \ A \ n \rightarrow \mathbb{V} \ A \ m \rightarrow \mathbb{V} \ A \ (n + m)$$

The explicit inputs are vectors of length n and m , respectively, and the output has length $n + m$. We have expressed an interesting semantic property of the append function, namely how the length of the output vector is related to the input vectors, within the type of the append function itself. This is different from the approach we were following with previous datatypes and their operations. For list append, for example, we can prove this theorem:

$$\forall \{\ell\} \{A : \text{Set } \ell\} (l1 \ l2 : \mathbb{L} \ A) \rightarrow \\ \text{length } (l1 ++ l2) \equiv (\text{length } l1) + (\text{length } l2)$$

But this theorem is external to the actual code for append ($++$), and hence is an instance of external verification. In contrast, with length-indexed vectors, the analogous property is baked right into the type of the append function itself, and is an example of internal verification.

5.1.1 The Vector Datatype

The vector datatype is declared in `vector.agda` in the Iowa Agda Library:

```
data  $\mathbb{V}$   $\{\ell\}$  (A : Set  $\ell$ ) :  $\mathbb{N} \rightarrow \text{Set } \ell$  where
  [] :  $\mathbb{V} \ A \ 0$ 
  _::_ : {n :  $\mathbb{N}$ } (x : A) (xs :  $\mathbb{V} \ A \ n$ )  $\rightarrow \mathbb{V} \ A \ (\text{suc } n)$ 
```

For comparison, recall the definition of the list datatype:

```
data  $\mathbb{L}$   $\{\ell\}$  (A : Set  $\ell$ ) : Set  $\ell$  where
  [] :  $\mathbb{L} \ A$ 
  _::_ : (x : A) (xs :  $\mathbb{L} \ A$ )  $\rightarrow \mathbb{L} \ A$ 
```

The definition of the vector datatype is very similar to this one for lists. We have two constructors in each case, one for the empty list and one for a nonempty list. Agda supports overloading of data constructors, so we are able to use `[]` and `_::_` for the constructors for vectors as well as lists. Unfortunately, Agda does not support overloading functions other than constructors, so we will be forced to choose different names to distinguish similar functions on lists and on vectors, like vector append and list append. But we may as well take advantage of constructor overloading to use the same names for vector constructors as for list constructors.

For a couple examples: in `vector-test.agda` you will see these definitions:

```

test-vector :  $\mathbb{V} \ \mathbb{B} \ 4$ 
test-vector = ff :: tt :: ff :: ff :: []

test-vector2 :  $\mathbb{L} \ (\mathbb{V} \ \mathbb{B} \ 2)$ 
test-vector2 = (ff :: tt :: []) ::
               (tt :: ff :: []) ::
               (tt :: ff :: []) :: []

```

The first declares a vector of booleans of length 4 called `test-vector`. Agda knows that the `_::_` and `[]` constructors used here are for the vector (\mathbb{V}) datatype, rather than the list (\mathbb{L}) datatype, because of the type we have declared `test-vector` to have. If instead we declared `test-vector` to have type $\mathbb{L} \ \mathbb{B}$, Agda would again accept the definition; but this time the constructors would be understood to be those for the list datatype.

The second example, `test-vector2` is a list of vectors of length 2. Again, Agda is sorting out to which datatype the ambiguous uses of `_::_` and `[]` belong, based on the type we have declared for `test-vector2`. As a final example, `test-vector3` is a vector of vectors:

```

test-vector3 :  $\mathbb{V} \ (\mathbb{V} \ \mathbb{B} \ 3) \ 2$ 
test-vector3 = (tt :: tt :: tt :: []) ::
               (ff :: ff :: ff :: []) :: []

```

Because the vector datatype, like the list datatype, requires each element of the data structure to have the same type, we cannot (directly) create a vector whose elements are vectors of different lengths. That is why this example has a vector (of length 2) storing vectors all of the same length (namely 3).

5.1.2 Appending vectors

To append two vectors, the code is essentially the same as for appending two lists, except with a different type (this is from `vector.agda`):

```

_++ $\mathbb{V}$ _ :  $\forall \{\ell\} \{A : \text{Set } \ell\} \{n \ m : \mathbb{N}\} \rightarrow$ 
         $\mathbb{V} \ A \ n \rightarrow \mathbb{V} \ A \ m \rightarrow \mathbb{V} \ A \ (n + m)$ 
[]      ++ $\mathbb{V}$  ys = ys
(x :: xs) ++ $\mathbb{V}$  ys = x :: (xs ++ $\mathbb{V}$  ys)

```

The function name is `_++ \mathbb{V} _`, to avoid a name conflict with list append (which is `_++_`). As an aside: an alternative to using a different name would be to defer the resolution of name conflicts to the user of the IAL. Agda's module system provides enough tools to allow library users to avoid such conflicts – but for the IAL we have chosen not to burden the user with this concern. So we choose different names for different, even if similar, functions. Returning to the code: other than the function's name and type, this really is exactly the same definition as for list append (from `list.agda`), which is

```

_++_ : ∀ {ℓ} {A : Set ℓ} → ℒ A → ℒ A → ℒ A
[]      ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)

```

The type for the vector append (`_++V_`) operation is more informative than the type for list append:

```

∀ {ℓ} {A : Set ℓ} {n m : ℕ} → V A n → V A m → V A (n + m)

```

This says that vector append takes in a vector of `A`'s of length `n` and a vector of `A`'s of length `m`, and returns a vector of `A`'s of length `n + m`. The lengths `n` and `m` of the input lists are implicit arguments; Agda can usually deduce them from the types of the input vectors. We can see this in action in this example, from `vector-test.agda`:

```

test-vector-append : V B 8
test-vector-append = test-vector ++V test-vector

```

The type for `test-vector` was given earlier in the file; it is `V B 4`. So appending that vector with itself produces a vector whose length is `4 + 4`, which is definitionally equal to 8.

5.1.3 Head and tail operations on vectors

Defining a function to return the head of a vector is more elegant than for lists. This is because for lists, we have to deal with the corner case that the input list is empty, and does not have a head. But for vectors, we can use types to express the requirement that the vector must be nonempty if we are to get its head. Here is the definition, from `vector.agda`:

```

headV : ∀ {ℓ} {A : Set ℓ} {n : ℕ} → V A (suc n) → A
headV (x :: _) = x

```

The type for the `headV` function says that it takes in a vector of `A`'s of length `suc n`, for some implicit argument `n`. Since the length is a successor number, it cannot be zero. Hence, the list cannot be empty, and Agda allows us just to write the equation for that case. We do not have to write the equation for the empty list, and we do not even need to write the absurd pattern. Agda's type checking algorithm infers that in the case where the input vector is empty, the length would be zero, which cannot possibly match the length `suc n` given in the type. Hence, the equation for the empty vector (`[]`) does not need to be written. We can then call `headV` on any non-empty vector, to get its head.

The definition of the `tailV` function to return the tail of the input vector also is interesting. We do not need to restrict the input vector to be nonempty, if we accept the convention that the tail of the empty vector is just the empty vector. But we can still record some interesting information in the type:


```
tailV : ∀ {ℓ} {A : Set ℓ} {n : ℕ} → V A n → V A (pred n)
tailV [] = []
tailV (_ :: xs) = xs
```

The type is saying that `tailV` takes in a vector of `A`'s of length `n` (where `n` is an implicit argument), and returns a vector of `A`'s of length `pred n`. The `pred` function, defined in `nat.agda`, returns `n` if called with input `suc n`; and it returns zero if called with input `zero`. So in the equation for `[]` (the empty vector), the code for `tailV` returns `[]`, and Agda deduces that the length of this vector is indeed `pred n`, because when the input vector is `[]`, the length `n` must be zero, and `pred zero` is definitionally equal to zero.

5.1.4 Using types to express properties of other vector operations

In `vector.agda` in the Iowa Agda Library, we have other vector operations where the types make use of the length index to `V` to express interesting properties of the operations. Agda is able to deduce that these properties hold just by type-checking; no proofs are required from the programmer for these. Let us look at several of these.

The `mapV` is just like the `map` function on lists (see Section 4.2.3), except that it operates on vectors:

```
mapV : ∀ {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'} {n : ℕ} →
      (A → B) → V A n → V B n
mapV f [] = []
mapV f (x :: xs) = f x :: mapV f xs
```

The code is essentially identical to that of `map` on lists. We take in a function `f` of type `A → B` and recurse down the vector applying `f` to each element. But the type of `mapV` records an interesting property: `mapV` takes in a vector of `A`'s of length `n` and returns a vector of `B`'s of the same length (`n`). So `mapV` is a length-preserving function. For lists, we had to prove this property separately, as the `length-map` theorem in `list-thms.agda`:

```
length-map : ∀ {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'}
            (f : A → B) (l : L A) →
            length (map f l) ≡ length l
```

Here, the statement of the property is elegantly folded into the function's type, and its proof is folded into the definition of the function. In fact, no additional code in the definition of the function is required; Agda can see that this more informative type holds of the existing code, without further assistance from the programmer.

Another interesting example of expressing a property of the lengths of vectors through the type of a function is with `concatV`, shown in Figure 5.1. This func-

```

concatV : ∀{ℓ}{A : Set ℓ}{n m : ℕ} →
          V (V A n) m → V A (m * n)
concatV [] = []
concatV (x :: xs) = x ++V (concatV xs)

```

Figure 5.1: Concatenation of a vector of vectors

tion takes a vector of vectors, and concatenates them all. More specifically, it takes a vector of length m , where each element of that vector is itself a vector of length n . The resulting vector then has length $m * n$. Amazingly, this code type-checks without any further annotations from the programmer. The trick is in being careful about exactly how we phrase the type. Because the length of the outer vector is m , the variables x and xs will have types $V A n$ and $V A m$, respectively, where m is the length of xs . So the entire input list has length $\text{succ } m$ in this case. Making the recursive call on xs will then result in a vector of length $m * n$. And now we are appending x and this vector. The type (see Section 5.1.2) for $_++V_$ says that the resulting vector will have length $n + (m * n)$. And if we look again at the definition of $_*_$ (in `nat.agda`), we will see that this is definitionally equal to $(\text{succ } m) * n$, which is the length that the type of `concatV` says we should have for an input list of length $\text{succ } m$.

Let us look at a couple more examples of expressing properties of the lengths of vectors in the types of vector operations. As we saw in Section 4.2.6, the `nth` function for lists (in `list.agda`) attempts to return the n 'th element of input list l (where n is also an input to the function). If the list's length is less than n , then `nth` returns `nothing`. Otherwise it returns `just x`, where x is the n 'th element of the list. We can recall the code:

```

nth : ∀{ℓ}{A : Set ℓ} → ℕ → L A → maybe A
nth _ [] = nothing
nth 0 (x :: xs) = just x
nth (suc n) (x :: xs) = nth n xs

```

With vectors, we can eliminate the need for this `maybe` type. We must just require that the index n is less than the length of the vector. Here is the code, from `vector.agda`:

```

nthV : ∀ {ℓ} {A : Set ℓ} {m : ℕ} →
        (n : ℕ) → n < m ≡ tt → V A m → A
nthV 0 _ (x :: _) = x
nthV (suc n) p (_ :: xs) = nthV n p xs
nthV (suc n) () []
nthV 0 () []

```

Let us look at the type of `nthV` first:

$$\forall \{\ell\} \{A : \text{Set } \ell\} \{m : \mathbb{N}\} \rightarrow \\ (n : \mathbb{N}) \rightarrow n < m \equiv \text{tt} \rightarrow \mathbb{V} A m \rightarrow A$$

This says that `nthV` takes three explicit arguments: a number `n`, a proof that `n` is less than implicit argument `m`, and a vector of `A`'s of length `m`. It is then guaranteed to return a result of type `A`, not `maybe A` as we had for `nth` on lists.

Now let us look at the first two equations defining `nthV`:

```
nthV 0 _ (x :: _) = x
nthV (suc n) p (_ :: xs) = nthV n p xs
```

These correspond to the second and third equations defining `nth`, which we saw above. The first says that if element 0 is requested from non-empty list with head `x`, then `x` is returned. The code uses underscores in the pattern for this equation (and the next) to indicate that the matching values are not used on the right-hand side of the equation. The tail of the list, for example, is not needed in this case, so we just put an underscore in `(x :: _)`. In the second equation, if the caller is requesting element `suc n` from a list with tail `xs`, then we just recurse to get element `n` of `xs`. It is interesting to see why the proof `p` can be reused in this recursive call. In this second equation for `nthV`, `p` is a proof that the index is less than the length of the list. The index is `suc n`, and the length of the list is `suc m`, for some implicit `m` (in fact, because we have not introduced a name for `m` in the pattern, Agda will implicitly introduce the name `.m` and consider the length to be `suc .m`). So `p` proves `suc n < suc m ≡ tt`. But this type is definitionally equal to `n < m ≡ tt`, by the definition of `_<_`. So `p` in fact does have the correct type to make the recursive call with `n` and `xs` as the other two explicit arguments.

Finally, let us look at the last two defining equations for `nthV`:

```
nthV (suc n) () []
nthV 0 () []
```

We are using the absurd pattern for the proof in both cases, because the length of the list is zero, and hence no index can be smaller than that length. We do have to case-split on the index, though, for Agda to be able to see the absurdity. This is because the definition of `_<_` splits on both inputs, and returns `ff` separately for when the first input is 0 and the second is 0, and for when the first input is `suc n` and the second is 0.

Here is one last easy example:

```
repeatV : ∀ {ℓ} {A : Set ℓ} → (a : A) (n : ℕ) → V A n
repeatV a 0 = []
repeatV a (suc n) = a :: (repeatV a n)
```

If we request to form a vector of length `n` containing the element `a` of type `A` repeatedly, then the resulting vector indeed has length `n`.

5.2 Braun Trees

An elegant data structure for balanced binary heaps is the **Braun tree**. Braun trees are either empty or else a node consisting of some data x (of type A , let us say), and a left and a right subtree. The data can be stored so that x is smaller than all the data in the left and right subtrees, if such an ordering property is desired. But the crucial property of Braun trees is about the sizes of the left and right trees: for each node in the tree, either $\text{size}(\text{left}) = \text{size}(\text{right})$ or else $\text{size}(\text{left}) = \text{size}(\text{right}) + 1$. This ensures that the depth of the trees is less than or equal to $\log_2(N)$, where N is the number of nodes. We will see a nice trick for maintaining this property while inserting data into a Braun tree.

In Agda, we can actually specify this crucial property for the nodes of Braun trees, using types. But first, it is convenient to use Agda's module system to make the type A and the ordering on that type parameters of the whole development in the module. This is done this way, in `braun-tree.agda` from the Iowa Agda Library:

```
module braun-tree{ℓ} (A : Set ℓ) (_<A_ : A → A → ℬ) where
```

To use this module later, we will need to supply values for these parameters; for example, we might instantiate A with \mathbb{N} and $_<A_$ with $<$ (the usual strict ordering on the natural numbers).

5.2.1 The `braun-tree` datatype, and sum types

Here is the definition of the Braun tree datatype (from `braun-tree.agda`):

```
data braun-tree : (n : ℕ) → Set ℓ where
  bt-empty : braun-tree 0
  bt-node : ∀ {n m : ℕ} →
    A → braun-tree n → braun-tree m →
    n ≡ m ∨ n ≡ suc m →
    braun-tree (suc (n + m))
```

We are defining the type `braun-tree n` for Braun trees storing n nodes; we will call these Braun trees of size n . So the first line of the datatype declaration, similarly to that for vectors above, says that the `braun-tree` type is indexed by a natural number n . Then there are two constructors. `bt-empty` creates a Braun tree of size 0. The interesting case, naturally, is the `bt-node` constructor. Its type says that it takes in implicit natural number n and m , and some data of type A . Then it takes in a left Braun tree of size n and a right one of size m . Finally, it takes in a proof of the following formula:

$$n \equiv m \vee n \equiv \text{suc } m$$

```

data _⊕_ {ℓ ℓ'} (A : Set ℓ) (B : Set ℓ')
  : Set (ℓ ⊔ ℓ') where
  inj1 : (x : A) → A ⊕ B
  inj2 : (y : B) → A ⊕ B

_∨_ : ∀ {ℓ ℓ'} (A : Set ℓ) (B : Set ℓ') → Set (ℓ ⊔ ℓ')
_∨_ = _⊕_

```

Figure 5.2: The sum type, and a definition

We are using the \vee operator, defined in `sum.agda`, for a disjunction of two types.

Either we have a proof that n equals m ($n \equiv m$), or else that n equals `suc m` ($n \equiv \text{suc } m$). This is the crucial property of Braun trees mentioned above. Since `bt-node` requires a proof of this property in order to build a node, we will know whenever we have a Braun tree that the property is satisfied, for every node. The `bt-node` constructor says that the size of the returned Braun tree is `suc (n + m)`, accounting for the data at the node and the data in the two subtrees.

Since we will need to use this sum type in the rest of the code for Braun trees, let us pause to see the relevant parts of `sum.agda`, in Figure 5.2. Following the Agda standard library, we define a type `_⊕_`, parametrized by two types A and B . These types are possibly at different levels, which is accounted for by using the operation `_⊔_` on levels in the return type of `_⊕_`. This operator `_⊔_` is part of Agda's primitive level system; you can see it imported from a special `Agda.Primitive` module in `level.agda` in the IAL.

Returning to Figure 5.2: there are two constructors. If we have a value x of type A , then `inj1 x` has type $A \oplus B$. Similarly, if we have a value y of type B , then `inj2 y` has type $A \oplus B$. So you can create a value of type $A \oplus B$ from a value of type A , and also from a value of type B . When we are using sum types in code we intend to run, let us use `_⊕_`. But if we want to use a sum type to represent a logical proposition, as with the crucial property of Braun trees, then we will instead use `_∨_`, which is just defined (in Figure 5.2) to be `_⊕_`. There is no semantic difference between these operators. The choice of one or the other is purely to aid human comprehension of code and proofs.

5.2.2 Inserting into a Braun tree

Insertion into a Braun tree is simple and quite elegant. To insert an element into an empty Braun tree (`bt-empty`), we just create a node storing the element, with `bt-empty` for the two subtrees. The Braun tree property is preserved when we do this, because the sizes of the left and right subtree are equal (both 0). Suppose now that the Braun tree into which we are inserting the element is a node (`bt-node`). Then this tree has left subtree l and right subtree r satisfying the Braun tree prop-

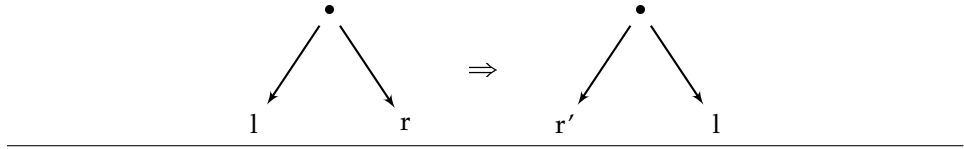


Figure 5.3: Insertion into a Braun tree node

```

bt-insert : ∀ {n : ℕ} → A → braun-tree n →
            braun-tree (suc n)
bt-insert a bt-empty =
  bt-node a bt-empty bt-empty (inj₁ refl)
bt-insert a (bt-node{n}{m} a' l r p)
  rewrite +comm n m
  with p | if a <A a' then (a , a') else (a' , a)
bt-insert a (bt-node{n}{m} a' l r _) | inj₁ p | (a1 , a2)
  rewrite p = (bt-node a1 (bt-insert a2 r) l (inj₂ refl))
bt-insert a (bt-node{n}{m} a' l r _) | inj₂ p | (a1 , a2) =
  (bt-node a1 (bt-insert a2 r) l (inj₁ (sym p)))

```

Figure 5.4: Code to insert data into a Braun tree

erty: either l and r are the same size, or else the size of l is one bigger. Then the insertion algorithm inserts the element recursively into r to obtain a resulting tree r' . The size of this r' is now either equal to the size of l , if the size of l was one bigger than the size of the original r ; or else the size of r' is one bigger than the size of l (if the size of l was the same as that of r). In either case, the Braun tree property is preserved if we use r' as the new left subtree, and l as the new right subtree. This is depicted in Figure 5.3. The figure does not show the data in the tree, only the structural changes to maintain the Braun tree property.

Figure 5.4 shows the code from `braun-tree.agda` implementing the insertion algorithm just described. The type for this `bt-insert` function is already interesting, since it says that given a Braun tree of size n , `bt-insert` will return a Braun tree of size `suc n`. This reflects the fact that we really will add a new element to the data structure, so the size is increased by one. Next, let us look at the equations defining `bt-insert`. There are three cases that are being covered. The first case is for when we are inserting into a `bt-empty` (this is the first equation in Figure 5.4):

```

bt-insert a bt-empty =
  bt-node a bt-empty bt-empty (inj₁ refl)

```

This is an easy case, as we would expect. We just need to create a new `bt-node` holding the element a . The new node has empty Braun trees (`bt-empty`) for its left and right subtrees. The fourth argument to the `bt-node` constructor is supposed to be a proof of the Braun tree property for the left and right subtrees.

Here, since the sizes are equal (both 0), we can just use `refl` as the proof, although we must wrap it in the `inj1` constructor for the \vee type, to indicate that we are proving the first disjunct of the disjunction that defines the Braun tree property. This is (see Section 5.2.1 above)

$$(n \equiv m) \vee (n \equiv \text{suc } m)$$

where n is the size of the left subtree and m is the size of the right one.

The next clause in the definition of `bt-insert` is

```
bt-insert a (bt-node{n}{m} a' l r p)
  rewrite +comm n m
  with p | if a <A a' then (a , a') else (a' , a)
```

This is for when we are inserting data a into a `bt-node` with left subtree l of size n and right subtree r of size m . The pattern variable p is the proof of the Braun tree property for this node. In this case, Agda deduces that the type of the return value of the function must be

```
braun-tree (suc (suc (n + m)))
```

This is because the type of the `bt-node` is `braun-tree (suc (n + m))`, and we are incrementing the size when we do an insertion. The clause shown above first applies commutativity of addition to swap n and m in the return type. So now we must produce an output value of type

```
braun-tree (suc (suc (m + n)))
```

The code applies this rewrite at this stage, because regardless of what happens next, we are going to swap the left and right subtrees, so the summation of their sizes will have m first. So we can factor out that rewrite, before we case split on which disjunct of the Braun tree property holds: is n equal to m or is it equal to `suc m`? This will not change the structure of the tree, but it will change what proof we use for the Braun tree property for the new node we will return. To do this case split, we do a `with` on p . Normally one would just split on p directly as part of the pattern for the input Braun tree. But this way, we only have to write the `rewrite` once. We also do a `with` on an `if_then_else_` term, which is just putting the minimum of a (the element we are inserting) and a' (the element currently at the root of the tree into which we are inserting) as the first component a_1 of a pair, and the maximum as the second component a_2 . Then we know we want a_1 (smaller element) to be the data at the root of the new `bt-node` we will return, and we want to insert a_2 (bigger) recursively into the right subtree r .

The next clause in the definition of `bt-insert` now considers the case where p is `inj1 p` for some new pattern variable p :

```
bt-insert a (bt-node{n}{m} a' l r _) | inj1 p | (a1 , a2)
  rewrite p = (bt-node a1 (bt-insert a2 r) l (inj2 refl))
```

In fact, the code has an underscore where the old variable p was, because we will no longer use it, now that we are considering the case where that original p is

inj_1 p . So here, p has type $n \equiv m$. That is, we are considering the first case for the Braun tree property, where the sizes of the left and right subtrees are the same. Here, the proof we use for the Braun tree property for the new node we are constructing (which has the smaller element a_1 at the root and then has swapped the left and updated right subtrees) is inj_2 refl . The required property for this bt-node is:

$$\text{suc } m \equiv n \vee \text{suc } m \equiv \text{suc } n$$

This is because the size of the new left subtree is $\text{suc } m$, since it is the updated version of the old right subtree. But we are in a case where we have a proof p that n equals m . So the code does a rewrite with this p , which ends up changing that property to

$$\text{suc } m \equiv m \vee \text{suc } m \equiv \text{suc } m$$

And then inj_2 refl suffices to prove this, since we can prove the left disjunct using refl .

The final clause from Figure 5.4 for the definition of bt-insert is even easier:

$$\text{bt-insert } a \text{ (bt-node}\{n\}\{m\} \text{ } a' \text{ } l \text{ } r \text{ } _) \mid \text{inj}_2 \text{ } p \mid (a_1 \text{ } , \text{ } a_2) = \\ (\text{bt-node } a_1 \text{ (bt-insert } a_2 \text{ } r) \text{ } l \text{ (inj}_1 \text{ (sym } p)))$$

Here, we have a proof p of $n \equiv \text{suc } m$. So what we have to prove for the fourth explicit argument to bt-node for the return value is

$$\text{suc } m \equiv n \vee \text{suc } m \equiv \text{suc } n$$

But this is very easy, just by applying sym to our proof p to obtain a proof of $\text{suc } m \equiv n$, and then using inj_1 to prove the disjunction.

5.2.3 Removing the minimum element from a Braun tree

The code to remove the minimum element from a Braun tree modifies the structure of the Braun tree in the opposite way as insertion (so just reverse Figure 5.3). Figure 5.5 shows the code, from `braun-tree.agda`. Let us walk through this slowly. First, there is the type of bt-remove-min , which is already interesting:

$$\forall \{p : \mathbb{N}\} \rightarrow \text{braun-tree } (\text{suc } p) \rightarrow A \times \text{braun-tree } p$$

This says that bt-remove-min takes as input a tree with at least one element, since the size is $\text{suc } p$; and returns a pair of an element of type A and a Braun tree of size p . So the type is expressing the property that bt-remove-min reduces the size of the Braun tree by one. Next come four simple cases. The first is for when we are removing the sole node from a Braun tree. That is, the Braun tree is a node where the subtrees are both bt-empty . In this case (the first equation in Figure 5.5), we return the data and bt-empty . Agda can easily deduce in this case that the size of the input Braun tree is $\text{suc } 0$ and the size of the returned tree is 0 . The next two equations are for if the left subtree is bt-empty and the right subtree


```

bt-remove-min :  $\forall \{p : \mathbb{N}\} \rightarrow \text{braun-tree } (\text{suc } p) \rightarrow$ 
                 $A \times \text{braun-tree } p$ 
bt-remove-min (bt-node a bt-empty bt-empty u) =
  a , bt-empty
bt-remove-min (bt-node a bt-empty (bt-node _ _ _ _))
  (inj1 ())
bt-remove-min (bt-node a bt-empty (bt-node _ _ _ _))
  (inj2 ())
bt-remove-min (bt-node a (bt-node{n'}{m'} a' l' r' u')
  bt-empty u)
  rewrite +0 (n' + m') = a , bt-node a' l' r' u'
bt-remove-min (bt-node a (bt-node a1 l1 r1 u1)
  (bt-node a2 l2 r2 u2) u)
  with bt-remove-min (bt-node a1 l1 r1 u1)
bt-remove-min (bt-node a (bt-node a1 l1 r1 u1)
  (bt-node a2 l2 r2 u2) u) | a1' , l'
  with if a1' <A a2 then (a1' , a2) else (a2 , a1')
bt-remove-min (bt-node a (bt-node{n1}{m1} a1 l1 r1 u1)
  (bt-node{n2}{m2} _ l2 r2 u2) u)
  | _ , l' | smaller , other
  rewrite +suc (n1 + m1) (n2 + m2) |
    +comm (n1 + m1) (n2 + m2) =
  a , bt-node smaller (bt-node other l2 r2 u2) l' (lem u)
where lem :  $\forall \{x y\} \rightarrow \text{suc } x \equiv y \vee \text{suc } x \equiv \text{suc } y \rightarrow$ 
             $y \equiv x \vee y \equiv \text{suc } x$ 
  lem (inj1 p) = inj2 (sym p)
  lem (inj2 p) = inj1 (sym (suc-inj p))

```

Figure 5.5: Code for removing the minimum element from a Braun tree

is a node. This is impossible, because the Braun tree property says that the sizes of the subtrees should be the same or else the left should be one bigger; and here the right is bigger than the left. So we can use the absurd pattern `()` to show Agda that these cases cannot happen. But we do have to break this case down into an absurdity if the first disjunct of the Braun tree property holds – this is `inj1 ()` – and an absurdity if the second one holds (`inj2 ()`). We cannot just put the absurd pattern for the entire proof in a single equation. Agda needs us to break the proof down to an absurdity in two equations. The last easy case (the fourth equation in Figure 5.5) is for when the right subtree is `bt-empty`, and the left subtree is a node. This implies that the size of the left subtree must be 1, but this is not needed here. We just return the data `a` and the left subtree. Agda needs a little help confirming that the size relationships expressed by the type of `bt-remove-min` have been satisfied, because the size of the input tree is `suc (suc (n' + m') + 0)`, while the size of the output tree is `suc (n' + m')`. We apply the `+0` lemma from `nat-thms.agda` to drop that `+ 0` from the type of the input tree, which then lets Agda see that the sizes of the input tree and output tree differ only by a single `suc` in the right place.

Before we continue to the more substantive cases, notice that Agda does not need us to write any clause for when the input Braun tree is empty. This is because then the size would be 0, which clashes with the size `suc p` stated in the theorem. Agda will automatically deduce from this clash (where the top-level constructors `0` and `suc` are different) that no clause is needed for when the input is `bt-empty`.

The next part of the code from Figure 5.5 is

```
bt-remove-min (bt-node a (bt-node a1 l1 r1 u1)
                      (bt-node a2 l2 r2 u2) u)
  with bt-remove-min (bt-node a1 l1 r1 u1)
bt-remove-min (bt-node a (bt-node a1 l1 r1 u1)
                      (bt-node a2 l2 r2 u2) u) | a1' , l'
  with if a1' <A a2 then (a1' , a2) else (a2 , a1')
```

The pattern is for when the left and right subtrees of the input tree are both nodes themselves (not empty). We are certainly going to return the data `a` of the input tree as the minimum data. But to reassemble the resulting Braun tree we need to remove the minimum from the left subtree. This is being done with the first `with` in the code shown just above. We match on the result of the recursive call to `bt-remove-min`. This produces the minimum element `a1'` of the left subtree, and an updated left subtree `l'`. Then we do another `with` to pick the smaller of `a1'` (the minimum of the left subtree) and `a2`, the minimum of the right subtree. This is similar to what we did in the code for `bt-insert` with an `if_then_else_` term.

The code from Figure 5.5 continues this way:

```
bt-remove-min (bt-node a (bt-node{n1}{m1} a1 l1 r1 u1)
                      (bt-node{n2}{m2} _ l2 r2 u2) u)
  | _ , l' | smaller , other
```

```

rewrite +suc (n1 + m1) (n2 + m2) |
      +comm (n1 + m1) (n2 + m2) =
a , bt-node smaller (bt-node other l2 r2 u2) l' (lem u)

```

We have extended our pattern with `smaller` , `other`. These are now what used to be called `a1'` and `a2`, in order by size. We have also dropped `a1'` and `a2` from the pattern, to remind ourselves that we should not reference them any more (instead using `smaller` and `other`). At this point, the size of the return Braun tree is supposed to be

```
suc (n1 + m1 + suc (n2 + m2))
```

Based on the Braun tree we are actually going to return, we want to reorganize this expression to

```
suc (suc (n2 + m2 + (n1 + m1)))
```

This reorganization is done by the rewrites with `+suc` and `+comm`. Then we return a pair of the data at the root of the tree, together with the new Braun tree where we have put the updated left subtree `l'` as the new right subtree, and the original right subtree as the new left subtree. The new data at the root is `smaller`, and `other` is the new data in the left subtree. All that is left to do is to prove the Braun tree property for the new tree we are returning. This is a simple case of arithmetic reasoning, which we carry out using a helper lemma in a `where` clause. A `where` clause allows one to define a recursive function locally within a single equation for another function. This is the definition of `lem` in this `where` clause (from Figure 5.5):

```

where lem : ∀{x y} → suc x ≡ y ∨ suc x ≡ suc y →
      y ≡ x ∨ y ≡ suc x
      lem (inj1 p) = inj2 (sym p)
      lem (inj2 p) = inj1 (sym (suc-inj p))

```

This proof allows us to go from the proof `u` of the Braun tree property for the `bt-node` we are processing in the current equation, to a proof of the Braun tree property for the new tree we are returning. The type of `u` is:

```

suc (n1 + m1) ≡ suc (n2 + m2) ∨
suc (n1 + m1) ≡ suc (suc (n2 + m2))

```

The Braun tree property for the tree we are returning is:

```
suc (n2 + m2) ≡ n1 + m1 ∨ suc (n2 + m2) ≡ suc (n1 + m1)
```

Abstracting `n1 + m1` as `x` and `suc (n2 + m2)` as `y`, this property matches the conclusion of `lem`, and the premise of `lem` matches the type of `u`. So that is all we need to conclude this case and thus the definition of `bt-remove`.

```
data  $\Sigma$  { $\ell$   $\ell'$ } (A : Set  $\ell$ ) (B : A  $\rightarrow$  Set  $\ell'$ )
  : Set ( $\ell \sqcup \ell'$ ) where
  _,_ : (a : A)  $\rightarrow$  (b : B a)  $\rightarrow$   $\Sigma$  A B
```

Figure 5.6: Declaration of sigma types, from `product.agda`

5.3 Sigma Types

As we have seen above, it is possible to express invariant properties of data in Agda using internally verified datatypes. Any data constructed using the constructors of the datatype are guaranteed to satisfy the property, due to restrictions imposed by the constructors. Declaring new datatypes to enforce new invariant properties is a common, effective practice in dependently typed programming. There are some situations, however, where one may prefer not to define a new datatype, but still have the ability to state that a property holds of some data. This can be done using so-called Σ -types (“sigma”). A Σ -type is a generalization of the usual Cartesian product type $A \times B$, and is often referred to as a dependent product type (though the notation comes from sum types, as we will note further below). The elements of $A \times B$ are ordered pairs (a, b) , where a is an element of type A , and b of type B . The generalization with Σ -types is that the type of the second element can depend on the type of the first. For a simple example, the file `nat-nonzero.agda` in the IAL defines a type for nonzero natural numbers using a Σ -type:

```
 $\mathbb{N}^+$  : Set
 $\mathbb{N}^+$  =  $\Sigma$   $\mathbb{N}$  ( $\lambda$  n  $\rightarrow$  iszero n  $\equiv$  ff)
```

An element of this type is a natural number n , together with a proof that `iszero n` equals `ff`. To understand better how this type works, let us imagine trying to write it as a Cartesian product:

```
 $\mathbb{N} \times$  (iszero n  $\equiv$  ff)
```

This shows the rough structure of the type: it is a pair of a number and an equality proof. But there is an obvious problem: the variable `n` is free, but we want it to refer to the value of the first component. This is what Σ -types enable. Figure 5.6 shows the definition, from `product.agda`. Let us puzzle through this definition carefully. The Σ type constructor is parametrized by a type A and then a function B that takes in an element of type A and returns a type. The types in question can be at different levels ℓ and ℓ' . Like sum types, the Σ type itself is then at level $\ell \sqcup \ell'$, the least upper bound of the two levels.

The constructor for Σ -types is `_,_`, which is a very nice choice (adopted in the IAL from the Agda standard library), because it lets us indeed write elements of Σ -types as pairs. The first element of the pair is some a of type A , and the second is

dependent
product

```

_+^+_ : ℕ+ → ℕ+ → ℕ+
(zero , ()) +^+ n2
(1 , p1) +^+ y = suc+ y
(suc (suc n1) , p1) +^+ y = suc+ ((suc n1 , refl) +^+ y)

```

Figure 5.7: Addition on nonzero natural numbers

some b of type $B\ a$. By using $B\ a$ instead of just B , we express the dependence of the type of the second component of the pair on the value of the first component.

For an example of using Σ -types, we can look further at `nat-nonzero.agda`, where successor and addition functions on nonzero natural numbers are defined. Let us look at the code for addition, in Figure 5.7. The code pattern-matches on the first nonzero natural number to be added. In each case, the pattern used is a pair, built with the `_ , _` constructor. We match further on the form of the first component of this pair, which is the number that is proved (by the second component of the pair) to be nonzero. If the first component is `zero` – well, that cannot happen, since the number is known to be nonzero. So the first defining equation for `_+^+_` uses the absurd pattern in that case. If the first component is `1` (the second defining equation), then we just take the successor of the second argument (y) to `_+^+_`, using a `suc+` function whose definition I have skipped here (see `nat-nonzero.agda`). Finally, we have a third equation for if the first component is `suc (suc n1)`. In this case, we know we can make a recursive call to `_+^+_` using `suc n1` as the first argument, because this quantity can be proved to be nonzero using `refl`. This is because `iszero (suc n1)` is definitionally equal to `ff`, as required for a nonzero natural. We use `suc+` to increment the result of the recursive call.

5.3.1 Why Sigma and Pi?

As an aside on notation: one might wonder why the notation Σ is used for the type of dependent pairs. The answer is that Σ -types can be thought of as generalizing disjoint unions. In mathematics, a disjoint union is a union of two sets where elements are tagged to indicate from which set they have come. The cardinality of $A \uplus B$, for example, is the sum of the cardinalities of A and B , even if A and B have a nonempty intersection. This is where we get the notation for sum types in Figure 5.2. The disjoint union can be defined mathematically as

$$(\{0\} \times A) \cup (\{1\} \times B)$$

In other words, each element of the union looks like (n, x) , where if the tag n is 0 then $x \in A$, and if n is 1, then $x \in B$. What if one wants to have a disjoint union of an infinite family of types:

$$B_1 \uplus B_2 \uplus \dots$$

In this case, the tags come from whatever the index set is for the infinite family. If the index set is \mathbb{N} as in the displayed union, then the disjoint union can be denoted $\Sigma n \in \mathbb{N}. B_n$. More generally, we can have any index set A , and then the set is $\Sigma a \in A. B_a$. Elements are pairs (a, b) , where $a \in A$ and $b \in B_a$.

Incidentally, dependent function types, written $(x : A) \rightarrow B$ in Agda, are mathematically denoted $\Pi x : A. B$, and considered a (different!) generalization of Cartesian products. For a Cartesian product of a finite number of sets, one just lists the elements in a tuple:

$$(b_1, b_2, \dots, b_n)$$

If the number of sets in the product is infinite, however, we need an infinite tuple:

$$(b_1, b_2, \dots)$$

One way to describe an infinite tuple is as a function from the position in the tuple to the value. If the infinite family of sets in the product is indexed by natural numbers, then the infinite tuple can be viewed as a function mapping $n \in \mathbb{N}$ to an element $b \in B_n$. More generally, the function may map $a \in A$ to $b \in B_a$, where A is the index set for the infinite family of sets. This explains why the elements of types $\Pi x : A. B$ are viewed as functions from $x \in A$ to B_x .

5.4 Binary Search Trees

Binary search trees are a basic data structure for storing ordered values. We assume we have some type A and an ordering relation $_ \leq_ A$ on A . The tree consists of leaves, which do not hold any value, and nodes, which hold a value (of type A) and a left and right subtree. For example, Figure 5.8 gives an example of a binary search tree storing natural numbers under the usual less-than-or-equal ordering. The essential property of binary search trees is that at any node N , the values in the left subtree are all less than or equal to the value at node N ; and the values in the right subtree are all greater than or equal to the value at node N . We will enforce this as an invariant using an internally verified data structure for binary search trees.

5.4.1 The `bst` module, and `bool-relations`

The development of binary search trees is in `bst.agda` in the Iowa Agda Library. We use Agda's module system to make the type A and the ordering $_ \leq_ A$ parameters of all the code we will write. To use the `bst` module, we will have to instantiate these parameters with a particular type and ordering, respectively. We actually also need a couple properties of the ordering, for the code we have written to type-check. These properties are also parameters of the module. So at the

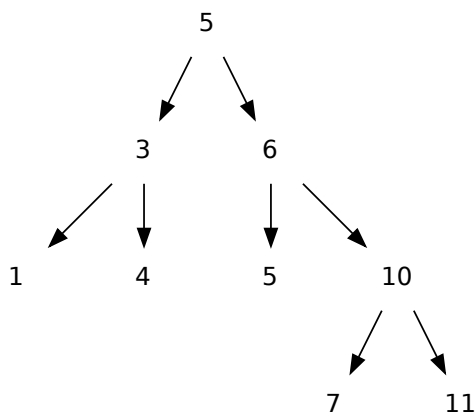


Figure 5.8: An example binary search, where values are natural numbers under the usual \leq ordering

top of `bst.agda`, you will see the code shown in Figure 5.9. The `open import` statement imports definitions of transitivity and totality from the module

`bool-relations.agda`

The `using` keyword indicates that we are importing only those definitions, not any others, from that module. We will look at those definitions in a moment. The `module bst` statement declares a module named `bst` (similarly to what we have seen in all the other Agda files we have studied), but with parameters:

- `A` is the type for the elements we will store
- `_≤A_` is the ordering, which takes two elements of type `A` and returns `tt` iff the first element is less than or equal to the second
- `≤A-trans` is a proof of transitivity for the ordering.
- `≤A-total` is a proof of totality of the ordering.

The IAL has separate files for relations (`relations.agda`) and boolean relations (`bool-relations.agda`). The former models a binary relation on `A` as a function from `A` to `A` to `Set`. The latter models them as functions from `A` to `A` to `ℬ`. We cannot use the former kind of relations to `Set` in code, because code cannot manipulate types. But code can pattern match on values of type `ℬ`, of course, so the latter boolean relations are more directly useful for programming. Sometimes we need the more general form of relation; we will see some examples in Chapter 9.

```

open import bool-relations using (transitive ; total)

module bst (A : Set) (_≤A_ : A → A →  $\mathbb{B}$ )
  (≤A-trans : transitive A _≤A_)
  (≤A-total : total A _≤A_) where

```

Figure 5.9: The parameter declarations for the `bst` module, and the import of `bool-relations`

```

module bool-relations { $\ell$  : level}{A : Set  $\ell$ }
  (_≤A_ : A → A →  $\mathbb{B}$ ) where

open import relations ( $\lambda$  a a' → a ≤A a'  $\equiv$  tt) public
  using (reflexive ; transitive)

total : Set  $\ell$ 
total =  $\forall$  {a b : A} → a ≤A b  $\equiv$  ff → b ≤A a  $\equiv$  tt

total-reflexive : total → reflexive
total-reflexive tot {a} with keep (a ≤A a)
total-reflexive tot {a} | tt , p = p
total-reflexive tot {a} | ff , p = tot p

_isoB_ : A → A →  $\mathbb{B}$ 
d isoB d' = d ≤A d' && d' ≤A d

isoB-intro :  $\forall$ {x y : A} → x ≤A y  $\equiv$  tt → y ≤A x  $\equiv$  tt →
  x isoB y  $\equiv$  tt
isoB-intro p1 p2 rewrite p1 | p2 = refl

```

Figure 5.10: Code from `bool-relations.agda`

```

module relations { $\ell$   $\ell'$  : level}{A : Set  $\ell$ }
  (_≥A_ : A → A → Set  $\ell'$ ) where

reflexive : Set ( $\ell \sqcup \ell'$ )
reflexive =  $\forall$  {a : A} → a ≥A a

transitive : Set ( $\ell \sqcup \ell'$ )
transitive =  $\forall$  {a b c : A} → a ≥A b → b ≥A c → a ≥A c

```

Figure 5.11: Code from `relations.agda`

If we pause for a moment and look at `relations.agda`, we will find the definitions in Figure 5.11 for reflexivity and transitivity. The ordering is pointed the other way for this file, so we have `_≥A_` instead of `_≤A_`. Such a relation on type `A` is reflexive iff every element `a` is related by the relation to itself. It is transitive iff whenever `a` is greater than or equal to `b`, which is in turn greater than or equal to `c`, then `a` is greater than or equal to `c`. In Figure 5.10 code from `bool-relations.agda` is shown. We do an `open import` on the `relations` module, to get the definitions of `reflexive` and `transitive`, for the relation

```
λ a a' → a ≤A a' ≡ tt
```

This relation is defined using a lambda-abstraction (see Section 4.2.5), and is just the `Set` relation that says that the boolean relation `_≤A_` returns `tt` for the two inputs. Figure 5.10 has a definition of totality for boolean relations, which we will need for our implementation of binary search trees. A relation is total iff whenever `a` is not less than or equal to `b`, then `b` must be less than or equal to `a`.

In mathematics, orderings satisfying transitivity and totality are called **total preorders**. Familiar orderings like `≤` on natural numbers are total preorders, for example. Being a total preorder is all that is required for the code that we will consider in `bst.agda` (and both properties are used). A preorder is a reflexive and transitive binary relation. For total preorders, reflexivity actually follows from totality. Thus the code shown in Figure 5.10 has a proof `total-reflexive` that totality implies reflexivity. Our use of Agda’s module system has enabled the very concise statement of the theorem: since the type `A` and the ordering on it are parameters to the module, we do not have to state them in the theorem. The intuition behind the proof is that we can split on the boolean value of `a ≤A a`. If it is `tt`, we are done, since proving `a ≤A a ≡ tt` is the goal. Note the use of the `keep` idiom, introduced in Section 4.3.4, to add a proof `p` to the context that `a ≤A a ≡ tt` (in the first equation following the `with`). In the second equation following the `with`, `p` is a proof of `a ≤A a ≡ ff`, to which we can apply our posited proof of totality (`tot`). Totality says that if `x ≤A y` is `ff`, then `x ≤A y` must be `tt`. Here, `x` and `y` are both `a`, giving us `a ≤A a ≡ tt` as required.

Let us consider the last two definitions shown in Figure 5.10. We define an infix operator `_isoB_`, short for “isomorphic” to hold of `d` and `d'` iff each is related by the relation to the other. Isomorphism is a derived notion of equality, and we can indeed prove that it is an equivalence relation (this is not shown). There follows just a simple convenience lemma for proving isomorphism when we have separate proofs for the required relations between the two elements in question. We will use `_isoB_ below` for `bst-search` (Section 5.4.3).

One small note on the use of the module system: in `bst.agda`, we saw that there is the following import directive.

```
open import bool-relations using (transitive ; total)
```

The module-attuned reader may have noticed that surprisingly, we did not instantiate the parameters of the relation module here. It is allowed in Agda not to in-

```

data bst : A → A → Set where
  bst-leaf : ∀ {l u : A} → l ≤A u ≡ tt → bst l u
  bst-node  : ∀ {l l' u' u : A} (d : A) →
    bst l' d → bst d u' →
    l ≤A l' ≡ tt → u' ≤A u ≡ tt →
    bst l u

```

Figure 5.12: The binary search tree datatype

stantiate all parameters of a module when importing it. In that case, all definitions in the module are automatically extended to require the omitted parameters as additional arguments. This is handy here, since it means we can apply `transitive` and `total` to the parameters `A` and `_≤A_` of the `bst` module. The `using` part of the `import` tells Agda we only wish to use the listed symbols (`transitive` and `total`) from the module; the other symbols in the module will not be brought into scope. Then, after we have declared the `bst` module with its parameters, the `bst.agda` file continues by opening the `bool-relations` module with the required parameters. We hide `transitive` and `total` in this new opening, to avoid conflicting with the definitions of those names we got with the first opening of `bool-relations`. We also open a module `minmax`, which will come into play for `bst-insert` (Section 5.4.4). The import directives in question are:

```

open import bool-relations _≤A_ hiding (transitive ; total)
open import minmax _≤A_ ≤A-trans ≤A-total

```

5.4.2 The `bst` datatype

Now let us move on to the definition of the `bst` datatype, in Figure 5.12. The basic idea is that the type for a binary search tree with values between a lower bound `l` and an upper bound `u` will be `bst l u`. So we are making a lower bound and an upper bound on the values in the binary search tree indices to the `bst` type. Just as we saw with vectors, this will enable us to express properties relating these bounds to other values, which will be sufficient to enforce the essential property of binary search trees. It turns out to be convenient to give ourselves a little leeway with the bounds: they will not be tight. So a tree with type `bst l 10` (with the usual ordering for natural numbers) might not store a number as small as `l` or as big as `10`. But all numbers in the tree will definitely be between those bounds.

5.4.3 Searching for an element in a binary search tree

Code for searching for an element in a binary search tree is given in Figure 5.13. Let us look first at the type given for `bst-search`. It says that `bst-search` takes in a value `d` and a binary search tree with values between `l` and `u`, and then either

```

bst-search : ∀{l u : A} (d : A) → bst l u →
              maybe (Σ A (λ d' → d isoB d' ≡ tt))
bst-search d (bst-leaf _) = nothing
bst-search d (bst-node d' L R _ _) with keep (d ≤A d')
bst-search d (bst-node d' L R _ _) | tt , p1
  with keep (d' ≤A d)
bst-search d (bst-node d' L R _ _) | tt , p1 | tt , p2 =
  just (d' , isoB-intro p1 p2)
bst-search d (bst-node d' L R _ _) | tt , p1 | ff , p2 =
  bst-search d L
bst-search d (bst-node d' L R _ _) | ff , p1 =
  bst-search d R

```

Figure 5.13: Searching for an element in a binary search tree

returns `nothing` (a constructor of the `maybe` type, as considered in Section 4.2.6); or else `just (d' , p)`, where `d'` is another value, and `p` is a proof that `d'` is isomorphic to `d`, in the sense explained in Section 5.4.1 (Figure 5.10). We are using a dependent product type (see Section 5.3) to return both the value `d'` and the proof that it is isomorphic to `d`.

The code in Figure 5.13 is relatively straightforward. If the tree is a `bst-leaf` then we return `nothing`, signalling that we have not found the value in the tree. Otherwise, we compare the value `d` for which we are searching with the value stored in the given `bst-node`. The code is checking whether or not these are isomorphic by checking first `d ≤A d'` (inside one `keep`), and then `d' ≤A d` (within another). If both conditions are true, we have found an isomorphic `d'`, and we return the value

```
just (d' , isoB-intro p1 p2)
```

This has the required return type `maybe (Σ A (λ d' → d isoB d' ≡ tt))`, since it is `just` (a constructor of `maybe`) of a pair of a value `d'` of type `A`, and a proof, constructed using the helper lemma `isoB-intro` from Figure 5.10, that this element is isomorphic to the one we were searching for. In the other two cases, we search recursively in either the left or right subtree, depending on whether `d` is less than or equal to `d'` but not vice versa; or whether `d'` is less than or equal to `d` but not vice versa.

We are using internal verification in the definition of `bst-search`, to show that the element returned by `bst-search`, if any, is isomorphic to the input element we were seeking. There are certainly many more properties of `bst-search` we might wish to prove. The most notable is probably that if `bst-search` returns `nothing`, then indeed, the value really is not in the tree. One issue with proving that property is that we would first have to formalize really not being in the tree. This could be done in several ways. One idea would be to write a function that converts a `bst` to a list of `As`, and then state that the element is not in the list.

5.4.4 Inserting an element into a binary search tree

Inserting an element into a `bst` is somewhat more involved, because we have some work to do to maintain the binary search tree property, as required by the `bst` datatype. The first question we have to think about is, what should the type for `bst-insert` be? We will be give a value `d` (of type `A`), and a tree of type `bst l u`, for some lower and upper bounds `l` and `u` on the values in the tree. What should the return type of `bst-insert` be? It is tempting to think maybe it should just be `bst l u`, but that is too simplistic: what if `d` does not fall within the bounds of the tree? That is, what if `d` is less than `l` or greater than `u`? Then we will not be able to insert `d` while keeping `l` and `u` as the bounds on the values in the tree. One way to deal with this issue would be to require that `d` is within those bounds. But this would be short-sighted, as then we could never grow a tree beyond its current bounds. Instead, we will return a tree whose bounds have been updated to account for the possibility that the element `d` that we inserted was outside the bounds. So the return type we will take for `bst-insert` is `bst (min d l) (max d u)`. The complete typing for the function is then:

```
bst-insert : ∀{l u : A} (d : A) → bst l u →
            bst (min d l) (max d u)
```

For this to make sense, we must first declare `min` and `max` functions based on the ordering `_≤A_` which is a parameter to this module. For this, the IAL has code already in `minmax.agda`. The `minmax` module is parameterized by a type `A` together with an ordering `_≤A_` which is a total preorder. The definitions of `min` and `max` (from `bst.agda`) are then given in Figure 5.14, using `if_then_else_` and the ordering. Some easily proved properties from `minmax.agda` are listed in Figure 5.15; these are used in the code for `bst-insert` below.

Before we look at the code for `bst-insert`, there are two helper functions in `bst.agda` we should consider. These are functions for weakening the bounds on a `bst`. If we have a value of type `bst l u`, and if we have a smaller lower bound `l'` or a larger upper bound `u'`, then the helper functions `bst-dec-lb` and `bst-inc-ub`, shown in Figure 5.16, can be used to change, in effect, the type of the tree to `bst l' u` or `bst l u'`. These helper functions are easily defined, making use of `≤A-trans` (transitivity of the ordering) in the cases where the tree is a `bst-node`.

Finally, we have everything we need to define `bst-insert`, in Figure 5.17. The heart of the algorithm is exactly as expected: if the input tree is a leaf, we create a new node holding the value `d` which is being inserted, where the two subtrees are just leaves (i.e., empty). If the tree into which we are inserting `d` is a node, then we recursively insert into either the left or the right subtree, depending on whether or not `d ≤A d'`, where `d'` is the value at the current node. The only difference here is that we are using `bst-dec-lb` and `bst-inc-ub` to adjust the lower and upper bounds, respectively, for the subtrees where we do not recursively insert `d`. This is needed to change their bounds to use the `min` and `max` functions, to match the expected return type of the function. For example, when `d ≤A d'`, the

```

min : A → A → A
min = λ x y → if x ≤A y then x else y

max : A → A → A
max = λ x y → if x ≤A y then y else x

```

Figure 5.14: Definitions of min and max functions in terms of the posited ordering

```

min-≤1 : ∀{x y : A} → min x y ≤A x ≡ tt
min-≤2 : ∀{x y : A} → min x y ≤A y ≡ tt
max-≤1 : ∀{x y : A} → x ≤A max x y ≡ tt
max-≤2 : ∀{x y : A} → y ≤A max x y ≡ tt

min2-mono : ∀{x y y' : A} → y ≤A y' ≡ tt →
  min x y ≤A min x y' ≡ tt
max2-mono : ∀{x y y' : A} → y ≤A y' ≡ tt →
  max x y ≤A max x y' ≡ tt

```

Figure 5.15: Some properties proved about min and max

right subtree R has type $\text{bst } d' .u'$, where $.u'$ is an (implicit) upper bound. Using `bst-inc-ub`, we can adjust this upper bound to $\text{max } d \ u$, as required. We omit further consideration of the details, but you can investigate by temporarily replacing some terms with holes, to see what we have and what Agda is expecting at various places in the code.

5.5 Discussion: Internal vs. External Verification

When should one use internal verification – where datatypes are defined with invariants, and functions take in proofs of preconditions – and when is external verification – where theorems about functions are proved separately – more appropriate? There is certainly an element of taste involved, and both approaches can make sense in the same situation. But here are some general considerations that can hopefully serve as a guide:

Algebraic properties are usually proved externally. Say one is proving that an operator is associative (like addition on natural numbers, or append on lists). There does not seem to be any natural way (or really any way at all that I know of) to

```

bst-dec-lb : ∀ {l l' u' : A} → bst l' u' →
            l ≤A l' ≡ tt → bst l u'
bst-dec-lb (bst-leaf p) q = bst-leaf (≤A-trans q p)
bst-dec-lb (bst-node d L R p1 p2) q =
    bst-node d L R (≤A-trans q p1) p2

bst-inc-ub : ∀ {l' u' u : A} → bst l' u' →
            u' ≤A u ≡ tt → bst l' u
bst-inc-ub (bst-leaf p) q = bst-leaf (≤A-trans p q)
bst-inc-ub (bst-node d L R p1 p2) q =
    bst-node d L R p1 (≤A-trans p2 q)

```

Figure 5.16: Helper functions to change the bounds of a bst

```

bst-insert : ∀ {l u : A} (d : A) → bst l u →
            bst (min d l) (max d u)
bst-insert d (bst-leaf p) =
    bst-node d (bst-leaf ≤A-refl) (bst-leaf ≤A-refl)
    min-≤l max-≤l
bst-insert d (bst-node d' L R p1 p2) with keep (d ≤A d')
bst-insert d (bst-node d' L R p1 p2) | tt , p
    with bst-insert d L
bst-insert d (bst-node d' L R p1 p2) | tt , p | L'
    rewrite p =
    bst-node d' L'
        (bst-inc-ub R (≤A-trans p2 max2-≤A))
        (min2-mono p1)
        ≤A-refl
bst-insert d (bst-node d' L R p1 p2) | ff , p
    with bst-insert d R
bst-insert d (bst-node d' L R p1 p2) | ff , p | R'
    rewrite p =
    bst-node d' (bst-dec-lb L p1)
        R' min2-≤A (max2-mono p2)

```

Figure 5.17: Inserting an element into a binary search tree

formulate this property as an internal property of the operator. We have to define the operator, and then separately state and prove the algebraic property of it. This is an example of external verification.

Internal verification can be applied for essential invariants of datatypes. Some datatypes have essential invariants that no reasonable use of the datatype will ever violate. We have seen two examples in this chapter: the balancing property of Braun trees, and the data ordering property of binary search trees. Any use of these data structures must respect those properties, or else it is a different data structure which is really being used. It makes sense, therefore, to enforce those properties statically using internal verification, as we saw above. This certainly takes additional effort, and so in some cases one may choose not to do this for a data structure, even if it has an essential invariant. But essential datatype invariants are good candidates for internal verification.

Internal verification can be easier to apply for complex programs. Because internal verification weaves proofs of properties through code and datatypes, it can be easier to use it for complex programs, particularly when the property to be verified is relatively simple compared to the program in question. For if we have to prove a simple theorem externally about a complex program, we will likely not avoid reasoning about some aspects of that complexity which are not relevant to our property. At the very least, we will likely have to follow the patterns of case-splitting and recursion that the program uses, and those could be complicated. With internal verification, we just have to feed proofs through complex sections of code, without necessarily reasoning about the behavior of those sections.

External verification is often needed for functions used in an internal verification's specification. When using internal verification for a program, one will typically invoke some functions in the types of internally verified programs. An example is the use of `min` and `max` in the type of the insertion function for binary search trees (Section 5.4.4). These functions are being used specificationally, to help describe the intended behavior of the internally verified function. One typically finds that doing the internal verification will then require some externally proved properties of these specificationally used functions. This was indeed true for `min` and `max` in the case of the `bst-insert` function, where several properties of these were needed in the course of writing the code for `bst-insert`.

5.6 Conclusion

In this chapter, we have seen several examples of what is known as internal verification: invariants are expressed as part of the definition of a datatype and/or operations on that datatype, and proofs are then interspersed through the code, to establish those invariants. We considered vectors, where the type of the vector includes its length; Braun trees, with the balancing property that the left subtree of each node has the same size as the right, or is one greater; and binary search trees,

where we statically enforce the binary search tree property that all data in the left subtree of each node are less than or equal to the data at the node, and all data in the right are greater than or equal. Internal verification is often a lighter way to establish certain properties of programs than proving theorems externally about them. It is one of the hallmarks of dependently typed functional programming.

5.7 Exercises

Matrices

- Using the vector type \mathbb{V} in a nested fashion, fill in the hole below to define a type for matrices of natural numbers, where the type lists the dimensions of the matrix:

```
_by_matrix : ℕ → ℕ → Set
n by m matrix = ?
```

- Define the following basic operations on matrices, using the definition you propose in the previous problem. You should first figure out the types of the operations, of course, and then write code for them (possibly using helper functions).
 - `zero-matrix`, which takes in the desired dimensions and produces a matrix of those dimensions, where every value in the matrix is zero.
 - `matrix-elt`, which takes in an `n by m matrix` and a row and column index within those bounds, and returns the element stored at that position in the matrix.
 - `diagonal-matrix`, which takes in an element `d` and a dimension `n`, and returns the `n by n matrix` which has zero everywhere except `d` down the diagonal of the matrix. Use this to define a function


```
identity-matrix
```

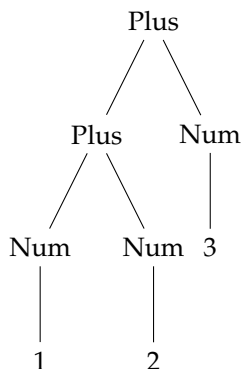
 returning a diagonal matrix where the diagonal is 1.
 - `transpose`, which turns an `n by m matrix` into a `m by n matrix` by switching the rows and columns.
 - `_·_`, the dot product of two vectors.
 - `_*matrix_`, which multiplies an `n by k matrix` and a `k by m matrix` to obtain a `n by m matrix`.
- `vector.agda` contains functions $\mathbb{V} \rightarrow \mathbb{L}$ and $\mathbb{L} \rightarrow \mathbb{V}$ for converting between vectors and lists. State and prove a theorem expressing the idea that converting a vector to a list and then back to a vector results in the same vector.

4. Write a function which takes a vector of type $\mathbb{V} (A \times B) \ n$ and returns a pair of vectors, one of type $\mathbb{V} A \ n$ and another of type $\mathbb{V} B \ n$. This is similar to the unzip function in Haskell, only with vectors instead of lists.
5. Implement `remove-min` and `remove-max` functions for the `bst` type. Using `remove-min`, you can then write a general `remove` function, that finds the first value isomorphic to a given one, and returns the `bst` without that value. For this, following the standard algorithm, it will be necessary, if the node holding the value has two (non-leaf) nodes as the left and right subtrees, to replace the removed element with its *successor*. This is the minimum value in the right subtree.
6. In `list-merge-sort.agda` in the IAL, there is an implementation of `merge-sort` using Braun trees. State and prove some theorems about this `merge-sort` function. One that hopefully would not be too bad would be to prove that the length of the input list and the length of the sorted list returned by `merge-sort` are the same.

Chapter 6

Generating Agda Parsers with `gratr`

In this chapter, we will see how to use the `gratr` tool, which I have developed together with John Bodeen and Nao Hirokawa, for parsing textual input into Agda. Parsing is an important step in any language-processing tool: we must go from a linear sequence of characters (stored in a file, or possibly streamed into the program), to a structured representation in memory, for manipulation by the program. For example, we might like to go from textual input like "1 + 2 + 3" to a structured representation like



Such a representation provides a good starting point for further processing, because it has made explicit the syntactic structure which is implicit in the linear textual input. For example, if this is indeed how we choose to parse "1 + 2 + 3", we see from the tree that `Plus` is a left-associative operator: the tree has group 1 and 2 more closely under an inner `Plus` node, leaving 3 to be grouped additionally after that.

In this chapter, we will see how we can generate Agda parsers from context-free grammars augmented with rewrite rules, using `gratr`. This `gratr` tool takes a grammar as input, and produces Agda code which can parse strings in the language described by the grammar. Context-free grammars are a formalism for describing the syntax of languages. They have a rich and interesting theory, which is, however, beyond the scope of the present book to study. Our focus here will be on writing grammars and using `gratr` to generate parsers from them. For extensive information on the many, many other methods which exist for parsing, see [8]. For an introduction to context-free grammars and related formalisms from *formal language theory*, see [13].

The chapter first begins with a basic introduction to context-free grammars (Section 6.1), and then discusses how to use the `gratr` parser generator to generate Agda code for parsing input strings (Section 6.2).

6.1 A Primer on Grammars

A context-free grammar consists of a set of *productions*, which look like:

$$\text{symbol} \rightarrow \text{symbol}_1 \cdots \text{symbol}_n$$

The right-hand side of a production is allowed to be empty (no symbols at all). The symbol on the left-hand side of the production cannot be an input character (like 'a'). It is required to be a **nonterminal**, which is just a symbol introduced by the grammar for purposes of describing some part of the syntax of the language. The symbols on the right-hand side of the production can either be nonterminals or input characters, which are also called **terminals**. A context-free grammar also must designate one of the nonterminals as the **start symbol** of the grammar. The strings we parse are then required to be ones generated (as will be explained below) by this start symbol.

Any software tool working with grammars must, of course, choose some particular syntax for grammars. An example of a grammar in `gratr` format is given in Figure 6.1. The grammar begins with its name, which is `prefix`. Then we have the `Start` keyword, followed by the nonterminal which is the start symbol. In this case, I chose to call that nonterminal `start`, but any other name is allowed. Then there is the `Syntactic` keyword, followed by some productions. A period ('.') is required after every production. Also, each syntactic production may optionally have a name, which comes before a colon and the rest of the production. So `Prefix` is the name of the first syntactic production of Figure 6.1. Input characters or strings of characters should be enclosed in single quotation marks. To indicate a single quotation mark itself, you write `quote`. Similarly, to indicate a single space, you write `space`. Spaces cannot otherwise be used in strings of characters.

The example grammar in Figure 6.1 contains some syntactic productions. The parsers generated by `gratr` will build parse trees for those productions. Sometimes, however, one does not want to build a structured representation of the input text. For a number like "123", usually we just want to remember the string that we recognized, rather than build some explicit tree structure containing each of the characters of the string separately. In such cases, you can use lexical productions, like the ones listed beneath the `Lexical` keyword for the `num` nonterminal. Using a thin arrow (`->`) in a lexical production means that we want to retain the string that we recognize as being in the language of the given nonterminal. These are called **reading** lexical productions in `gratr`. Using a thick arrow (`=>`) indicates that we just want to recognize the string, but not retain it in our parse tree. This example grammar is using such **recognizing** lexical productions for `maybe-nl`.

```

prefix

Start

start

Syntactic

Prefix : start -> prefix maybe-nl.
Plus : prefix -> '+' prefix space prefix.
Num : prefix -> num .

Lexical

num -> '0'.
num -> '1'.
num -> '2'.

maybe-nl => '\n'.
maybe-nl => .

```

Figure 6.1: An example grammar in `gratrr` format

6.1.1 Derivations

How does a grammar like the one in Figure 6.1 describe a language? Mathematically, the definition is very simple. The language is the set of input strings which can be generated from the start symbol by applying the productions to replace symbols with sequences of symbols. For example, the input string `+1 +2 3` is in the language described by the `prefix` grammar, as shown by the following **derivation**:

```

start =>
prefix maybe-nl =>
'+' prefix space prefix maybe-nl =>
'+' prefix space '+' prefix space prefix maybe-nl =>
'+' num space '+' prefix space prefix maybe-nl =>
'+' '1' space '+' prefix space prefix maybe-nl =>
'+' '1' space '+' num space prefix maybe-nl =>
'+' '1' space '+' '2' space prefix maybe-nl =>
'+' '1' space '+' '2' space num maybe-nl =>
'+' '1' space '+' '2' space '3' maybe-nl =>
'+' '1' space '+' '2' space '3' =
'+1 +2 3'

```

A derivation shows the step-by-step transformation of the start symbol into the string in question. At each step, we are allowed to replace a single nonterminal N ,

with ss , where there is a production $N \rightarrow ss$ in the grammar. So we are using the grammar to transform nonterminals matching the left-hand sides of productions into the production's right-hand sides.

6.1.2 From derivations to syntax trees

An alternative to using derivations is to use **derivation trees**. These are trees whose root is the start symbol, and where each node is either a terminal symbol (i.e., input character) or else is labeled with a nonterminal N and has children labeled s_1, \dots, s_n , where there is a production $N \rightarrow s_1, \dots, s_n$ in the grammar. For example, Figure 6.2 shows the derivation tree corresponding to the derivation in the previous section. The original input string `'+1 +2 3'` can be found by reading the leaves of the tree from left to right. Derivation trees are based on the same idea as derivations: replace nonterminals with sequences of symbols, according to the productions of the grammar. The difference is that derivations show all the intermediate strings that result on the way from transforming the start symbol to a particular string in the language of the grammar. In contrast, derivation trees just show how the productions are applied, without showing all the intermediate strings. This is a more compact representation in general, and makes the tree structure more explicit.

It is usually very helpful, when processing parsed input, to know exactly which production was applied to create a given node of the tree. Derivation trees do not make this information as explicit as they could: at every node labeled N with children labeled s_1, \dots, s_n , we could determine which production was applied, but only by searching through the list of productions for one of the form $N \rightarrow s_1, \dots, s_n$. It is more efficient just to store some identifying information for the production itself, at the node in question. This is what is done in **syntax trees**. They are just like derivation trees, except that nodes which would have been labeled with nonterminals in a derivation tree are instead labeled with the name of the production that was applied to create the node. An example is in Figure 6.3.

Finally, there is some information in the syntax tree that is redundant, in the sense that it is completely determined by the production which was applied. This is the case with `'+'` in the tree in Figure 6.3, for example. Once we see that a node is a `Plus` node, we know that the input string included the actual `'+'` terminal. There is no need to store that terminal as part of the syntax tree. Similarly, grammar designers often choose to drop whitespace from their syntax trees. We can also remove the nodes which just give the names of lexical nonterminals, as they are always followed in the tree by either strings or nothing; and which lexical nonterminals they are is determined by the label of the parent node. This kind of clean-up – whether done by special-purpose code or, as in `gratr`, as part of parsing – results in an **abstract syntax tree**, or AST. For the sample string `'+1 +2 3'`, the AST obtained using the grammar of Figure 6.1 is given in Figure 6.4.

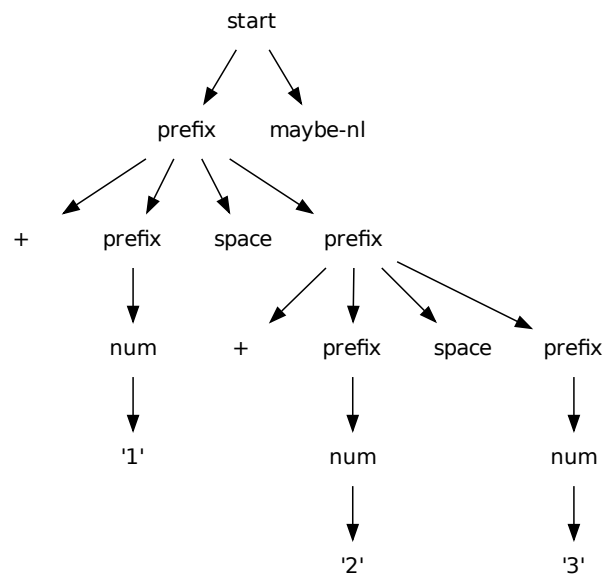


Figure 6.2: An example derivation tree

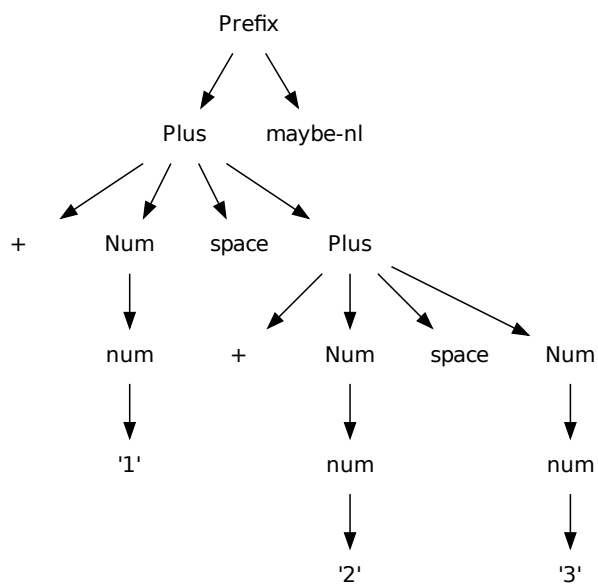


Figure 6.3: An example syntax tree

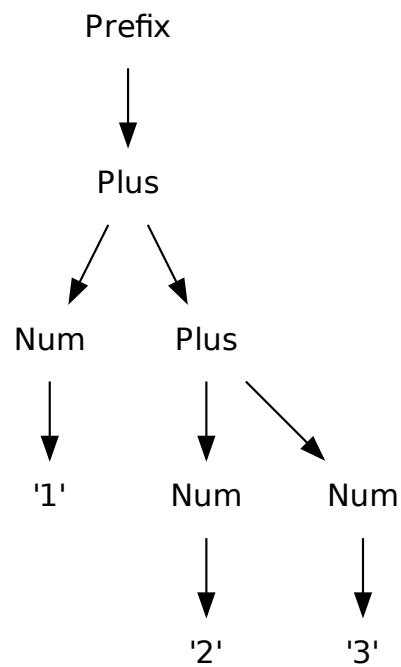


Figure 6.4: An example abstract syntax tree

6.1.3 Regular-expression operators for grammars

`gratr` supports the use of certain extra operators, called regular-expression operators, on the right-hand sides of productions. These operators may be applied to individual symbols (either input symbols or nonterminals), or to groups of symbols and operator uses, enclosed in parentheses. These extra operators are:

operator use	meaning
N^*	zero or more copies of N
N^+	one or more copies of N
$N^?$	zero or one copy of N (so N is optional)
$N \mid M$	either N or M

Using these operators, it is easy to modify our example grammar so that numbers are more than just `'0'`, `'1'`, and `'2'`. We could just as well add additional productions for `num` to accomplish this, but it is more convenient to use the regular-expression operators. In fact, it is not hard to translate away any use of a regular-expression operator, by introducing extra nonterminals. For example, if we have $N^?$, we can replace it by a new nonterminal M , adding these new productions:

```
M -> N .
M -> .
```

These say that an M can be an N or else nothing – exactly the meaning of $N^?$. The updated grammar is shown in Figure 6.5. This grammar is included with `gratr` in this file with the top-level `gratr` directory:

```
grammars/expression/prefix.gr
```

Currently, it is probably best to limit use of regular-expression operators to lexical productions, because for syntactic productions, `gratr` has to generate names for the productions (like those shown just above for $N^?$), and it is usually best to avoid writing code to process parse trees containig generated names. A better approach for `gratr` would be to generate lists of parse trees for strings matching N^* and N^+ , and similarly for $N^?$ and $N \mid M$. But this is not implemented in `gratr` currently.

`gratr` also supports **character ranges**. You can write `['a' - 'z']` for all the lowercase letters, and similarly `['A' - 'Z']` for all the capitals, or `['0' - '9']` for the decimal digits. These can also be combined, so `['a' - 'c' 'x' - 'z']` matches characters `a`, `b`, `c`, `x`, `y`, `z`. Unfortunately, while `gratr` does support Unicode characters as terminal symbols, character ranges do not work with these.

6.2 Generating Parsers with `gratr`

The `gratr` tool may be downloaded using `subversion` from the following URL:

```
https://svn.divms.uiowa.edu/repos/clc/projects/gratr2
```

```
prefix

Start

start

Syntactic

Prefix : start -> prefix maybe-nl.
Plus : prefix -> '+' prefix space prefix.
Num : prefix -> num .

Lexical

num -> ['0' - '9']+.

maybe-nl => '\n'?.
```

Figure 6.5: An example grammar in `gratr` format, using regular-expression operators

The username and password are both “guest” (without the quotation marks). `gratr` is written in the OCaml functional programming language, which you will need to install on your computer to compile `gratr`. See the `README` file in the `gratr2` directory for instructions on compiling on Windows, Mac, and Linux. Once you have compiled `gratr`, you can run it by just giving it the name of a file containing a grammar like the one in Figure 6.5, as the sole command-line argument. If everything runs correctly, then `gratr` will print nothing. Otherwise, it will print a short error message, with possibly a lot of detailed explanation in a report file that gets generated. For a grammar named `prefix`, the report file will be written to `prefix_report.txt`.

When `gratr` runs successfully on a file like `prefix.gr` defining a grammar named `prefix`, it will produce a number of different files:

- `prefix.agda`: this Agda file contains the central code for parsing input text according to the given grammar. You will not need to look at the contents of this file to use `gratr` to parse inputs.
- `prefix-types.agda`: this file contains datatype definitions derived from the grammar, which you will likely wish to consult when writing programs to process parse trees obtained by parsing textual input. For each nonterminal defined by syntactic productions, `gratr` will declare an Agda datatype. The constructors of this datatype are the names of the productions for that nonterminal. For example, for the `prefix` grammar (of Figure 6.5), `gratr` will generate these datatype definitions:

```

data start : Set where
  Prefix : prefix → start

data prefix : Set where
  Num : num → prefix
  Plus : prefix → prefix → prefix

```

There is one datatype definition for each of the two syntactic nonterminals in the grammar: `start` and `prefix`. There is one constructor for `start`, for the sole production for that nonterminal. And there are two constructors for `prefix`, for the two productions `Num` and `Plus` from the grammar for that nonterminal. `gratr` looks at the right-hand sides of the productions to determine the arguments to the constructors. For example, the `Plus` production looks like this:

```
Plus : prefix -> '+' prefix space prefix.
```

There are four symbols on the right-hand side, but the Agda datatype constructor `gratr` accepts only two arguments, both of type `prefix`. The reason is that `gratr` does not add arguments for input symbols to the datatype constructors. Here, `'+'` and `space` (denoting an actual space character `' '`) are input symbols, and hence are not included as arguments to the `Plus` constructor. Note that `gratr` will also omit recognizing lexical nonterminals from the list of arguments to the Agda datatype constructors.

- `prefix-main.agda`: this defines the entry point to the actual executable program, `prefix-main`, which can be used to parse input text. It contains a function called `process-X`, where `X` is the start symbol for your grammar. If the parser is given textual input which can indeed be parsed, then this function will be called with the parse tree associated with the start symbol. The function may then do whatever you like with that parse tree. In the end, the function should return a string, which the surrounding code in `prefix-main.agda` will print out before it exits.
- `prefix_report.txt`: this is a text file giving a rather large amount of information about what `gratr` is doing. At the top of the file, you will find a modified version of the grammar `gratr` has been asked to process (`prefix.gr` in this case), with all the extra grammar operators (Section 6.1.3) removed.

6.2.1 Compiling the emitted parsers

To compile the emitted parser for a grammar like `prefix.gr`, you can call Agda from a command shell (Windows) or terminal (Mac/Linux). Here is an example of what this call to Agda has to look like:

```
agda -i PATH-TO-IAL -i PATH-TO-GRATR/agda -i . -c prefix-main.agda
```

There are several components here:

- The `-i` arguments are telling Agda where to search for Agda files which are imported in the code being compiled. For `-i PATH-TO-IAL`, you should replace `PATH-TO-IAL` with the path to your copy of the IAL, on your computer. Similarly, replace `PATH-TO-GRATR` with the path to your copy of `gratr`. The directory for `gratr` contains an `agda` subdirectory with some additional Agda code needed for parsing.
- The `-c` directive tells Agda we want to compile the given Agda file – with a `main` entry point – to an actual executable.

Compilation using the above command can take a while, depending on how big the `prefix.agda` file is. Compilation will produce a `prefix-main` executable.

To check a file like `prefix-main.agda` using Agda within Emacs, you must either copy into the same directory as your `prefix-main.agda` file all the `.agda` files from the `agda` subdirectory of the `gratr` directory; or else (and this is a bit nicer), just add the path to that `agda` subdirectory of the `gratr` directory to the list `agda2-include-dirs` in your `.emacs` file (similarly to what is listed at the very end of the emacs code shown in the “Some Extra Emacs Definitions” section at the end of the book).

6.2.2 Running the emitted executable

To run the executable compiled as just described, you supply the name of a file as a command-line argument. This is the file that the emitted parser will attempt to parse. For example, if `prefix-yes1.txt` is a file containing a string like `+1 +1 2`, then you can parse it with `prefix-main` by executing the following command from a command shell or terminal:

```
prefix-main prefix-yes1.txt
```

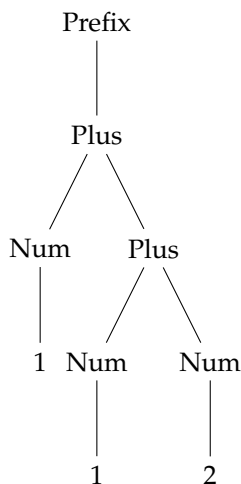
If parsing is successful (as it will be in this case), there will be no output from executing this command. If you run with the `--showParsed` command-line option, then you will see the actual parse tree; executing

```
prefix-main --showParsed prefix-yes1.txt
```

will produce this output:

```
(Prefix (Plus (Num (num 1)) (Plus (Num (num 1)) (Num (num 2)))))
```

This is the linear form of the following parse tree:



6.2.3 Modifying the emitted code to process parse trees

Once you have crafted a grammar and are correctly recognizing or rejecting certain input strings, the next step is to be able to process those parse trees with your own Agda code. The parsers emitted by `gratr` have a place where you can insert your own code to process parse trees that the parser constructs. This is the function `process-X`, where `X` is the start symbol of the grammar. For the prefix grammar of Figure 6.5, the function is `process-start`. If you look in `prefix-main.agda`, you will see the current placeholder definition:

```
process-start : start → string
process-start s = ""
```

The type of `process-start` says that it takes in a parse tree of type `start` (declared in `prefix-types.agda` based on the nonterminals and production names in the grammar), and returns a `string`. This string is whatever we wish to report to the user invoking the `prefix-main` executable, and will be printed out by the executable after `process-start` returns.

To do something with parse trees recognized by the parser, we just need to modify `process-start`. It is wisest first to copy the file, say to `my-prefix-main.agda`. This is because `gratr` will overwrite `prefix-main.agda` every time `gratr` is run. So if we make changes to our grammar and rerun `gratr`, we will regenerate `prefix-main.agda`. If we have been working on this file and making modifications to it to add our own form of processing for parse trees, our work will get wiped out – not something we want! So just make a copy of the file, like `my-prefix-main.agda`, change the module declaration at the very start of the file to name the module `my-prefix-main`, and then make your changes to that file (which will not, of course, get overwritten by `gratr` when `gratr` is rerun).

For an example, suppose we wish to print out the size of the parse tree. We already saw (at the start of Section 6.2) the parse-tree datatypes that are generated by `gratr` for `prefix`, in `prefix-types.agda`:

```
data start : Set where
  Prefix : prefix → start

data prefix : Set where
  Num : num → prefix
  Plus : prefix → prefix → prefix
```

To compute the size of a `Prefix` value, we can write this function in Agda, in `my-prefix-main.agda`:

```
size-prefix : prefix → ℕ
size-prefix (Num _) = 1
size-prefix (Plus p1 p2) = 1 + (size-prefix p1) + (size-prefix p2)
```

Note that this is computing the size of the parse tree, not the numeric value of the expression denoted by that parse tree. Then we just modify the `process-start` function so it uses pattern-matching to extract the `prefix` parse tree `p` from under the `Prefix` constructor (for the `start` datatype), get the size of `p`, convert that to a string (with the library function `ℕ-to-string`, defined in `nat-to-string.agda` in the IAL), and concatenate that (using `^`) to the end-of-line string `"\n"`:

```
process-start : start → string
process-start (Prefix p) = (ℕ-to-string (size-prefix p)) ^ "\n"
```

6.2.4 Reorganizing rules to resolve ambiguity

There is one somewhat unique feature of `gratr` that we will consider now, called *reorganizing rules*. To explain what these are and when they are useful, let us consider the example grammar in Figure 6.7. This grammar describes a simple language of arithmetic expressions, built from numeric literals using plus and times operators, and parentheses. Optional whitespace (`ows`, defined with lexical rules) is allowed throughout the expression. An example string in the language of this grammar is

```
1 + 2 * 3
```

If we ask `gratr` to process this grammar for us, it will print this error message:

```
The parsing rules have critical pairs that need additional
reorganizing rules to resolve (see expr_report.txt for details).
```

Looking at the bottom of the `expr_report.txt` file that `gratr` generates, we find messages like the one shown in Figure 6.6. `gratr` is telling us that there are two ways we could parse a string consisting of three `exprs` separated by plus symbols (and optional whitespace). We could either parse it to `Plus (Plus (x3, x0), x1)`,

```

(We have two ways to rewrite the following term:
  expr(x3) . ows . '+' . ows . expr(x0) . ows . '+' . ows . expr(x1) . x2
We could have this rewriting sequence
(
  expr(x3) . ows . '+' . ows . expr(x0) . ows . '+' . ows . expr(x1) . x2
  expr(Plus(x3, x0)) . ows . '+' . ows . expr(x1) . x2
  expr(Plus(Plus(x3, x0), x1)) . x2
)
or this one:
(
  expr(x3) . ows . '+' . ows . expr(x0) . ows . '+' . ows . expr(x1) . x2
  expr(x3) . ows . '+' . ows . expr(Plus(x0, x1)) . x2
  expr(Plus(x3, Plus(x0, x1))) . x2
)

The overlapping rules are:
  Plus: expr(x3) . ows . '+' . ows . expr(x4) . x5 -> expr(Plus(x3, x4)) . x5
  Plus: expr(x0) . ows . '+' . ows . expr(x1) . x2 -> expr(Plus(x0, x1)) . x2
)

```

Figure 6.6: Error message with a critical pair, produced by `|gratr`— for the grammar of Figure 6.7

where `x3`, `x0`, and `x1` are the parse trees for the three `exprs` in order; or else we could parse it to `Plus(x3, Plus(x0, x1))`. What we are encountering here is a classic case of grammatical **ambiguity**: there are two distinct derivation trees for the string in question. One tree is left-associated (`Plus(Plus(x3, x0), x1)`), and the other is right-associated (`Plus(x3, Plus(x0, x1))`). Without further direction, `gratr` (and even the mathematically uninstructed reader) does not know whether sequences of additions should be grouped to the left or right. Another way to state the issue: should we parenthesize $1 + 2 + 3$ as $(1 + 2) + 3$ or as $1 + (2 + 3)$? Of course, addition is associative (as we proved back in Section 3.4), so both groupings yield the same answer. But this sort of *semantic* information is not available at the point where we are just describing the *syntax* of the language.

Generally, grammar writers have to resolve ambiguities like this somehow. `gratr` provides an interesting method to do this. We can add rules of a certain kind, to tell `gratr` which syntax tree we prefer. The rules tell `gratr` to rewrite parse trees matching a pattern, to a certain result form. For example, to tell `gratr` that we want plus to be left-associated, we will add this rule:

```
Plus(x, Plus(y, z)) -> Plus(Plus(x, y), z)
```

Any parse tree that matches that pattern on the left will be transformed to reorganize a right-nested addition so that it becomes left-associated. Figure 6.8 shows the complete set of reorganizing rules needed to make both plus and times left-associative, and give times higher precedence than plus (so that $1 + 2 * 3$ is parsed as $1 + (2 * 3)$). There is a `VARS` section to declare the variables `x`, `y`, and `z` used in the reorganizing rules. Then there is a `RULES` section, to give the reorganizing rules. These two sections come, optionally, at the end of `.gr` grammar files processed by `gratr`.


```

expr

Start

strt

Syntactic

Strt: strt -> ows expr ows.
Plus: expr -> expr ows '+' ows expr.
Times: expr -> expr ows '*' ows expr.
Parens: expr -> '(' ows expr ows ')' .
Num : expr -> num.

Lexical

aws => ('\n' | '\t' | space ).
ws => aws+.
ows => aws* .

num -> ['0'-'9']+.

```

Figure 6.7: Sample grammar for basic arithmetic expressions

```

Vars

x y z

Rules

Times(x, Plus(y, z)) -> Plus(Times(x, y), z)
Plus(x, Plus(y, z)) -> Plus(Plus(x, y), z)
Times(x, Times(y, z)) -> Times(Times(x, y), z)
Times(Plus(x, y), z) -> Plus(x, Times(y, z))

```

Figure 6.8: Addendum to the grammar in Figure 6.7, with reorganizing rules

6.3 Conclusion

This chapter has given a brief introduction to context-free grammars, and shown how to use the `gratr` parser generator to generate Agda parsers for grammars represented in `gratr`'s grammar format. This format supports regular-expression operators, and distinguishes between syntactic productions, reading lexical productions, and recognizing lexical productions. Parsing with syntactic productions will construct parse trees. Reading lexical productions are for remembering the string parsed but not constructing a parse tree for it. Recognizing lexical productions are just for recording that the input string matched, without storing any other information (neither the string that matched nor a parse tree for it). These features make it reasonably convenient to describe abstract syntax trees directly with the grammar used for parsing. Such syntax trees may drop some of the information from the input string, for the sake of simpler parse trees. A final note is that `gratr` is the subject of ongoing research, and hence fuller explanation of its workings must be deferred to eventual papers or a `gratr` reference manual.

6.4 Exercises

1. Create a grammar called `postfix.gr` (the name of the grammar given at the start of the file should be `postfix`) for the language of postfix addition expressions and nonnegative integers (leading zeros are fine), possibly followed by a newline, and with a single space just in between the arguments to addition (+).
2. Some functional languages like OCaml and Haskell allow one to write lists like `1 :: 2 :: 3 :: []` using the notation `[1, 2, 3]`. For this problem, create a grammar called `plll.gr` (and the name in the file should be `plll`) describing what we can call the pure language of list literals:
 - the empty list `[]` is a list literal
 - if `L` is a comma-separated sequence of list literals like `[]`, `[], []` (for example), then `[L]` is also a list literal
 - whitespace (spaces, tabs, and newlines) is allowed anywhere, except in `[]`
 - there are no other list literals (this is what I mean by “pure”)
3. Write a grammar called `app.gr` describing nested applications. Each of these is either a variable (we will say these consist of just lowercase letters), an application of one expression to another with whitespace in between, or a parenthesized expression. Whitespace (spaces, tabs, newlines) is allowed everywhere, and application is left-associative. For this problem, you should

generate parse trees using constructors `Var`, `App`, `Parens`. Use the constructor `Term` for the entire expression you parse. So the test input string `(f a b)` should parse as follows, when you run your generated parser with `--showParsed`:

```
(Term (Parens (App (App (Var (var f)) (Var (var a)))
                  (Var (var b)))))
```

4. Create a grammar called `para.gr` for describing strings which are sequences of *paragraphs*. Each paragraph consists of a sequence of at least one *sentence*, where a sentence begins with a capitalized word, followed by possibly zero other words, which may or may not be capitalized. Sentences can also contain commas, colons, and semi-colons. A sentence ends with a period or a question mark. Words are just sequences of letters. We will allow words to contain capital letters in the middle of the word, since this makes it easier to describe what words are. (We are not including numbers, quotations, parentheses, and a few other things as parts of sentences.) Words are separated by a single space or newline, although the various punctuation marks should come right after the word, with no space or newline. Words within a sentence which immediately follow a punctuation mark should be separated by a single space or newline from the punctuation mark. Sentences should be separated by one or two spaces, or else a newline, and paragraphs should be separated by two or more newlines. There should not be any whitespace before or after the sequence of paragraphs, except that it is allowed to end the last paragraph with a final newline. Your grammar should not try to enforce any other requirements (for example, subject-verb agreement, proper uses of colons and semicolons, etc.).

Tip: I suggest starting small (words), and working your way up (then sentences, then one paragraph, then multiple paragraphs). If you try to write the whole grammar at once, it can be hard to debug.

Chapter 7

Type-level Computation

In previous chapters, we have seen how to write proofs about programs (external verification) and program with proofs (internal verification). Certainly, the ability to write proofs is one of the features which makes Agda very different from mainstream programming languages like Python or Java. In this chapter, we will look at a second such feature, which is the ability to compute types by pattern matching and recursion on values. In some ways, this feature may be even more exotic than programming with proofs, because it changes the kinds of programs we can write (while maintaining static typing), rather than changing what we can prove statically about them. We will look at three examples related to type-level computation. For the first, we will use type-level computation as part of a datatype for the integers \mathbb{Z} (Section 7.1). For the second, we will give a type-safe version of formatted printing (C's `printf` function), following an idea of Augustsson's [2]. Finally, we will see an example which, in full generality often makes use of type-level computation (though here this will not be needed). This is **proof by reflection**, which we will use to write a verified simplifier for list expressions, which we could use instead of applying `rewrite` to apply various lemmas about list operations. A more complicated example involving type-level computation will be developed in Chapter 10.

7.1 Integers

How does one represent numbers beyond the naturals in a language like Agda? We are thinking here of representing numbers for purposes of theorem-proving, since as we have already noted (at the start of Chapter 3), efficient numeric representations would take some further work in Agda, which we are not going to attempt. In this section, let us look at how we might represent just the mathematical integers \mathbb{Z} . The development discussed here can be found in `integer.agda` in the Iowa Agda Library.

The basic initial question is, how should integers be represented as a datatype? There are several natural possibilities. Here, we will try out a particular one which will demonstrate the use of type-level computation. Roughly, we will view an integer as a natural number plus a sign bit; i.e., a boolean telling whether the number is positive or negative. So we could try using $\mathbb{N} \times \mathbb{B}$ as the definition of the \mathbb{Z} type. But there is an ugly little wrinkle: that definition would give us two

representations of 0. We would have a positive 0 and a negative 0, since every integer would have a sign bit. This may seem like a small inconvenience, but it will complicate all our subsequent theorems. 0 does not have a sign in traditional mathematics, and so working with signed 0 will require constant little deviations from usual theorems and definitions.

We can do better than this, using type-level computation. We want to express the idea that ~~only non-zero numbers have a sign~~, while 0 is unsigned. Alternatively, we will view non-zero numbers as having a sign of type \mathbb{B} , while 0 has a sign of type \top . This type \top is defined in `unit.agda`:

```
data  $\top$  : Set where
  triv :  $\top$ 
```

This type, also called the unit type, has a sole inhabitant, `triv`. So if you have an element of type \top , you have no information, as there is only one possibility for what element this could be (and hence, having a particular element resolves no uncertainty). We will define integers so that the sign bit for non-zero numbers is a boolean, while the sign bit for 0 is an element of type \top (in other words, there is no sign for 0).

7.1.1 The \mathbb{Z} datatype

Delightfully, this idea of having no sign bit for integer 0 can be implemented in Agda. The first step is to define a type-level function to compute the type for the sign bit:

```
 $\mathbb{Z}$ -pos-t :  $\mathbb{N} \rightarrow$  Set
 $\mathbb{Z}$ -pos-t 0 =  $\top$ 
 $\mathbb{Z}$ -pos-t (suc _) =  $\mathbb{B}$ 
```

If the input natural number is 0, then we return the unit type \top ; if it is a successor number, we return the boolean type \mathbb{B} .

Now we can define the datatype of integers, as follows:

```
data  $\mathbb{Z}$  : Set where
  mk $\mathbb{Z}$  : (n :  $\mathbb{N}$ )  $\rightarrow$   $\mathbb{Z}$ -pos-t n  $\rightarrow$   $\mathbb{Z}$ 
```

The sole constructor `mk \mathbb{Z}` for datatype \mathbb{Z} takes in a natural number n and then a second argument whose type is computed by the `\mathbb{Z} -pos-t` function from n . The `integer.agda` file then gives a couple useful representative examples, shown in Figure 7.1. `0 \mathbb{Z}` , the integer 0, is defined to be `mk \mathbb{Z} 0 triv`. This is exactly as we intended, since `triv` is the sole (trivial) constructor for the top type \top , which is what the second argument to `mk \mathbb{Z}` must be when the first argument is the natural number 0. The examples `1 \mathbb{Z}` and `-1 \mathbb{Z}` show that the second argument is a boolean

```

0ℤ : ℤ
0ℤ = mkℤ 0 triv

1ℤ : ℤ
1ℤ = mkℤ 1 tt

-1ℤ : ℤ
-1ℤ = mkℤ 1 ff

```

Figure 7.1: Some integer constants

when the first argument is nonzero. Note that we are intending `tt` to mean the integer is positive, and `ff` to mean it is negative.

7.1.2 Addition on integers

The `integer.agda` file in the IAL has a definition of addition, shown in Figure 7.2. We pattern match on the integer arguments. If either is zero, we return the other. Otherwise, we can compare the sign bits to decide what to do next. The sign bits are different if their exclusive or (`xor`) is `tt`, and they are the same if it is `ff`. In the latter case, we just add the natural number parts and use the same sign bit. In the former case, where the sign bits of the two numbers are different we will subtract one natural number part from the other, using a helper function `diffℤ`, shown in Figure 7.3. Which number we subtract from which is determined by `p1 imp p2` (the guard in the `if_then_else` term), which is `tt` if the first number is negative and the second positive; and `ff` if it is the second which is negative with the first positive (since the sign bits are different in this case, one of those two conditions must hold). The `diffℤ` function calls `ℕ-trichotomy` from `nat-thms.agda`, to obtain a proof of the following disjunction: either $n < m$, $n = m$, or $m < n$ is true (`tt`). There are then different cases for these three conditions. The code for `diffℤ` uses a second function from `nat-thms.agda`, with this typing:

$$\begin{aligned} <\div\text{suc} : \forall \{x \ y : \mathbb{N}\} \rightarrow (y < x \equiv \text{tt}) \rightarrow \\ &\quad \Sigma \mathbb{N} (\lambda n \rightarrow x \div y \equiv \text{suc } n) \end{aligned}$$

This takes in a proof that $y < x$ (for implicit x and y), and returns a natural number together with a proof the subtracting y from x gives you the successor of that natural number. In other words, this function does the subtraction and also gives you a proof that the result of the subtraction is nonzero. `diffℤ` uses this in order to apply the `mkℤ` constructor to the result, as well as a boolean sign bit.

```

_+Z_ : Z → Z → Z
(mkZ 0 _) +Z x = x
x +Z (mkZ 0 _) = x
(mkZ (suc n) p1) +Z (mkZ (suc m) p2) with p1 xor p2
(mkZ (suc n) p1) +Z (mkZ (suc m) p2) | ff =
  mkZ (suc n + suc m) p1
(mkZ (suc n) p1) +Z (mkZ (suc m) p2) | tt =
  if p1 imp p2 then diffZ m n else diffZ n m

```

Figure 7.2: Integer addition

```

diffZ : N → N → Z
diffZ n m with N-trichotomy n m
diffZ n m | inj1 p with <¬suc{m}{n} p
diffZ n m | inj1 p | x , _ = mkZ (suc x) ff
diffZ n m | inj2 (inj1 p) = mkZ 0 triv
diffZ n m | inj2 (inj2 p) with <¬suc{n}{m} p
diffZ n m | inj2 (inj2 p) | x , _ = mkZ (suc x) tt

```

Figure 7.3: Subtracting the second natural from the first to obtain an integer

7.1.3 Antisymmetric integer comparison

The example of integer addition showed some of the work we have to do with our representation based on computing the type for the sign bit. In this section, we will see an example of the payoff for doing this work. We will define a comparison operation on integers and prove it is antisymmetric. This would not have been possible, strictly speaking, if all integers had a boolean sign bit. For then we would have had positive and negative zero, which presumably would have had to be isomorphic (each less than or equal to the other). But then antisymmetry would fail, since that is exactly the property that isomorphic elements are equal.

The code for the integer comparison function is given in Figure 7.4. There are some obvious cases if one of the integers being compared is the integer for 0. The comparison if the other number is non-zero is then given by the sign bit of that other number. For the cases where both integers are nonzero, we again use the `xor` and `imp` operations on the signs, in the course of making the comparison. For this comparison function, `integer.agda` then has a proof of the following antisymmetry property:

```

≤Z-antisymm : ∀ (x y : Z) → x ≤Z y ≡ tt → y ≤Z x ≡ tt →
  x ≡ y

```

The proof is somewhat lengthy, so we omit it here. The main point is that if we had distinct representations of the same mathematical integer, this property would not


```

_≤ℤ_ : ℤ → ℤ → ℬ
(mkℤ 0 triv) ≤ℤ (mkℤ 0 triv) = tt
(mkℤ 0 triv) ≤ℤ (mkℤ (suc _) pos) = pos
(mkℤ (suc _) pos) ≤ℤ (mkℤ 0 triv) = ~ pos
(mkℤ (suc x) pos1) ≤ℤ (mkℤ (suc y) pos2)
  with pos1 xor pos2
(mkℤ (suc x) pos1) ≤ℤ (mkℤ (suc y) pos2) | tt =
  pos1 imp pos2
(mkℤ (suc x) pos1) ≤ℤ (mkℤ (suc y) pos2) | ff =
  if pos1 then x ≤ y else y ≤ x

```

Figure 7.4: Integer comparison

hold. So this is an example of the benefit of having a unique representation, like the one we have achieved using type-level computation.

7.2 Formatted Printing

While the previous example may seem a bit technical, the one we will consider now is concerned with very practical programming. The standard library for the C programming language has, for decades, contained a function `printf`, which takes a format string and some arguments, and prints the arguments according to that format string. The format string is just a string with special codes like `%s` (for strings) or `%d` (for integers), instructing `printf` to expect an argument of a certain type, to be printed a certain way. For example, `printf("%d %ss", 25, cat)` will print the string "25 cats". More modern languages also include formatted printing; for example, `java.io.PrintStream` also has a `printf` function which works in essentially the same way.

Augustsson proposed static typing for `printf`, by computing the type of the arguments from the format string [2]. Let us consider code for this in Agda, from `string-format.agda` in the IAL. We will define a `format` function for formatted printing (to strings, not actual output channels). The typing for this function is

```
format : (f : string) → format-t f
```

So `format` takes in a format string `f`, and then returns a value whose type is computed, by the helper function `format-t`, from `f`. The type of this helper function is thus `string → Set`. It will turn out that a direct definition of these functions by pattern-matching on lists of characters runs into trouble. We will consider this attempt first, before turning to a working solution.

```

format-th :  $\mathbb{L}$  char  $\rightarrow$  Set
format-th ('%' :: 'n' :: f) =  $\mathbb{N} \rightarrow$  format-th f
format-th ('%' :: 's' :: f) = string  $\rightarrow$  format-th f
format-th (c :: f) = format-th f
format-th [] = string

format-t : string  $\rightarrow$  Set
format-t s = format-th (string-to- $\mathbb{L}$ char s)

```

Figure 7.5: Computing the type of the rest of the `format` function, from the format string

7.2.1 An unsuccessful attempt

The code shown below can be found in `string-format-issue.agda` in the IAL. First, we will want to work lists of characters rather than strings. We will then convert strings to these, using a function `string-to- \mathbb{L} char` in `string.agda` (in the IAL). So we will write helper functions for `format-t` and `format`, which operate on lists of characters instead of strings. The first of these is shown in Figure 7.5. If the list of characters begins with `' % '` and then `' n '`, then that is the format specifier we are using to indicate a natural-number argument. So the type we compute (on the right-hand side of that first equation for `format-th`) is for a function taking in a natural-number argument and then proceeding as specified by the recursive call to `format-th` on the rest of the format string. Similarly, if the format string, as a list of characters, begins with `' % '` and then `' s '`, the type we compute says that the function should take in a string argument. If the format string begins any other way, then there is no argument to take in, and `format-th` just recurses on the rest of the format string. If the format string is empty, then the type is just `string`; this is the ultimate return type of the `format` function. As an example of how `format-t` works, if we ask Agda to evaluate `format-t "The %n %s are %s."` (using Control-c Control-n), we will get

```
 $\mathbb{N} \rightarrow$  string  $\rightarrow$  string  $\rightarrow$  string
```

This is what we would expect, because the format string `"The %n %s are %s."` contains a `%n` format specifier, and then two `%s` specifiers. So the `format` function should take a natural-number argument, then two string arguments, and then return a string, as the type displayed just above specifies.

Now let us try to write the `format` function. Code for this, with one hole which we cannot fill in, is in Figure 7.6. Let us look first at the parts of this code which type-check. First, the types of the functions in the figure show how the `format-t` and `format-th` functions are used, just as we noted in the previous section. For `format-h`, we are taking in the format string `f` as a list of characters, so the return type of the function is computed from `f` by `format-th`. The format string is the second argument to `format-h`. The first argument is an accumulator holding

the string we have generated so far, from the prefix of the format string, and any corresponding arguments, which we have already processed. Let us look closely at the first defining equation for `format-h`:

```
format-h s ('%' :: 'n' :: f) =
  λ n → format-h (s ++ (string-to-Lchar (ℕ-to-string n))) f
```

We know that the type of the right-hand side is supposed to be

```
format-th ('%' :: 'n' :: f)
```

This is definitionally equal to $\mathbb{N} \rightarrow \text{format-th } f$, by the definition of `format-th`. So indeed, the right-hand side takes in an argument `n` (implicitly of type \mathbb{N}), and then makes a recursive call with format string `f`. That recursive call has type `format-th f`, as required. The accumulator argument is updated with the result of converting the natural-number argument `n` to a string, and then a list of characters. The second defining equation for `format-h` is typed similarly. The last (fourth) defining equation is where we just convert the accumulator back to a string and return it.

So surprisingly, the problem is arising in the third equation, when we do not have a format specifier in the format string. If we look at the type of the hole (with `Control-c Control-;`), we will see that it is `format-th (c :: f)`. You might find this strange. Agda is saying that it cannot simplify this expression any further. But surely, we might think, this should simplify to just `format-th f`, because that is what the code for `format-th` says to do in case the format string does not match the pattern in one of the first two defining equations for `format-th`. Ah – we might imagine Agda replying – but how do we know that `c :: f` does not match one of those patterns? Because – we might indignantly respond – we are in a defining equation for `format-h` where those patterns have already been checked in earlier equations. And that is where we get stuck. **Agda does not use the information that a particular term has failed to match some patterns when doing partial evaluation.** So just because, if we reach this point in the code for `format-h`, `c :: f` will have failed to match the patterns `'%' :: 'n' :: f` and `'%' :: 's' :: f`; that does not mean that Agda will use that information when trying to reduce the type expression `format-th (c :: f)`. As far as Agda is concerned, the fact that the expression `c :: f` could, if we do not consider the context in which it appears, match one of those patterns, is enough to prevent partial evaluation from simplifying `format-th (c :: f)`. Agda needs to see explicitly that the input to `format-th` is not of the form `'%' :: 'n' :: f` nor `'%' :: 's' :: f` in order to consider subsequent defining equations for `format-th`. And `c :: f` cannot, by itself, be seen not to match those patterns. So partial evaluation gets stuck, and we cannot complete this code. Another way to identify the problem is that `format-th` has a default case, where the pattern alone is not sufficient to show that evaluation should go to that case.

The fact that Agda does not keep track of information on what patterns a term must have failed to match is disappointing, though one can only imagine the complexity that would be involved in adding such support, in conjunction with all of

```

format-h :  $\mathbb{L}$  char  $\rightarrow$  (f :  $\mathbb{L}$  char)  $\rightarrow$  format-th f
format-h s ('%' :: 'n' :: f) =
   $\lambda$  n  $\rightarrow$  format-h (s ++ (string-to- $\mathbb{L}$ char
                        ( $\mathbb{N}$ -to-string n))) f
format-h s ('%' :: 's' :: f) =
   $\lambda$  s'  $\rightarrow$  format-h (s ++ (string-to- $\mathbb{L}$ char s')) f
format-h s (c :: f) = ?
format-h s [] =  $\mathbb{L}$ char-to-string s

format : (f : string)  $\rightarrow$  format-t f
format f = format-h [] (string-to- $\mathbb{L}$ char f)

```

Figure 7.6: Attempt at the `format` function, with a hole we cannot fill in

Agda's other features. Fortunately, we can work around this problem, as we will see next.

7.2.2 A working solution

The failure of the previous solution is due to the use of a default pattern in the definition of the `format-th` function. We can get a working solution by rewriting `format-th` and `format-h` so there are no default patterns. To do this, we will introduce an auxiliary datatype which gives the same information as the `format` string, but with the structure we wish to use in `format-th` and `format-h`. This auxiliary datatype is reminiscent of the idea of a view, introduced for dependently typed programming by McBride and McKinna [14]; though views are indexed by the data in question (here, the format strings), while for us, an unindexed type is sufficient. The code may be found in `string-format.agda`.

Figure 7.7 gives the definition of the auxiliary datatype `format-d`, and a function `format-cover`, which converts a format string to an element of this type. The constructors of the `format-d` type represent the possibilities for format specifiers at the current position in the format string. This is made clear by the `format-cover` code to convert a format string into a `format-d` value. The current position in the format string might have a format specifier `%n` or `%s`, in which case we apply the `format-nat` or `format-string` constructor, to the result of recursively converting the reset of the format string. This is done in the first and second equations for `format-cover`. The second equation is for the case where there is no format specifier. We need to record the character `c` at that point in the format string, since this should be printed out by `format`. The last equation is for when the end of the format string is reached.

Now we can revisit the definitions of `format-t` and `format`, using the `format-d` representation of format strings. The code for `format-t` and its helper function

```

data format-d : Set where
  format-nat  : format-d → format-d
  format-string : format-d → format-d
  not-format  : (c : char) → format-d → format-d
  empty-format : format-d

format-cover :  $\mathbb{L}$  char → format-d
format-cover ('%' :: 'n' :: s) =
  format-nat (format-cover s)
format-cover ('%' :: 's' :: s) =
  format-string (format-cover s)
format-cover (c :: s) = not-format c (format-cover s)
format-cover [] = empty-format

```

Figure 7.7: Auxiliary data structure for representing format strings

```

format-th : format-d → Set
format-th (format-nat v) =  $\mathbb{N}$  → format-th v
format-th (format-string v) = string → format-th v
format-th (not-format c v) = format-th v
format-th empty-format = string

format-t : string → Set
format-t s = format-th (format-cover (string-to- $\mathbb{L}$ char s))

```

Figure 7.8: Defining `format-t` using the `format-d` representation of format strings

`format-th` is given in Figure 7.8. The code is very similar to that in Figure 7.5, except that the helper function `format-th` recurses over a `format-d` value, rather than a format string. The benefit of this change of representation is that now, `format-th` has no default cases: any input to the function matches exactly one equation. This was not true for the definition in Figure 7.5, where a string beginning with `%n` could match both the pattern for numeric format specifiers and the default case for non-empty strings.

The payoff for using `format-d` values instead of format strings comes in the definition of `format` and particularly its helper function `format-h`, which can now be completed (in contrast to the attempt in Figure 7.6). The first two defining equations for `format-h` are similar to those from our earlier failed attempt (Figure 7.6): if the format specifier is for a numeric argument or a string argument, then the type of `format-h` in those two cases, which is computed by `format-th d` from the `format-d` value `d`, says that arguments of the appropriate types should be accepted. So the equations use a λ -abstraction to take in a natural number `n` and a string `s'`, respectively, in those two cases. The third case is the one that

```

format-h :  $\mathbb{L}$  char  $\rightarrow$  (d : format-d)  $\rightarrow$  format-th d
format-h s (format-nat v) =
   $\lambda$  n  $\rightarrow$  format-h (s ++
                    (string-to- $\mathbb{L}$ char ( $\mathbb{N}$ -to-string n))) v
format-h s (format-string v) =
   $\lambda$  s'  $\rightarrow$  format-h (s ++ (string-to- $\mathbb{L}$ char s')) v
format-h s (not-format c v) = format-h (s ++ [ c ] ) v
format-h s empty-format =  $\mathbb{L}$ char-to-string s

format : (f : string)  $\rightarrow$  format-t f
format f = format-h [] (format-cover (string-to- $\mathbb{L}$ char f))

```

Figure 7.9: Formatted printing (to strings) using the `format-d` representation of format strings

posed problems in the code of Figure 7.6. But here, as Agda refines the return type `format-th d` by replacing `d` with the pattern `not-format c v`, we obtain a return type `format-th (not-format c v)` which, unlike in the earlier failed attempt case, is able to reduce, to `format-th v`. That is exactly what is needed to type the recursive call to `format-h`, where the character `c` from the format string is appended to the string being accumulated in the first argument to `format-h`.

Finally, we can see the fruits of our labors from some examples of using `format`. First, if we ask Agda (with Control-c Control-d) what the type of a test like

```
format "%n% of the %ss are in the %s %s"
```

is, we get `$\mathbb{N} \rightarrow$ string \rightarrow string \rightarrow string \rightarrow string`. There are four format specifiers in the format string, and our code for `format-t` computes this type for a function taking in four arguments of the appropriate types, and then returning a string. If we ask Agda to evaluate a call to `format` with all the required arguments, like

```
format "%n% of the %ss are in the %s %s"
25 "dog" "toasty" "doghouse"
```

we indeed get the expected result

```
"25% of the dogs are in the toasty doghouse"
```

Note the inclusion of a percent sign which is not part of a format specifier, to show that we are handling such cases correctly.

7.3 Proof by Reflection

Agda does not have much support for automatic theorem-proving, where algorithms are applied to try to prove formulas of certain relatively simple kinds. The general theorem-proving problem is provably unsolvable (like the famous Halting Problem), so restrictions on the kind of formulas to be proved are needed. But there are many useful classes of formulas that can be tested for validity fully automatically. This could save a lot of effort for the human trying to verify properties of her/his code. The research literature on automated theorem-proving has not neglected the question of how to integrate automatic and human construction of proofs. There are approaches which try to import or reconstruct proofs found by external solvers [1, 3]. One can also implement automated theorem-proving algorithms directly in the source code for the proof assistant (e.g., [10]). And some proof assistants like Coq and Isabelle have mechanisms to allow the programmer to guide or augment automated theorem-proving implementations.

In this section, we will study a different approach, called proof by reflection. This is a particularly elegant way to add automated reasoning to a proof assistant like Agda. The starting point for proof by reflection is to reflect (hence the name) part of the logical language of the proof assistant into a datatype within the language. Values of this datatype are representations for expressions from the logical language. A semantic function $\llbracket _ \rrbracket$ maps those representations to their actual meaning in the logical language of the tool. By representing a formula as a value of a datatype, we can implement automated-reasoning algorithms by pattern matching and recursion over representations (where pattern matching over formulas of the logic – i.e., elements of type `Set` – is not supported by Agda or similar tools). For example, we will see how to write a simplifier, that transforms one representation into another which is simpler in some way. Finally, to make use of the simplifier for actual proofs in the logical language, one proves a soundness result for the solver: transforming a representation using the solver preserves the meaning of that representation. So anywhere we see an expression which is equal to the semantics $\llbracket r \rrbracket$ of some representation, we have a proof that this expression is equal to $\llbracket \text{simplify } r \rrbracket$.

For this abstract description of proof by reflection to make more sense, let us look at an example, from `list-simplifier.agda` in the Iowa Agda Library, of automatic simplification of certain expressions involving the list operators for append and map, as well as the list constructors `_:` and `[]`. As noted at the start of this chapter, our example will not actually require type-level computation, but more complex examples of proof by reflection typically do, as one seeks to prove certain kinds of formulas (i.e., types) purely automatically. In such cases, the semantic function $\llbracket _ \rrbracket$ computes a type, not just a list. This is the reason for including this topic in the current chapter.

```

data  $\mathbb{L}^r$  : Set → Set lone where
   $_{}^r$  : {A : Set} →  $\mathbb{L}$  A →  $\mathbb{L}^r$  A
   $_{++}^r$  : {A : Set} →  $\mathbb{L}^r$  A →  $\mathbb{L}^r$  A →  $\mathbb{L}^r$  A
   $\text{map}^r$  : {A B : Set} → (A → B) →  $\mathbb{L}^r$  A →  $\mathbb{L}^r$  B
   $_{::}^r$  : {A : Set} → A →  $\mathbb{L}^r$  A →  $\mathbb{L}^r$  A
   $[]^r$  : {A : Set} →  $\mathbb{L}^r$  A

```

Figure 7.10: Datatype for representations of certain list expressions

7.3.1 The datatype of representations, and its semantics

The first step in proof by reflection is to define a datatype representing the expressions we wish to simplify or solve. For our list simplifier, this is shown in Figure 7.10. A datatype \mathbb{L}^r is declared, for representations of list expressions. There are constructors for representing the two operations and the list constructors; $_{++}^r$, for example, is for representing an application of the append operator $_{++}$. The first constructor, $_{}^r$, allows us to embed an actual list expression, of type \mathbb{L} A, inside a list representation of type \mathbb{L}^r A. Note that we are keeping track of the types of the elements in our list representations, so the type of a list representation is \mathbb{L}^r A, indexed by the type A of the elements, just like the type \mathbb{L} A of list expressions. The quantifications over types (values of type Set) in the types for the constructors forces us to bump the level of the \mathbb{L}^r datatype up, from Set level 0 to Set level 1 (lone, from level.agda).

The names of the constructors for the type \mathbb{L}^r have been chosen to try to make it clear which list operations or constructors they represent. But we can remove all doubt by defining a semantics, mapping list representations to actual lists. This is given in Figure 7.11. The definition of this $\mathbb{L}[_]$ function already makes clear the reason for reflecting list expressions into a datatype: we can pattern match and recurse on values in this datatype. In contrast, we cannot use $\text{t1} ++ \text{t2}$ as a pattern in Agda, as patterns can only involve constructors, and $_{++}$ is a recursively defined function, not a constructor. By creating a datatype of representations, we are introducing constructors like $_{++}^r$ to correspond to the non-constructor operations on lists we are interested in. Like many semantic functions, $\mathbb{L}[_]$ is very straightforward, and exactly what you would expect. The representations mean exactly what their syntax is chosen to suggest they mean. So $_{++}^r$ represents $_{++}$, map^r represents map , and so on. If we have an embedded list under the $_{}^r$ constructor, then the first equation in Figure 7.11 says that the meaning of this representation is just the embedded list.

The same list expression can be represented in different ways. For example, to represent $(1 :: []) ++ (2 :: [])$, we could use

```
(1 ::r []) ++r (2 ::r [])
```

This is the most complete representation possible for the given expression; there


```


$$\begin{aligned}
\mathbb{L}[\_] &: \{A : \text{Set}\} \rightarrow \mathbb{L}^r A \rightarrow \mathbb{L} A \\
\mathbb{L}[1^r] &= 1 \\
\mathbb{L}[t1 \mathbin{++}^r t2] &= \mathbb{L}[t1] \mathbin{++} \mathbb{L}[t2] \\
\mathbb{L}[\text{map}^r f\ t] &= \text{map}\ f\ \mathbb{L}[t] \\
\mathbb{L}[x \mathbin{::}^r t] &= x \mathbin{::} \mathbb{L}[t] \\
\mathbb{L}[\_]^r &= []
\end{aligned}$$


```

Figure 7.11: The semantics for list representations

is no way to use more \mathbb{L}^r constructors to represent this expression. We can use fewer, though. For example, here is the least complete representation:

```
((1 :: []) ++ (2 :: []))r
```

Both representations have type $\mathbb{L}^r \mathbb{N}$, for representations of list expressions where the elements are natural numbers. They both have the same meaning, as computed by the $\mathbb{L}[_]$ function, namely the expression $(1 :: []) \mathbin{++} (2 :: [])$, which is definitionally equal to the list value $1 :: 2 :: []$. The representations, however, are not definitionally equal to any other representation of this value. This is a consequence of the main benefit of reflecting expressions into representations: the representations are constructor terms, and hence are not definitionally equal to other representations. What we lose in automatic definitional equality we more than gain in the ability to write functions which manipulate representations by pattern matching and recursion. As we have already noted, such manipulation is not possible directly on the list expressions themselves.

7.3.2 The list simplifier

To write our list simplifier, we will write one function to perform a single step of simplification, and another to traverse the representation to perform single steps wherever possible. The function for a single top-level simplification step is given in Figure 7.12. $\mathbb{L}^r\text{-simp-step}$ matches on the form of a representation t . The first equation re-associates left-nested append operations to the right. The second and third equations implement the definitional equalities for append: uses of $\text{cons}(_, _)$ are pulled out of the first position in a use of append, and the empty list is dropped. Next, we will drop the empty list if it is used in the second position of an append. This is justified not by definitional equality, but by a lemma from `list-thms.agda`:

```

$$\mathbin{++} [] : \forall \{\ell\} \{A : \text{Set}\} \ell \rightarrow (l : \mathbb{L} A) \rightarrow l \mathbin{++} [] \equiv l$$

```

$\mathbb{L}^r\text{-simp-step}$ applies the transformation implied by this lemma in two situations: if the first argument to the append is 1^r , and if it is $\text{map}^r f\ t1$. If the first argument to append is one of the other three possibilities allowed by the $\mathbb{L}^r A$ datatype, then the situation has already been covered by the first three equations

for $\mathbb{L}^r\text{-simp-step}$. So consider the following clause (from Figure 7.12):

```
 $\mathbb{L}^r\text{-simp-step} \ ((1 \ ^r) \ ++^r \ t2) \ \text{with} \ \text{is-empty}^r \ t2$ 
```

We are not pattern matching on $t2$ to detect if it is $[]^r$. Rather, we call a small helper function (not shown), which just returns `tt` or `ff` depending on whether the input is $[]^r$ or not. This lets us avoid having default cases here in the definition of $\mathbb{L}^r\text{-simp-step}$, which would cause similar problems as we encountered in the Section 7.2.1 for formatted printing. If `is-emptyr t2` returns `tt`, then we drop $t2$ from the append; otherwise, we retain it. We do a similar split on a call to `is-emptyr` if the first argument of the append is `mapr f t1`. After this, we have the cases for when the input represents a map expression. We can perform some simplification in all the cases for the list to which `map` is being applied:

- `mapr f (t1 ++r t2)`: push the `mapr` into the append to apply to $t1$ and $t2$ separately.
- `mapr f (1 ^r)`: push the `mapr` under the r constructor, changing it to a real `map`.
- `mapr f (mapr g t)`: coalesce the two maps, using the composition $f \circ g$ of f and g .
- Finally there are equations implementing the definitional equalities for `map`.

The code for $\mathbb{L}^r\text{-simp-step}$ then concludes with a few trivial cases where no transformation takes place. As an example, `list-simplifier.agda` defines a parametrized submodule `test2`, containing a value `lhs`:

```
mapr f (((11 ^r) ++r (12 ^r)) ++r (13 ^r))
```

If we apply $\mathbb{L}^r\text{-simp-step}$ to this, as done to compute `test2.one-step` (this is the fully qualified name for the value in the submodule), we will get this representation:

```
mapr f ((11 ^r) ++r (12 ^r)) ++r mapr f (13 ^r)
```

Now we need a function that can apply $\mathbb{L}^r\text{-simp-step}$ multiple times. Following standard terminology in rewriting, let us call any expression which will be transformed to a distinct representation by applying $\mathbb{L}^r\text{-simp-step}$ a **redex**. Figure 7.13 shows the code for functions $\mathbb{L}^r\text{-simp-sdev}$ and $\mathbb{L}^r\text{-simp}$, which perform multiple simplification steps. $\mathbb{L}^r\text{-simp-sdev}$ performs what is sometimes called a **superdevelopment** in the rewriting literature: we recursively transform the immediate subterms of the representation t , and then try to apply a single step of reduction to the result. This has the effect of simplifying redexes which are created by recursive simplification. (In contrast, a **development** transforms only redexes that occur in the original term.) The result of calling $\mathbb{L}^r\text{-simp-sdev}$ on `test2.lhs`, defined to be `test2.sdev` in `list-simplifier.agda`, has value:

```
mapr f (11 ^r) ++r mapr f ((12 ^r) ++r (13 ^r))
```

```

ℒ'-simp-step : {A : Set} (t : ℒ' A) → ℒ' A
ℒ'-simp-step ((t1a ++r t1b) ++r t2) = t1a ++r (t1b ++r t2)
ℒ'-simp-step (x ::r t1) ++r t2 = x ::r (t1 ++r t2)
ℒ'-simp-step ([]r ++r t2) = t2
ℒ'-simp-step ((lr) ++r t2) with is-emptyr t2
ℒ'-simp-step ((lr) ++r t2) | tt = lr
ℒ'-simp-step ((lr) ++r t2) | ff = ((lr) ++r t2)
ℒ'-simp-step ((mapr f t1) ++r t2) with is-emptyr t2
ℒ'-simp-step ((mapr f t1) ++r t2) | tt = mapr f t1
ℒ'-simp-step ((mapr f t1) ++r t2) | ff = ((mapr f t1) ++r t2)
ℒ'-simp-step (mapr f (t1 ++r t2)) =
  (mapr f t1) ++r (mapr f t2)
ℒ'-simp-step (mapr f (lr)) = (map f l)r
ℒ'-simp-step (mapr f (mapr g t)) = mapr (f ∘ g) t
ℒ'-simp-step (mapr f (x ::r t)) = (f x) ::r (mapr f t)
ℒ'-simp-step (mapr f []r) = []r
ℒ'-simp-step (lr) = lr
ℒ'-simp-step (x ::r t) = (x ::r t)
ℒ'-simp-step []r = []r

```

Figure 7.12: Function for performing a single step of simplification

So what has happened to the original representation is that we have right-associated the left-nested append, and then pushed the `mapr` in to the two lists being appended. There is some more simplification that can happen, namely to push `mapr` into `(l2r) ++r (l3r)`, but this cannot happen in a single superdevelopment. The transformation which we are applying is terminating, but convincing Agda of that fact will take more work than I wish to invest in the simplifier at this point. So we will fall back to iterated superdevelopments: the user of the simplifier must specify a natural number, for the number of times we will apply `ℒ'-simp-sdev` to the input representation. This is how `ℒ'-simp` is defined. Three iterated superdevelopments are sufficient to reduce `test2.lhs` to the following term, which cannot be further reduced:

```
(map f l1r) ++r ((map f l2r) ++r (map f l3r))
```

7.3.3 Proving that the simplifier preserves the semantics

The final step in developing our simplifier for list expressions is to prove that the simplifier just defined preserves the semantics of list representations. That is, we want to prove that the meaning of a simplified representation is the same as the original representation. As stated in the main theorem of `list-simplifier.agda`:

```

 $\mathbb{L}^r\text{-simp-sdev} : \{A : \text{Set}\} (t : \mathbb{L}^r A) \rightarrow \mathbb{L}^r A$ 
 $\mathbb{L}^r\text{-simp-sdev } (l^r) = (l^r)$ 
 $\mathbb{L}^r\text{-simp-sdev } (t1 ++^r t2) =$ 
 $\quad \mathbb{L}^r\text{-simp-step } ((\mathbb{L}^r\text{-simp-sdev } t1) ++^r (\mathbb{L}^r\text{-simp-sdev } t2))$ 
 $\mathbb{L}^r\text{-simp-sdev } (\text{map}^r f t1) =$ 
 $\quad \mathbb{L}^r\text{-simp-step } (\text{map}^r f (\mathbb{L}^r\text{-simp-sdev } t1))$ 
 $\mathbb{L}^r\text{-simp-sdev } (x ::^r t1) =$ 
 $\quad \mathbb{L}^r\text{-simp-step } (x ::^r (\mathbb{L}^r\text{-simp-sdev } t1))$ 
 $\mathbb{L}^r\text{-simp-sdev } []^r = []^r$ 

 $\mathbb{L}^r\text{-simp} : \{A : \text{Set}\} (t : \mathbb{L}^r A) \rightarrow \mathbb{N} \rightarrow \mathbb{L}^r A$ 
 $\mathbb{L}^r\text{-simp } t \ 0 = t$ 
 $\mathbb{L}^r\text{-simp } t \ (\text{suc } n) = \mathbb{L}^r\text{-simp-sdev } (\mathbb{L}^r\text{-simp } t \ n)$ 

```

Figure 7.13: Superdevelopments and iterated simplification

$$\left[\begin{array}{l} \mathbb{L}^r\text{-simp-sound} : \{A : \text{Set}\} (t : \mathbb{L}^r A) (n : \mathbb{N}) \rightarrow \\ \quad \mathbb{L} \llbracket t \rrbracket \equiv \mathbb{L} \llbracket \mathbb{L}^r\text{-simp } t \ n \rrbracket \end{array} \right.$$

This theorem is the one we can use to simplify list expressions in our regular Agda proofs. For example, `test2.test` proves the following theorem in one line, just by calling `$\mathbb{L}^r\text{-simp-sound}$ lhs 3`:

```
map f ((l1 ++ l2) ++ l3)  $\equiv$  map f l1 ++ map f l2 ++ map f l3
```

To understand how this works, we have to consider the type of `$\mathbb{L}^r\text{-simp-sound}$` , instantiated with the particular representation `test2.lhs` and number of iterations 3. That type is then:

```
 $\mathbb{L} \llbracket \text{lhs} \rrbracket \equiv \mathbb{L} \llbracket \mathbb{L}^r\text{-simp lhs 3} \rrbracket$ 
```

So we are showing that the meaning of the representation is equal to the meaning of the simplified representation. Now, the meaning of the representation (this is `test2.lhs`)

```
mapr f ((l1r ++r (l2r ++r (l3r)))
```

is just `map f ((l1 ++ l2) ++ l3)`, our list expression of interest. So we are proving that this is equal to the meaning of its simplified form. We already saw that the simplified representation returned by `$\mathbb{L}^r\text{-simp}$` with number of iterations 3 is:

```
(map f l1r ++r (map f l2r ++r (map f l3r)))
```

And the meaning of this is the desired resulting list expression

```
map f l1 ++ map f l2 ++ map f l3
```

The heart of proving `$\mathbb{L}^r\text{-simp-sound}$` is proving the following helper lemma, which states that `$\mathbb{L}^r\text{-simp-step}$` (one-step simplification) preserves the seman-

tics for list representations:

$$\mathbb{L}'\text{-simp-step-sound} : \{A : \text{Set}\} (t : \mathbb{L}' A) \rightarrow \\ \mathbb{L} \llbracket t \rrbracket \equiv \mathbb{L} \llbracket \mathbb{L}'\text{-simp-step } t \rrbracket$$

The proof is straightforward, if somewhat long, so the details are omitted here. A typical case is the one showing that re-associating a left-nested append preserves the semantics:

$$\mathbb{L}'\text{-simp-step-sound} ((t1a ++^r t1b) ++^r t2) = \\ ++\text{-assoc } \mathbb{L} \llbracket t1a \rrbracket \mathbb{L} \llbracket t1b \rrbracket \mathbb{L} \llbracket t2 \rrbracket$$

This uses the theorem `++-assoc` from `list-thms.agda`, which says that append is associative. The other simplification steps performed by `\mathbb{L}' -simp-step` are justified by similar uses of theorems from `list-thms.agda`.

7.4 Conclusion

In this chapter, we have seen some of the amazing examples that are possible in Agda using type-level computation. Types are generally giving us compile-time approximations to the run-time behavior of programs. For example, if you know that an expression has type `int` in Java, say, you know that if evaluation of that expression terminates normally (it does not run forever, and it does not raise exceptions), then what you get back will really be an `int` value like 12 or -121. By having more expressive types, Agda enables more refined type approximations of the programs' runtime behavior. The ability to compute a type from a value is a rather astonishing example of this, enabling typing of functions like formatted printing that one would likely have thought could not be meaningfully statically typed (and certainly are not in languages like C or Java). Computing types from values is arguably the essential feature which distinguishes dependently typed programming from other forms of richly typed programming. Indexed types like \mathbb{V} for vectors (Section 5.1) enable intriguingly expressive types for programs, but programs like vector append can just as well be written without type indices. Indeed, for the example of vectors, the programs just become operations on lists, which can be typed without the fancy machinery of indexed types. In contrast, programs like `format` (Section 7.2.2) cannot be written in a typed language without type-level computation. There is no simply typed analog of `format` one can type in OCaml or Haskell (though see posts by Oleg Kiselyov for some remarkable approximations [12]).

7.5 Exercises

1. Define some further operations on the type \mathbb{Z} of Section 7.1, such as negation, subtraction, and multiplication.
2. Prove commutativity of `_+Z_`. The proof will invoke commutativity of natural-number addition for the case where that function is actually called.
3. Try out some examples of formatted printing using `format` from Section 7.2.
4. Try proving a sample list theorem using the list simplifier, as shown in Section 7.3.3.

Chapter 8

A Case Study: Huffman Encoding and Decoding

In this chapter, we will consider a larger example of dependently typed programming in Agda, namely the implementation of a standalone tool for Huffman encoding and decoding. The basic idea of the coding scheme invented by David A. Huffman when he was still a doctoral student at MIT, is that words in the text which occur more frequently will get assigned a shorter bitstring (sequence of 1's and 0's) as a code. Also, no code is a prefix of any other, so there is no ambiguity when decoding. We will apply Huffman coding at the level of English words, to encode simple unpunctuated text. Consider this example input:

```
hi hi bye hello hi hi bye hello hi hi bye hello
```

Here, the word `hi` occurs twice as often as `bye` and `hello`, so it should get a shorter code. We will encode this text as follows:

```
! bye <- 11
hello <- 10
hi <- 0
001110001110001110
```

There is a special `!` at the start of the file to signal to our tool that this is encoded input. Then comes a representation of the Huffman code (from the `!` to just before the last line). The more frequent word `hi` will be mapped to `0` which is half the length of the codes `11` and `10` assigned for the less frequent words `bye` and `hello`. Using this code, our program will be able to decode the bitstring in the last line (`001110001110001110`) to get back the original text.

Huffman coding can be implemented elegantly using pure functional data structures and functions. We will make use of Braun trees (Section 5.2), to implement a priority queue (a data structure storing ordered elements which supports efficient removal of the smallest element). The central idea of the algorithm is to construct a Huffman tree by first initializing the queue with leaf trees storing words to be encoded with their frequencies (number of occurrences) in the input; and second, repeatedly combining the two smallest elements from the priority queue, where smaller elements are those that occur less frequently in the input. The end result is a tree where words that occur less frequently are deeper in the tree, and words that occur more frequently are nearer the root. The paths in the tree then determine the codes. Higher-frequency words are nearer the root, and hence have shorter paths. They get shorter codes, as desired.

While there are many properties one could wish to verify about the implementation, we will just use a little internal verification to rule out some simple errors. Even this ends up requiring some theorem-proving. It would be nice to extend the example to prove things like the fact that the code that is computed is prefix-free. But further verification must remain to future work – or perhaps projects of interested readers!

The code for this example can be found in the `huffman` subdirectory of the IAL. It requires `gratr` (Chapter 6) to compile (or generate) the parser for the tool. The above example input and its encoded version are `test1.txt` and `test1.huff` in this directory.

8.1 The Files

The `huffman` directory of the IAL contains the following source files:

file	description
<code>huffman.gr</code>	the <code>gratr</code> grammar for the input files
<code>main.agda</code>	entry point for the program, also contains encoding and decoding functions
<code>huffman-tree.agda</code>	data structure representing Huffman trees
<code>huffman.agda</code>	parser generated by <code>gratr</code>
<code>huffman-types.agda</code>	datatypes for the input files (also generated by <code>gratr</code>)
<code>Makefile</code>	for compiling everything using Linux <code>make</code>
<code>huffman-main.agda</code>	the generated main file, which can be compiled just to test parsing of the input files

In addition, there are these test files, where `X` is 1, 2, or 3:

file	description
<code>testX.txt</code>	a test input file to be encoded
<code>testX-frequencies.txt</code>	output file showing the frequencies of words in the input file to be encoded
<code>testX-mapping.txt</code>	output file showing the Huffman code computed for an input file
<code>testX.gv</code>	an output file in Graphviz format showing the computed Huffman tree
<code>testX.huff</code>	the encoded output file
<code>testX-decoded.txt</code>	the result of decoding the encoded output

8.2 The Input Formats

Figure 8.1 shows the `gratr` grammar used to generate several of the source files listed above, for the parser for the codec (encoder/decoder). Input files are either simple textual inputs parsed as words; or else encoded files that begin with a `!`, then list `codes` mapping words to `bvlits`, and then containing a `bvlit` representing the input text encoded with the given code. A `bvlit` is a *bit-vector literal*, which is just a sequence of 0 and 1 digits. We have already seen an example of input to be encoded and the encoded result, at the start of the chapter.

8.3 Encoding Textual Input

In `main.agda`, we have code to process the two forms of `cmd` which the grammar above allows: `Encode` and `Decode` commands. Most of the code in `main.agda` is devoted to handling `Encode` commands, as for these we must compute the Huffman coding and then apply it, while for `Decode` commands the Huffman coding is given as part of the encoded input. Encoding is broken into the following steps, covered in the rest of this section:

1. Compute frequencies of words in the input
2. Build a priority queue (using a Braun tree) with leaves of Huffman trees storing words and their frequencies. This priority queue is referred to as a `pqueue` in the code in `main.agda`.
3. Process the `pqueue` as described at the start of the chapter: the two Huffman trees in the `pqueue` with the smallest frequencies are removed and combined to create a new Huffman tree, whose frequency is the sum of the frequencies of its two subtrees. This new tree is inserted back into the `pqueue`.
4. When only one Huffman tree remains in the `pqueue`, this is the final tree that determines the codes. Each path from the root of that tree to a leaf determines the code for the word stored at the leaf. Each Huffman tree is either a leaf or has two subtrees. A path into a left subtree is associated with digit 0, and into a right subtree with digit 1. Since distinct leaves are reached by distinct paths where moreover, neither path can be a prefix of the other, the code is prefix-free (no code is a prefix of any other code). This enables deterministic decoding of an encoded text. Note that depending on the order the `pqueue` is processed, different Huffman trees, and hence different codes, can be computed for the same input text.
5. The resulting code is output to a `.huff` file, together with the result of encoding the input text using that code.

We will not walk through the code in `main.agda` which puts these steps all together to generate the encoded output. Instead, we will just focus on the imple-

```

huffman

Start

start

Syntactic

File : start -> ows cmd ows .

Encode : cmd -> words .

Decode : cmd -> '!' ws codes ws bvlit .

CodesStart : codes -> code .
CodesNext : codes -> code ws codes .
Code : code -> word ws '<-' ws bvlit.

WordsStart : words -> word.
WordsNext : words -> word ws words.

BvlitStart : bvlit -> digit .
BvlitCons : bvlit -> digit bvlit .

Zero : digit -> '0'.
One : digit -> '1'.

Lexical

word -> character .
word -> character word .

character -> ['a' - 'z'] | ['A' - 'Z'] | '(' | ')' | ',' | '.' .

aws => ('\n' | '\t' | space ) .
ws => aws+.
ows => aws* .

```

Figure 8.1: Grammar for files to encode and decode

mentations of these steps.

8.3.1 Computing word frequencies using tries

Let us see first how we compute word frequencies from the input text. So for `test1.txt` (the example input to be encoded, listed at the start of the chapter), the frequencies are:

```
bye -> 3
hello -> 3
hi -> 6
```

Indeed, running the main executable on `test1.txt` will generate output file `test1-frequencies.txt` with that data. Let us see now how the code in `main.agda` is computing this.

To map strings efficiently to values, we are using a trie, a data structure devised for that purpose. A trie is a tree where the root corresponds to the starting position of the strings in the mapping. Each node maps characters for the next position in the string to sub-tries. This scheme allows common prefixes of strings in the mapping to be shared, saving space and making the operations of finding or inserting strings efficient. Tries are defined in `trie.agda` in the IAL. For an example: suppose we have the following mapping:

```
ant -> 1
antique -> 2
anger -> 3
hanger -> 4
```

The (unique) trie representing this mapping is shown in Figure 8.2. Only the prefixes “an” and “ant” are shared across paths to nodes containing data values 1, 2, and 3. The suffix “anger” shared by “hanger” and “anger” is not shared in a trie.

The code in Figure 8.3 defines a function `compute-frequencies` to map words (the parse tree for a sequence of words) to a trie mapping each distinct word to its frequency in the input words. There is also a helper function `inc-frequency`. Since we are using pure functional data structures, updating the trie is a non-destructive operation that creates a new trie, which nevertheless shares as much of the old trie as possible. `compute-frequencies` takes in the old trie – initially an `empty-trie` – and updates for each word in the input words. The grammar for words requires that there is at least one word in the input, so the two cases for `compute-frequencies` are for whether we are considering that last single word (`WordsStart`), or else a word followed by more words (`WordsNext`). In each case, we use the helper function `inc-frequency` to take in a word and a trie and produce a new trie which is just like the old one except that the frequency for the word has been incremented if the word was already in the trie; or else the frequency is set to 1 if the word was not yet in the trie.

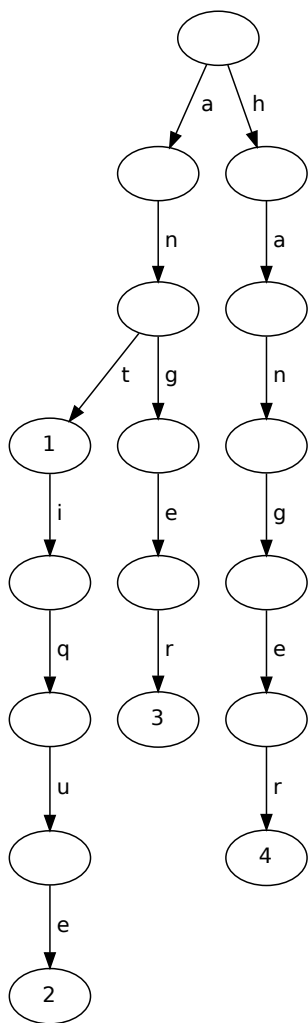


Figure 8.2: An example trie

```

inc-frequency : word → trie ℕ → trie ℕ
inc-frequency w t with trie-lookup t w
inc-frequency w t | nothing = trie-insert t w 1
inc-frequency w t | just c = trie-insert t w (suc c)

compute-frequencies : words → trie ℕ → trie ℕ
compute-frequencies (WordsStart w) t = inc-frequency w t
compute-frequencies (WordsNext w ww) t =
  compute-frequencies ww (inc-frequency w t)

```

Figure 8.3: Computing the frequencies of words

```

inc-frequency-nonempty :
  ∀(w : word)(t : trie ℕ) →
    trie-nonempty (inc-frequency w t) ≡ tt

compute-frequencies-nonempty :
  ∀(ws : words)(t : trie ℕ) →
    trie-nonempty (compute-frequencies ws t) ≡ tt

```

Figure 8.4: Lemmas about the computed trie of frequencies

Figure 8.4 gives two lemmas about the functions for computing the trie of frequencies. Since the grammar requires that there is at least one input word, we can prove that the frequency trie must be nonempty: there is at least one string mapped by the trie. These lemmas rest on the following theorem, proved in `trie-thms.agda` in the IAL:

```

trie-insert-nonempty :
  ∀{A : Set}(t : trie A)(s : string)(a : A) →
    trie-nonempty (trie-insert t s a) ≡ tt

```

This proves the intuitively obvious fact that if one inserts a mapping associating value a with string s into a trie t , then the resulting trie is nonempty. We will use this property to show that we will always be able to compute a (non-empty) Huffman tree from the frequency trie.

8.3.2 Initializing a priority queue with Huffman leaves

Once we have computed the frequency trie as described in the previous section, the next step is to initialize a `pqueue` with Huffman trees that are leaves. Let us first look at the Huffman tree datatype, found in `huffman-tree.agda` in the `huffman` subdirectory of the IAL, and shown in Figure 8.5. The `ht-leaf` constructor builds a `huffman-tree` from a string, which will be a word from the input text; and a natural number, which will be that word's frequency in the input text. The `ht-node` constructor builds a new `huffman-tree` from left and right subtrees, and a combined frequency, which should be the sum of the frequencies

of the subtrees. The helper function `ht-frequency` returns the frequency (the natural number) from either an `ht-leaf` or an `ht-node`. There is an opportunity for some further verification here, since we could seek to enforce that a combined frequency stored in an `ht-node` really is the sum of the frequencies of the left and right subtrees. We could do this by indexing the `huffman-tree` type by its frequency. In this case, I opted not to do this. While it would be reasonably simple to state and enforce this invariant, there is not too much risk of an error here. There is only one place in the code where this combined frequency is calculated and given to the `ht-node` constructor, and so it seems reasonable to rely on visual inspection for that one location in the code, to ensure that the combined frequency is being correctly calculated. Certainly, if one had a bigger “budget” (of time, say) for verification for this code, it would be reasonable to use Agda’s type system to ensure correctness of these combined frequencies.

Figure 8.5 also defines a function `ht-compare`, which we use to instantiate the Braun tree module so that our priority queue can order Huffman trees by frequency. This instantiation is done in `main.agda`, near the top of the file. The code is as follows:

```
import braun-tree as bt
open bt huffman-tree ht-compare
```

The instantiation occurs where we supply `huffman-tree` and `ht-compare` as the type `A` and ordering `<A_`, respectively, that the `braun-tree` module (in `braun-tree.agda` in the IAL) is expecting as module parameters. Here in the code from `main.agda`, there is a little fiddling going on in the module instantiation. It should work just as well to have

```
open import braun-tree huffman-tree ht-compare
```

For reasons I do not understand, however, if we do this, then Agda qualifies the names of symbols from that module in a peculiar way. For example, if we ask Agda to normalize `bt-empty`, which is the empty Braun tree, then we will see something like this:

```
.#braun-tree-36725352.bt-empty
```

On the other hand, with the two-line instantiation listed above, where `braun-tree` is first imported as `bt` (in other words, `bt` is being given as the name we wish to use in `main.agda` to refer to the module), and then its instantiation is opened, we get the following more readable normal form for `bt-empty` (and similarly for other symbols from `braun-tree.agda`):

```
bt.bt-empty
```

Moving on from this minor point: `main.agda` then defines the type `pqueue` and some operations on it, using the Braun tree type and corresponding operations. The code is in Figure 8.6. The type of data in the priority queue has been specialized, by our module instantiation, to `huffman-tree`. We have functions for inserting an element into the `pqueue`, and removing the minimum element.

```

data huffman-tree : Set where
  ht-leaf : string → ℕ → huffman-tree
  ht-node : ℕ → huffman-tree → huffman-tree →
    huffman-tree

ht-frequency : huffman-tree → ℕ
ht-frequency (ht-leaf _ f) = f
ht-frequency (ht-node f _ _) = f

ht-compare : huffman-tree → huffman-tree → ℬ
ht-compare h1 h2 = (ht-frequency h1) < (ht-frequency h2)

```

Figure 8.5: The Huffman tree datatype, and some helper functions

```

pqueue : ℕ → Set
pqueue = braun-tree

pq-empty : pqueue 0
pq-empty = bt-empty

pq-insert : ∀ {n : ℕ} → huffman-tree → pqueue n → pqueue (suc n)
pq-insert = bt-insert

pq-remove-min : ∀ {p : ℕ} → pqueue (suc p) →
  huffman-tree × pqueue p
pq-remove-min = bt-remove-min

```

Figure 8.6: Priority queue implemented with Braun trees

The types for these operations come from the types for the corresponding operations on the `braun-tree` type. In particular, inserting increases the size of the `braun-tree`, and removing decreases the size.

Finally, we can build our initial `pqueue` storing leaf Huffman trees. The code, from `main.agda` in the `huffman` subdirectory of the IAL, is shown in Figure 8.7. This code just builds `ht-leaf` trees for each pair of a word and a frequency, contained in the input list. These leaf trees are then inserted, using `pq-insert`, into the `pqueue`. The input list is derived from the frequency trie that we computed in the previous section, using a function from `trie.agda` in the IAL with the following typing:

```

trie-mappings : ∀ {A : Set} → trie A → ℒ (string × A)

```

This function just returns a list of all the mappings represented by the given trie.

```

build-huffman-pqueue : (l :  $\mathbb{L}$  (string  $\times$   $\mathbb{N}$ ))  $\rightarrow$  pqqueue (length l)
build-huffman-pqueue [] = pq-empty
build-huffman-pqueue ((s , f) :: l) =
  pq-insert (ht-leaf s f) (build-huffman-pqueue l)

```

Figure 8.7: Code to build the initial `pqueue`, which will then be processed to compute the Huffman tree

```

process-huffman-pqueue :  $\forall \{n\} \rightarrow n \neq 0 \equiv \text{ff} \rightarrow$ 
                        pqqueue n  $\rightarrow$  huffman-tree
process-huffman-pqueue{0} () b
process-huffman-pqueue{suc n} _ t with pq-remove-min t
process-huffman-pqueue{suc 0} _ t | h , _ = h
process-huffman-pqueue{suc (suc n)} _ _ | h , t with pq-remove-min t
process-huffman-pqueue{suc (suc n)} _ _ | h , _ | h' , t =
  process-huffman-pqueue{suc n} refl
  (pq-insert (ht-node ((ht-frequency h) + (ht-frequency h')) h h')
    t)

```

Figure 8.8: Code to compute a single Huffman tree by processing the initial `pqueue`

8.3.3 Processing the priority queue to compute a Huffman tree

Now that we have an initial `pqueue`, we can process it to compute a single Huffman tree by combining all the `ht-leaf` trees in the `pqueue`. By combining Huffman trees in order from lowest to highest frequency, we create a Huffman tree where less frequently used words are found at leaves that are deeper in the tree. Such leaves have longer paths from the root, and will get longer codes, which is the central idea of Huffman coding.

The code to process the `pqueue` is given in Figure 8.8. One interesting point here is that the function requires a proof that the size `n` of the `pqueue` is not zero. If the size were zero, then we could not return a `huffman-tree`, because the definition of that type in Figure 8.5 does not allow for empty trees. We already saw in Figure 8.4 that we have a proof that the computed frequency trie is nonempty. We can then use another theorem from `trie-thms.agda` in the IAL to prove that the list of mappings we obtain with `trie-mappings` from that frequency trie must also be nonempty:

```

trie-mappings-nonempty :
   $\forall \{A : \text{Set}\} (t : \text{trie } A) \rightarrow$ 
    trie-nonempty t  $\equiv$  tt  $\rightarrow$  is-empty (trie-mappings t)  $\equiv$  ff

```

So we can satisfy the requirement of `process-huffman-pqueue`.

Returning to Figure 8.8: the code splits on the size of the `pqueue`. The size cannot be zero, as just discussed, and we can use the absurd pattern in that case. If the size is nonzero, we first remove the minimum element in a `with` expression. We

can then refine the pattern to consider two cases: if the size is one or at least two. If the size is one (`suc 0`), then the algorithm completes and we just return the sole Huffman tree contained in the `pqueue`. This happens in this equation from the definition in Figure 8.8:

```
process-huffman-pqueue{suc 0} _ t | h , _ = h
```

If the size of the `pqueue` is at least two, then we can perform the basic step of combining the two lowest-frequency Huffman trees into a new tree with the combined frequency. So we call `pq-remove-min` again to get the second smallest element (of the original `pqueue`). In the code in Figure 8.8, at this point we have the lowest-frequency Huffman tree as `h`, the second lowest as `h'`, and the remaining `pqueue` (after removing those two trees) as `t`. The last equation in the figure shows how we combine `h` and `h'` with the `ht-node` constructor, with the sum of their frequencies as the new combined frequency, and then insert this new Huffman tree back into the `pqueue`.

8.3.4 Computing the code from the Huffman tree

Given the Huffman tree computed in the previous section, we can build the actual Huffman code. We will represent this code as a trie mappings words occurring in the input text to strings of ones and zeros representing the associated codes. Figure 8.9 shows the code for the `build-mapping` function which computes this trie. The main work is done in the helper function `build-mappingh`, which takes in a Huffman subtree, a current coding (the second explicit argument to `build-mappingh`, of type `trie string`), and a list of characters representing the (reversed) path to the Huffman subtree. It returns the updated coding. If we encounter an `ht-leaf`, then we need to map the string `s` stored in that leaf to the proper code, which is obtained from the list of characters `l` representing the path. As noted, this list must first be reversed, since we build it up from the root down, which makes the earlier characters in the path appear later, while we need them to appear earlier. If the Huffman tree is an `ht-node`, then we recursively process the left and right subtrees, where the path has been extended with a '0' for the left subtree and a '1' for the right. By nesting the recursive calls to `build-mappingh`, we combine the effects of these calls in building the updated coding.

8.3.5 Encoding the input words using the computed code

Finally, we can use the mapping we computed to encode a sequence of words, just by concatenating (with the string concatenation operator `_^_` from `string.agda` in the IAL) the encodings of all the individual words. The code for this is in Figure 8.10. We have not attempted to prove that the coding (the argument of type `trie string` to `encode-words`) has mappings for all the input words. Doing

```

build-mappingh : huffman-tree → trie string →
  ℒ char → trie string
build-mappingh (ht-leaf s _) m l =
  trie-insert m s (ℒchar-to-string (reverse l))
build-mappingh (ht-node _ h1 h2) m l =
  build-mappingh h2 (build-mappingh h1 m ('0' :: l)) ('1' :: l)

build-mapping : huffman-tree → trie string
build-mapping h = build-mappingh h empty-trie []

```

Figure 8.9: The code to compute a mapping from input words to associated codes (represented as strings of ones and zeros), from the computed Huffman tree

```

encode-word : trie string → word → string
encode-word t w with trie-lookup t w
encode-word t w | nothing = "error"
encode-word t w | just s = s

encode-words : trie string → words → string
encode-words t (WordsNext w ww) =
  encode-word t w ^ encode-words t ww
encode-words t (WordsStart w) = encode-word t w

```

Figure 8.10: Function to use a coding, represented as a trie, to transform the input words into an encoded bitstring

so would likely be rather involved, as we would need more reasoning about the effect of inserting mappings into tries, and we would need to state and maintain an invariant relating the strings mapped by the trie with the words in the input text. Since we are not doing such verification here, the code for `encode-word` has to deal with the *a priori* possibility that the word to be encoded is not mapped by the coding – even though we have designed our code so that this should never happen. Still, we have not proved this to Agda, so `encode-word` has to consider the case where looking up the word in the coding returns `nothing`. In that case, `encode-word` just returns `"error"`. Again, this should not happen, but we have not done the work necessary to rule it out statically.

This completes the implementation of the `process-cmd` function when it is called with an `Encode` command, to encode textual input parsed by the `gratr`-generated parser. In `main.agda`, `process-cmd` returns strings representing the contents of the intermediate files described in Section 8.1 (like `testX-mapping.txt`). Other code in `main.agda` then uses functions from `io.agda` in the IAL to print out those strings to the various files. This `io.agda` file just imports a number of Haskell functions for doing input/output. We will not consider further how these work, but the interested reader can consult one of the many references or tutorials online about IO in Haskell, as this is the same method used (on a very limited basis) in the IAL.

8.4 Decoding Encoded Text

As noted at the start of the chapter, decoding encoded text is simpler than the process of encoding it, because we do not have to compute the code: the encoded file contains it. Recalling the example given at the start of the chapter, we can see the code first, followed by the encoded string:

```
! bye <- 11
hello <- 10
hi <- 0
001110001110001110
```

The approach used here is first to build a tree data structure here dubbed a *code tree*, and then use the code tree to decode the encoded input. We build up the code tree by inserting codes one at a time into an initially empty code tree.

8.4.1 Code trees

The definition of the `code-tree` datatype is in Figure 8.11, along with a few helper functions. A code tree can be empty (`ct-empty`), a leaf storing a word (`ct-leaf`), or else a `ct-node` with left and right subtrees. The path through the code tree to a leaf storing string `s` is the code to which `s` has been mapped in the encoding. So we can reverse the encoding – and hence decode the input string – by following a path through the code tree to a word, emitting that word, and then repeating the process by following the path corresponding to the next code in the encoded input. The helper function `flip-digit` just swaps `Zero` and `One`, the sole constructors for the type `digit`, derived from the grammar in Figure 8.1. The function `sub-ct` takes a `digit` and a `code-tree`, and tries to return either the left or right subtree, depending on whether the `digit` is `Zero` or `One` respectively. For Leaves and empty code trees, `sub-ct` returns the empty code tree. Finally, `ct-node-digit` behaves just like `ct-node`, except that it swaps the subtrees if the `digit` given as the first argument is `One`.

8.4.2 Computing the code tree

Figure 8.12 shows code to make a code tree from an input of type `codes`, representing the code as parsed in by the `gratr`-generated parser. The `make-code-tree` function just repeatedly calls `ct-insert` to insert a code into the code tree. The interesting work is done by `ct-insert`. The idea is that we are building up a modified version of the input code tree, where we have added a path corresponding to the bitstring in the given input `code`. If the bitstring is just the single ending digit (in the first equation, with constructor `BvLitStart`), then we have a new `ct-leaf` to create. We make this leaf the left or right subtree of the updated code

```

data code-tree : Set where
  ct-empty : code-tree
  ct-leaf : string → code-tree
  ct-node : code-tree → code-tree → code-tree

flip-digit : digit → digit
flip-digit Zero = One
flip-digit One = Zero

sub-ct : digit → code-tree → code-tree
sub-ct _ ct-empty = ct-empty
sub-ct _ (ct-leaf _) = ct-empty
sub-ct Zero (ct-node t1 t2) = t1
sub-ct One (ct-node t1 t2) = t2

ct-node-digit : digit → code-tree → code-tree → code-tree
ct-node-digit Zero t1 t2 = ct-node t1 t2
ct-node-digit One t1 t2 = ct-node t2 t1

```

Figure 8.11: The definition of code trees, and some helper functions for them

tree, depending on whether the digit is Zero or One (this is handled for us by `ct-node-digit`). For the other subtree of the updated code tree, we use `sub-ct` to get the other subtree of the input code tree. So if we are adding our `ct-leaf` as a new left subtree, we will use `sub-ct` to get the old right subtree, and we will use that as the new right subtree. Similar reasoning is used in the case where the code has a bitstring of length greater than one (in the second equation, with constructor `BvLitCons`), except that instead of creating a `ct-leaf`, we make a recursive call to `ct-insert` to create the new subtree.

```

ct-insert : code-tree → code → code-tree
ct-insert t (Code s (BvLitStart d)) =
  ct-node-digit d (ct-leaf s)
  (sub-ct (flip-digit d) t)
ct-insert t (Code s (BvLitCons d v)) =
  ct-node-digit d (ct-insert (sub-ct d t) (Code s v))
  (sub-ct (flip-digit d) t)

make-code-tree : code-tree → codes → code-tree
make-code-tree t (CodesNext c cs) =
  make-code-tree (ct-insert t c) cs
make-code-tree t (CodesStart c) = ct-insert t c

```

Figure 8.12: Computing a code tree from input codes

```

decode-stringh : code-tree → code-tree → bvlit → string
decode-stringh orig n (BvlitCons d v) with sub-ct d n
decode-stringh orig n (BvlitCons d v) | ct-leaf s =
  s ^ " " ^ (decode-stringh orig orig v)
decode-stringh orig n (BvlitCons d v) | ct-empty = "error\n"
decode-stringh orig n (BvlitCons d v) | n' =
  decode-stringh orig n' v
decode-stringh orig n (BvlitStart d) with sub-ct d n
decode-stringh orig n (BvlitStart d) | ct-leaf s = s ^ "\n"
decode-stringh orig n (BvlitStart d) | _ = "error\n"

decode-string : code-tree → bvlit → string
decode-string t v = decode-stringh t t v

```

Figure 8.13: Decoding an encoded bitstring (the `bvlit`) using a code tree

8.4.3 Decoding the input

To decode an encoded input string, we must traverse the code tree following the sequence of 0s and 1s we see at the current point in the bitstring which constitutes the encoded text. If we see a 0 in the bitstring we should go left in the code tree, and if we see a 1 we should go right. When we reach a `ct-leaf`, we know we have reached the end of the code for a single word, so we can emit that word. We must then restart the process back at the root of the code tree, to follow the next sequence of 0s and 1s in the bitstring.

Code implementing this algorithm is in Figure 8.13. The function `decode-stringh` takes in the original complete code tree that we computed in the previous section, as its first argument. For its second argument, it takes in the subtree of this code tree, which we have reached by processing some bits from the bitstring for the encoded input. The third argument is the remaining bits of that bitstring (`bvlit`, from the grammar of Figure 8.1). There are two main cases considered by the code: if the `bvlit` has length greater than one (the `BvlitCons` case), and if the `bvlit` has length equal to one. In the former case, we use the `sub-ct` function defined in the previous section to attempt to choose a subtree of the current subtree `n` of the code tree. The subtree is determined by the current bit `d` in the `bvlit`. If the subtree is a `ct-leaf` with string `s`, then we have reached the end of a code in the bitstring, and we can output `s`. We then proceed by resetting the current subtree back to `orig` as we make a recursive call to `decode-stringh` to continue the decoding. If the subtree returned by `sub-ct` is `ct-empty`, then there is an error: the input bitstring is using a code not contained in the code tree (or else conceivably there is a bug in our code). Otherwise, if the subtree is some `ct-node n'`, we recurse with the rest of the `bvlit` and `n'` as our new subtree. The cases for when the `bvlit` is just length one (`BvlitStart`) are similar.

8.5 Conclusion

In this chapter, we have seen an example of implementing a complete executable program in Agda, for Huffman encoding and decoding. We used a `gratr`-generated parser (see Chapter 6) to parse textual input, and a simple file format for encoded data. We saw pure functional algorithms for computing Huffman trees for encoding, and code trees for decoding. The executable produces intermediate outputs, including Graphviz files to depict the Huffman trees (Graphviz is a graph-drawing program, which also has several web interfaces you can use for rendering its `.gv` files). We used a number of functional data structures like lists and Braun trees, for which we already proved numerous theorems. We also did some verification specific to this example, to show Agda that the initial priority queue we use in computing the Huffman tree is not empty. This guarantees that we can indeed compute a Huffman tree from this priority queue. The judicious use of verification is a delicate art: we wish to improve code quality as much as possible, while expending as little effort on the often difficult work of theorem-proving as we can. There are many avenues one could explore for further work on this example, described in the exercises.

8.6 Exercises

1. Build the `main` executable in the `huffman` subdirectory of the IAL, and run it on the test files included in that directory.
2. A basic theorem about tries that is currently unproved in the IAL relates how inserting into a trie with `trie-insert` effects the list of mappings returned by `trie-mappings`. State and then prove this theorem. This will likely be somewhat involved.
3. Modify the `huffman-tree` datatype so that it is indexed by its associated frequency, as discussed in Section 8.3.2, to ensure statically that frequencies are calculated correctly. Update the code in `main.agda` to work with the modified definition.
4. Prove some basic lemmas relating the helper functions in Figure 8.11. For example, you could prove

$$\forall (d : \text{digit}) (t1\ t2 : \text{code-tree}) \rightarrow \\ \text{sub-ct } d\ (\text{ct-node } d\ t1\ t2) \equiv t1$$

and a similar lemma using `flip-digit`.

Chapter 9

Formalizing Deductive Systems

While the focus of this book has been on using Agda for verified functional programming, in this chapter, we will consider some examples of using Agda for formalizing systems based on logical rules, as opposed to functional programs. Such **deductive systems** play an important role in the theoretical study of functional programming languages, and in other parts of theoretical Computer Science. We will start (Section 9.1) by considering how one may explicitly prove that binary relations are terminating, in Agda – meaning that one cannot decrease in the relation for ever, but must always hit some stopping point. This is an important tool for reasoning about deductive systems, as we will see through several examples. We will go on (Section 9.2) to see how to represent deductive systems in Agda with inductive types, starting with an operational semantics for a very minimal Turing-complete programming language, the language of SK combinators. This language is of interest for functional programming in its own right, as will be explained. We will prove termination of a class of programs in this language, using the tools of Section 9.1. The work we do in this chapter will prepare us for one further interesting example of formalizing deductive systems (Chapter 10), namely intuitionistic logic, which again has strong connections to functional programming.

9.1 Termination Proofs

Termination is a property of binary relations . If we think of the relation as an ordering, we are saying that one cannot decrease in the ordering forever. Eventually a point will be reached from which no further descent is possible. This is closely related to the mathematical idea of well-founded relations. For a basic mathematical example: the $>$ ordering on natural numbers is terminating. Every decreasing sequence of natural numbers must stop eventually, at least at 0. Of course many orderings from mathematics are not terminating. Just think about the usual $>$ ordering on integers. We can decrease forever with a chain like $3 > 2 > 1 > 0 > -1 > -2 > \dots$. Or think about the usual $>$ ordering on the positive rationals. We can decrease forever with a positive sequence like

$$1 > \frac{1}{2} > \frac{1}{4} > \frac{1}{8} > \frac{1}{16} > \frac{1}{32} > \dots$$

We can also speak of a relation's being terminating from a given starting point: from that starting point, one cannot descend forever in the relation, even if one

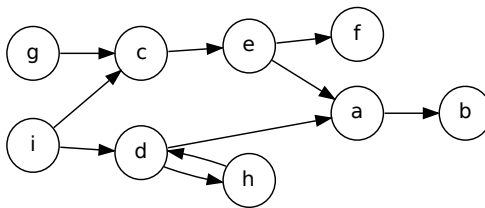


Figure 9.1: Graphical depiction of an example relation which is terminating from some starting points and not from others

could do so from other starting points. Proving that a program cannot run forever can be viewed as showing that a certain relation, namely the one which shows step-by-step how the program runs, is terminating from that program as a starting point. Mathematically, one can view the problem of termination from a starting point as just the termination problem for a version of the relation, restricted to the objects reachable from the starting point. Figure 9.1 gives a graphical example (where an edge in the graph from x to y means that x is bigger than y in the relation). The relation depicted is terminating starting from g , because all paths from g eventually terminate in either f or b . But it is not terminating starting from i , for example, because there is an infinite path starting at i : we can move from i to d and then to c and back to d again and again forever.

9.1.1 Defining termination in Agda

The file `termination.agda` in the IAL gives an Agda definition for what it means to be a terminating relation. First of all, we need to model relations in Agda. This is done by modeling a relation on a set A as a value R of type $A \rightarrow A \rightarrow \text{Set}$. Such an R takes in two elements of type A , and returns a Set . Under the Curry-Howard isomorphism on which Agda is based, a Set can just as well be viewed as a formula. So a binary relation as modeled in Agda is a function which takes two elements of A and returns a formula. Which formula is it? We should think of it as a formula which is provable iff the two elements indeed stand in the relation to one another. This is more general than modeling a relation as a function of type $A \rightarrow A \rightarrow \mathbb{B}$. To take a concrete example, the function `_<_` has type $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{B}$. We can model this as the relation

$$\lambda (x \ y : \mathbb{N}) \rightarrow (x < y) \equiv \text{tt}$$

This has type $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{Set}$, because it takes in arguments x and y of type \mathbb{N} , and then returns $(x < y) \equiv \text{tt}$, which is a Set . So any function of type $A \rightarrow A \rightarrow \mathbb{B}$ can be turned into one of type $A \rightarrow A \rightarrow \text{Set}$. The reverse is not


```

data ↓ {ℓ ℓ'} {A : Set ℓ} (⟦_>_⟧ : A → A → Set ℓ')
  : A → Set (ℓ ⊔ ℓ') where
  pf↓ : ∀ {x : A} → (∀ {y : A} → x > y → ↓ ⟦_>_⟧ y) →
    ↓ ⟦_>_⟧ x

```

no infinite chain from x → no i.c. from y

Figure 9.2: The datatype representing termination of a given relation

possible in general in Agda, however. So `termination.agda` uses the more general notion of relation as function of type $A \rightarrow A \rightarrow \text{Set}$, though there is also a helper definition for the more specific type of relation.

The first definition we find in `termination.agda` is the short but tricky one for the type \downarrow , which we will use to express termination, in Figure 9.2 For yet greater generality, we actually make this a level-generic definition, so that the type A and the binary relation $_>_$ can be at different levels ℓ and ℓ' . Constraints on Agda's infinite hierarchy of type levels require the type \downarrow to be at the level which is the maximum of ℓ and ℓ' . This maximum is denoted $\ell \sqcup \ell'$. Disregarding this rather cluttering generality, we see that the type \downarrow takes in an explicit argument $_>_$, which is the binary relation about which the \downarrow type will express a termination property. \downarrow also takes in an element of type A . This is for a starting point, from which the \downarrow statement will express termination of the relation $_>_$. For convenience in the common case that we are interested in termination of a function to \mathbb{B} , `termination.agda` has the following definition, which just wraps the boolean-valued function as a relation, as we saw above for $_>_$ on the natural numbers:

```

↓ℕ : ∀ {ℓ} {A : Set ℓ} (⟦_>_⟧ : A → A → ℤ) → A → Set ℓ
↓ℕ {ℓ} {A} ⟦_>_⟧ x = ↓{ℓ}{ℓzero} (λ (x y : A) → (x > y) ≡ tt) x

```

For a simple but important example of termination, the type

```

∀ (n : ℕ) → ↓ℕ ⟦_>_⟧ n

```

expresses the idea that for every natural number n , the greater-than relation is terminating. This is, in fact, true, and will be proved just below in `termination.agda`.

Returning to Figure 9.2: how do we express termination of $_>_$ from starting point a ? The sole constructor `pf↓` of the inductive type \downarrow shows how this is done. We can prove that $_>_$ is terminating from starting point x iff we can show that for any y such that $x > y$, we have $\downarrow _>_ y$. We see this in the following part of the input type of `pf↓`:

```

(∀ {y : A} → x > y → ↓ ⟦_>_⟧ y)

```

The idea is that if the relation is terminating starting from any point y that can be reached in one step of the $_>_$ relation from x , then it is also terminating from x . We can justify this informally by hypothesizing that the relation were not terminating from x . In that case, there must be some infinite chain $x > y > \dots$. But we are assuming that the relation is terminating from all y reachable from x in one

step with $_>_$. So any chain beginning with y where $x > y$ must be finite. This contradicts the assumption that $x > y > \dots$ is infinite. So the relation is indeed terminating from x .

This type is an interesting example of an inductive type in Agda. The sole explicit argument to the constructor $\text{pf}\downarrow$ is a function of type

$$(\forall \{y : A\} \rightarrow x > y \rightarrow \downarrow _>_ y)$$

The inductive definitions of the types for lists, natural numbers, and others we have seen previously in the book have not had constructors which take functions as arguments. Agda allows such constructors, but only so long as the type being defined (here \downarrow) is only used as the return type for the function which is to be the constructor's input. In such cases, termination of functions and hence logical consistency is preserved. One way to think of constructors with functional arguments like this is as infinitely branching nodes in a tree. If a constructor has three arguments, then this can be thought of as three subtrees. If one wishes to represent an infinite number of subtrees, then the constructive way to do this is to have a function mapping some index values to those subtrees. In this case, the function can be thought of as mapping pairs consisting of y of type A and a proof of $x > y$, to a subtree. When writing recursive functions over such a datatype, there is then no problem with making recursive calls on any of this infinite set of subtrees. Intuitively, one can understand the subtrees to be smaller in all cases, even though there are infinitely many of them. We will see below (Section 9.1.3) that Agda indeed allows recursive calls to be made on any value returned by a functional argument to a datatype constructor. This preserves termination and logical consistency.

As an aside, it is interesting to see how allowing the datatype to be used as the domain type for that functional argument causes problems. We will return to this in Section 9.1.4 below.

9.1.2 Termination of greater-than on natural-numbers

Let us see an example of proving termination, just next in `termination.agda`. We will see another, more computational example below (Section 9.2.3). This is $\downarrow->$, shown in Figure 9.3, which proves that the natural-number ordering $_>_$ (defined in `nat.agda`) is terminating, starting from any x of type \mathbb{N} . The proof is somewhat tricky. We use a recursive helper function h , defined in a `where` clause. This helper recurses on x , to show that for any smaller y , the ordering is terminating starting from y . This is the property that the $\text{pf}\downarrow$ function called in the equation for $\downarrow->$ requires of x . So how does the h helper function work? If x is 0 we can derive obvious absurdities from the assumption that x is strictly greater than y (once we case-split on y). So suppose x is actually $\text{suc } x$, for some different x . This means that our ordering assumption is actually $\text{suc } x > y \equiv \text{tt}$ in this case. Now, `nat.agda` defines $x > y$ to be just $y < x$, so h gets an assumption p

```

↓-> : ∀ (x : ℕ) → ↓B _>_ x
↓-> x = pf↓ (h x)
  where h : ∀ x → ∀ {y} → x > y ≡ tt → ↓B _>_ y
        h 0 {0} ()
        h 0 {suc y} ()
        h (suc x) {y} p with <-drop {y} p
        h (suc x) {y} p | inj₁ u rewrite u = ↓-> x
        h (suc x) {y} p | inj₂ u = h x u

```

Figure 9.3: A proof that the greater-than relation on natural numbers in terminating

of the following fact:

$y < \text{suc } x \equiv \text{tt}$

In `nat-thms.agda`, we have the following theorem that `h` now uses to do a case-split based on this assumption `p`:

```

<-drop : ∀ {x y : ℕ} → (x < (suc y) ≡ tt) →
          x ≡ y ∨ x < y ≡ tt

```

(Note that we are instantiating `x` with `y` and `y` with `x` in calling this theorem.) If `y` is less than `suc x`, either `y` equals `x`, or else `y` is less than `x`. In the former case, `h` does a rewrite to change `y` to `x` in the goal. This means we must just prove $\downarrow B _>_ x$ in this case, which is done with the outer recursive call $\downarrow-> x$. Otherwise, we can make an inner recursive call `h x u`, since the type of `u` is definitionally equal to $y > x \equiv \text{tt}$, and so we have decreased the first argument of `h` from `suc x` to just `x`.

9.1.3 Proving termination using a measure function

One classic way to prove termination of a relation $_>A_$ on some set A is using a **measure function**. We will see an interesting use of this in Section 9.2.3 below. Suppose there is some quantity we can compute from objects of type A , where that quantity decreases in a terminating relation every time we take a step with $_>A_$. Then this must mean that $_>A_$ is terminating as well. This is formally expressed in the Agda code shown in Figure 9.4, from `termination.agda`. The code declares a module, so that we can take in several parameters to be used in the theorem that comes next. In this case, it is just a matter of taste to make these parameters of a module rather than arguments to the function. One can prefer them as module parameters to avoid cluttering the statement of the `measure-↓` theorem. The module parameters are two sets A and B with binary relations $_>A_$ and $_>B_$. We also have a measure function m from A to B , and a proof `decreasem` saying that m preserves the ordering relationship: whenever $a >A a'$, we have

```

module measure {ℓ ℓ' ℓ1 ℓ2 : level}
  {A : Set ℓ} {B : Set ℓ'}
  (_>A_ : A → A → Set ℓ1)
  (_>B_ : B → B → Set ℓ2)
  (m : A → B)
  (decreasem : ∀ {a a' : A} → a >A a' →
               m a >B m a')

  where

  measure-↓ : ∀ {a : A} → ↓ _>B_ (m a) → ↓ _>A_ a
  measure-↓ {a} (pf↓ fM) = pf↓ h
    where h : {y : A} → a >A y → ↓ _>A_ y
            h {y} p = measure-↓ (fM (decreasem p))

```

Figure 9.4: Termination using a measure function

$m\ a\ >B\ m\ a'$. The theorem of interest is then that if $_>B_$ is terminating starting from $m\ a$, then $_>A_$ is terminating starting from a . The proof of this is short, but again a bit tricky. We are assuming the $_>B_$ ordering terminates starting from $m\ a$, for an arbitrary a . The proof of this termination assumption is $pf\downarrow\ fM$, for some function fM of type

$$\{y : B\} \rightarrow m\ a\ >B\ y \rightarrow \downarrow\ _>B_ y$$

Now $measure\text{-}\downarrow$ returns a proof of type $\downarrow\ _>A_ a$, by applying the $pf\downarrow$ constructor (for the \downarrow type) to a helper function h , declared in a `where` clause. This helper function must show that $_>A_$ is terminating from any starting point y which is less than a . By applying our module parameter $decreasem$ to p , we get a proof of

$$m\ a\ >B\ m\ y$$

We can apply fM to this, to obtain a proof of $\downarrow\ _>B_ (m\ y)$. Now something rather remarkable happens. We make a recursive call to $measure\text{-}\downarrow$ on the value returned by fM . From a typing perspective, everything makes sense: the call to $measure\text{-}\downarrow$ is given a proof of $\downarrow\ _>B_ (m\ y)$, and it produces a proof of $\downarrow\ _>A_ y$, which is exactly what we need to finish this case for h . But what is surprising is that this recursive call passes Agda's termination checker. This is just what we considered in Section 9.1.1 above: if a constructor contains an argument of functional type, Agda allows recursive calls to values returned by that function. This does not violate termination of recursive functions, as each such value can be viewed as smaller than the data value containing that function.

9.1.4 Nontermination with negative datatypes

The file `neg-datatype-nonterm.agda` in the IAL shows what could happen if Agda were to allow a functional argument to a datatype constructor, where the datatype is used in the domain type of the type for that functional argument. Such positions within the type are called **negative**, borrowing terminology from logic. In classical logic, an implication $A \rightarrow B$ is equivalent to $\neg A \vee B$. So A occurs negatively in the implication.

The file begins with a compiler directive telling Agda not to enforce the requirement that datatypes are used only positively.

```
{-# OPTIONS --no-positivity-check #-}
```

In fact, Agda requires **strict** positivity: the datatype simply cannot be the left of any arrow. Positivity would allow it to occur to the left of an even number of arrows (since an even number of negations cancel out). Next, we find this datatype definition:

```
data Bad : Set where
  bad : (Bad → ⊥) → Bad
```

The constructor `bad` takes in a function as an argument, and the datatype `Bad` in question is used negatively: it is the domain type in `Bad → ⊥`. It is now very easy to derive a contradiction, which we demonstrate by writing an expression of type `⊥`. This type, defined in `empty.agda`, is the empty datatype with no constructors. Under the Curry-Howard isomorphism, it corresponds to falsity.

```
badFunc : Bad → ⊥
badFunc (bad f) = f (bad f)
```

The code for `badFunc` does something funny. It pattern matches on the input of type `Bad`. There is only one constructor for `Bad`, of course, namely `bad`. So the pattern variable `f` has type `Bad → ⊥`. We just need an input of type `Bad` and we can call this function `f`. Of course, we have such an input handy, namely `bad f` itself, the input to the `badFunc` function. Now it is very easy to write down an expression of type `⊥`:

```
inconsistency : ⊥
inconsistency = badFunc (bad badFunc)
```

We just call `badFunc` on an input of type `Bad`, which we can construct from applying `bad` to `badFunc` itself. If we ask Agda to normalize (with Control-c Control-n) this `inconsistency` term, it will run forever:

```
badFunc (bad badFunc) = badFunc (bad badFunc) = ...
```

This is because what `badFunc` does when it is called with an argument matching the pattern `bad f` is call `f` on `bad f`. Here, `f` is `badFunc`, so `f (bad f)` just becomes `badFunc (bad badFunc)`, and we are back where we started, in an infinite loop.

9.2 Operational Semantics for SK Combinators

As we have seen in earlier chapters, every function defined in Agda is required to terminate on all inputs. While many, perhaps even most, useful programs consist of terminating functions, there are notable exceptions. Interpreters for Turing-complete languages constitute one. Given the proliferation of programming languages, particularly dynamically typed scripting languages, in the web-programming era, this is not a class of examples that should be ignored. We will now look at one very minimal untyped language, of so-called SK combinators. This language has important connections to the lambda calculus, which is at the heart of functional programming. We will see more of these connections below.

9.2.1 Mathematical syntax and semantics for combinators

First, let us define the syntax and operational semantics of this language mathematically. Then we will see how to formalize these definitions in Agda. The syntax is given in Figure 9.5. The notation with the $: =$ indicates that we are defining the format of abstract syntax trees for one or more syntactic categories. We will write these abstract syntax trees in linear form, and so we will need to use parentheses to show the structure of the tree. (We could also use parsing conventions, but we will just be fully explicit about parentheses to avoid confusion.)

In this case, the sole syntactic category is that of combinators. By writing a, b, c after this, we are indicating that we will denote combinators, when speaking about them, with variables that look like a, b , or c . This includes a' or a_1 and a_2 , for example. Often such variables are called **metavariables**, because they are the variables we are using in our mathematical meta-language to talk about some language we are studying. This language of study is often called the **object language**. If it is a programming language, it will almost always have its own notion of variables, and hence the distinction between object-language variables and metavariables is useful. This language of combinators, however, is one of the few cases of a variable-free programming language. The only expressions in the language of SK combinators are the special symbols S and K, and the juxtaposition $c\ c'$ of two combinators c and c' . Just as in functional programming as we have seen it in Agda, the juxtaposition notation $c\ c'$ means that combinator c is being called as a function with argument c' .

How does one program with combinators? They barely resemble any programming language we have likely seen, not even Agda. We will see later how lambda calculus, the basis of functional programming, can be implemented using only SK combinators. First, we need to know what combinator programs mean. How do we execute them? This is defined mathematically in Figure 9.6, which lists four rules explaining how combinator programs execute step by step. These rules use metavariables like a, b , and c , to indicate that the rules apply for any combinators one wishes, in place of the metavariables.

combinators $a, b, c ::= S \mid K \mid c \ c'$

Figure 9.5: Mathematical definition of the syntax for SK combinators

$$\frac{}{\overline{(K \ a) \ b} \rightsquigarrow a} \quad \frac{}{\overline{((S \ a) \ b) \ c} \rightsquigarrow (a \ c) \ (b \ c)} \quad \frac{a \rightsquigarrow a' \quad b \rightsquigarrow b'}{a \ b \rightsquigarrow a' \ b'} \quad \frac{b \rightsquigarrow b'}{a \ b \rightsquigarrow a \ b'}$$

Figure 9.6: Mathematical definition of the operational semantics for SK combinators

The first rule says that in one step of execution, the program $(K \ a) \ b$ will become just a . So $K \ a$ behaves as a constant-valued function that always returns a when called with any argument b . The next rule says that in one step, $((S \ a) \ b) \ c$ becomes $(a \ c) \ (b \ c)$. I always remember this definition by thinking of the classic rock band AC-DC (although here it is AC-BC). The S combinator is routing argument c to two combinators a and b , and applying the results. It is not intuitively obvious how this is useful, but we will see more below. The third rule says that if $a \rightsquigarrow a'$ is true, then $a \ b \rightsquigarrow a' \ b$ is also true. The fourth rule makes a similar implicational statement. So these last two rules let you apply the first two rules anywhere you want within nested applications.

Figure 9.7 shows what is called a **derivation** using the rules. This derivation is proving the fact that

$$(K \ K) \ (K \ ((K \ S) \ K)) \rightsquigarrow (K \ K) \ (K \ S)$$

What is happening here is shown by underlining the term that is being reduced, and what it reduces to:

$$(K \ K) \ (K \ \underline{((K \ S) \ K)}) \rightsquigarrow (K \ K) \ (K \ \underline{S})$$

We are really just applying the rule for K (the first one in Figure 9.6). But this is taking place in the right parts of two applications. So we have two inferences with the fourth rule of Figure 9.6.

Let us write a (very) simple example program using combinators, and see how it executes on a sample input using the rules. The program is $(S \ K) \ K$. This program

$$\frac{\frac{\frac{}{(K \ S) \ K \rightsquigarrow S}}{K \ ((K \ S) \ K) \rightsquigarrow K \ S}}{(K \ K) \ (K \ ((K \ S) \ K)) \rightsquigarrow (K \ K) \ (K \ S)}$$

Figure 9.7: Example derivation using the rules of Figure 9.6

```

data comb : Set where
  S : comb
  K : comb
  app : comb → comb → comb

```

Figure 9.8: Combinators, declared as an Agda datatype

cannot step at all using any of the rules, because S needs to be applied (in a left-nested way) to three arguments before the second rule of Figure 9.6 can be applied. So let us apply it to some argument a (just assumed to be any combinator term), and see what happens:

$$\underline{((S K) K) a} \rightsquigarrow \underline{(K a) (K a)} \rightsquigarrow a$$

The first step is by the second rule of Figure 9.6. That rule says what you can do if you see $((S a) b) c$. Here, we have an expression matching that pattern, namely $((S K) K) a$. The a , b , and c in the rule have been instantiated with K , K , and a respectively. The second step is by the first rule, where we are instantiating the a and b from the rule with a and $(K a)$, respectively. So we can see that $(S K) K$ behaves like an identity function: when applied to a , in two steps it reduced to a again. Note that we did not, in this particular case, have to apply the third or fourth rule of Figure 9.6.

9.2.2 Defining the syntax and semantics in Agda

The code in this section may be found in `combinators.agda` in the IAL. Defining the syntax for combinators is very easy. We already saw in Chapter 6 how syntax trees can be represented in Agda. The simple definition for SK combinators is given in Figure 9.8. We must use some explicit operator for applications, instead of juxtaposition, so we use a prefix `app` constructor. The $(S K) K$ combinator we saw above is then represented with this datatype in Agda as

```
app (app S K) K
```

We can define operations on these syntax trees, just as for any other datatype. One operation that we will rely on later is a size operation, to compute the size of a combinator expression:

```

size : comb → ℕ
size S = 1
size K = 1
size (app a b) = suc (size a + size b)

```

Now we come to the more interesting point, which is how to represent the operational semantics we saw in Figure 9.6 in Agda. This is done by the datatype


```

data _~_ : comb → comb → Set where
  ~K : (a b : comb) →
      (app (app K a) b) ~ a
  ~S : (a b c : comb) →
      (app (app (app S a) b) c) ~ (app (app a c) (app b c))
  ~Cong1 : {a a' : comb} (b : comb) →
      a ~ a' →
      (app a b) ~ (app a' b)
  ~Cong2 : (a : comb) {b b' : comb} →
      b ~ b' →
      (app a b) ~ (app a b')

```

Figure 9.9: Operational semantics of SK combinators, in Agda

definition in Figure 9.9. We are encoding the relation $a \leadsto a'$ using an indexed datatype, where we use the Unicode symbol (which can be typed with “\leadsto”) matching the mathematical notation.

Remember that under the Curry-Howard isomorphism, an object of type `Set` can be viewed both as a type for data, and as a formula. For example, $1 \equiv 1$ has type `Set`, and is viewed as expressing an idea, namely that 1 equals 1. With the definition in Figure 9.9, we are declaring that `_~_` takes two values of type `comb` (so, two combinators), and returns a formula. Then for each of the four rules of Figure 9.6, we have one constructor for the `_~_` datatype. These constructors are showing all the different ways to prove one of those `_~_` formulas about two combinators, just as the mathematical rules express all the possible reductions in the mathematical semantics.

Now let us see how one represents mathematical derivations as values of type $a \leadsto b$. For a very simple example, suppose we want to represent the following derivation:

$$\overline{(K\ S)\ (K\ S)} \leadsto S$$

This is an inference using the K rule of Figure 9.6. It is represented as an application of the `~K` constructor of the `_~_` datatype:

```
~K S (app K S) : app (app K S) (app K S) ~ S
```

The constructor requires values to use for a and b , and then produces a value of type `app (app K a) b ~ a`. Here, a is S and b is `app K S`, which represents $K\ S$.

For a slightly bigger derivation, consider the example from Figure 9.7. This is represented with our Agda datatype as follows:

```
~Cong2 (app K K) (~Cong2 K (~K S K))
```

The `~Cong2` constructor requires an explicit argument which is the first part of the application which we are trying to reduce by reducing the second part of the

application. That second part, and what it reduces to, are implicit arguments. We only have to supply $\sim\text{Cong2}$ with the proof that the second part of the application steps, as the last explicit argument. We can confirm (with Control-c Control-d in an emacs buffer with `combinators.agda` loaded) that the type of the above example term is the following (broken across two lines just for spacing reasons):

```
app (app K K) (app K (app (app K S) K))  $\leadsto$ 
app (app K K) (app K S)
```

This indeed represents the statement $(K K) (K ((K S) K)) \leadsto (K K) (K S)$, derived in Figure 9.7.

9.2.3 Termination for S-free combinators

Suppose a combinator c does not contain the S combinator. So it consists solely of applications and the combinator K . Then we can prove that reduction is terminating from c . The argument is actually quite straightforward, as we can see that the reduction $K a b \leadsto a$ reduces the size of the combinator term, wherever that reduction takes place. To formalize this in Agda, we will use our implementation of termination via measure functions (Section 9.1.3 above). It is very easy to express being S -free:

```
Sfree : comb  $\rightarrow$   $\mathbb{B}$ 
Sfree S = ff
Sfree K = tt
Sfree (app a b) = Sfree a && Sfree b
```

To use size as a measure function for termination, we will use the `measure- \downarrow` theorem discussed in Section 9.1.3. We must specify a type A and an ordering $_>A_$ on that type, and then give a measure function which maps A to B , for some additional type with ordering $_>B_$. Here, our rough idea is to use:

- `comb` as A ,
- $_ \leadsto _$ as $_>A_$,
- \mathbb{N} as B ,
- $_> _$ as $_>B_$, and
- `size` as the measure function.

This is not quite correct, however, as `size` is only decreased for S -free combinators. We will see how to rectify this below.

Reduction of S-free combinators decreases the size

It is not very hard to prove that if a is S-free and a reduces to b , then the size of b is strictly less than the size of a . The proof of this fact is shown in Figure 9.10. There are several things to consider here. First of all, the function `Sfree-→-size` is defined by recursion on the value of type $a \rightsquigarrow b$. After all, \rightsquigarrow is an inductive datatype, so we can recurse on its values. Those values represent derivations using the rules for the operational semantics of combinators (Figure 9.6). So we are recursing on the structure of derivations. This is a common technique in reasoning about operational semantics and other deductive systems.

For the $\rightsquigarrow_K a b$ case, Agda will deduce for us that the a in the statement of the theorem is `app (app K a) b` (the prefix representation of the mathematical expression $(K a) b$), and the b in the statement of the theorem is b . The goal is then definitionally equal to the following, because the size of `app (app K a) b` is actually three plus the sum of the sizes of a and b (one for the K and one more for each of the `app`'s):

```
size a < suc (suc (suc (size a + size b))) ≡ tt
```

The code is using a lemma `<+2` from `nat-thms.agda`, to prove

```
size a + size b < suc (suc (suc (size a + size b))) ≡ tt
```

The lemma's typing is

```
<+2 : ∀ {x y : ℕ} → x < (suc y) + x ≡ tt
```

This is getting combined by transitivity with a proof derived by the following helper lemma, also from `nat-thms.agda`:

```
≤+1 : ∀ (x y : ℕ) → x ≤ x + y ≡ tt
```

The $(\rightsquigarrow_S a b c)$ case of `Sfree-→-size` is for when the derivation of $a \rightsquigarrow b$ is by the second rule of Figure 9.6. But that case actually cannot arise, because then the a in $a \rightsquigarrow b$ is actually `app (app (app S a) b) c`, which is not S-free. The assumption that this a is S-free reduces then to `ff ≡ tt`, which Agda can see is absurd if we write the absurd pattern `()`.

Let us look at the case where the derivation is $\rightsquigarrow_{\text{Cong1}\{a\}\{a'\}} b u$. In terms of the mathematical operational semantics, this means that the derivation looks like the following:

$$\frac{a \rightsquigarrow a'}{a b \rightsquigarrow a' b}$$

By the induction hypothesis, we know that the size of a is greater than the size of a' , since if `app a b` is S-free, so is a . The latter reasoning is formalized using the boolean lemma `&&-elim` to turn a proof of `Sfree a && Sfree b ≡ tt` into a pair of proofs of `Sfree a ≡ tt` and `Sfree b ≡ tt`, respectively (of which we only need the first). Invoking the induction hypothesis is, of course, formalized in Agda by making a recursive call: `Sfree-→-size` `p1` `u`. This gives us

```

Sfree-~>-size> : ∀{a b : comb} → Sfree a ≡ tt →
                  a ~> b → size a > size b ≡ tt
Sfree-~>-size> p (λK a b) =
  ≤<-trans {size a} (≤+1 (size a) (size b))
    (<-iter-suc-trans 2 (size a + size b))
Sfree-~>-size> () (λS a b c)
Sfree-~>-size> p (λCong1{a}{a'} b u)
  with &&-elim{Sfree a} p
Sfree-~>-size> p (λCong1{a}{a'} b u) | p1 , _ =
  <+mono2 {size a'} (Sfree-~>-size> p1 u)
Sfree-~>-size> p (λCong2 a u) with &&-elim{Sfree a} p
Sfree-~>-size> p (λCong2 a u) | _ , p2 =
  <+mono1{size a} (Sfree-~>-size> p2 u)

```

Figure 9.10: Proof that reduction of S-free combinators decreases the size

size a' < size a ≡ tt

From this, we can almost complete the proof of the goal, which is

size a' + size b < size a + size b ≡ tt

We just need to apply the following monotonicity lemma for the first argument to `_+_` from `nat-thms.agda`, where `x` is instantiated with `size a'`, and then happily Agda can deduce the needed instantiations of `y` and `z`:

```

<+mono2 : ∀ {x y z : ℕ} → x < y ≡ tt →
          x + z < y + z ≡ tt

```

The proof for the case where the derivation is by the other congruence rule is similar, using a lemma for monotonicity of the second argument of `_+_`.

Reduction of an S-free combinator results in another S-free combinator

As noted at the start of Section 9.2.3, we need to show not only that reduction of S-free combinators decreases the size of the combinator, but also that reduction preserves the property of being S-free. The typing for this proof is:

```

~>-preserves-Sfree : ∀{a b : comb} → Sfree a ≡ tt → a ~> b →
                      Sfree b ≡ tt

```

The proof is similar in structure to the one we just considered in the previous section, so we omit the details. We will use this in the concluding proof below.

Concluding termination using size as a measure function

As explained above, we are proving termination of reduction of S -free combinators using a measure function. Combinators containing S can certainly increase in size under reduction. So actually, we will use our `Sfree` function to define a new type, for the S -free combinators, using a dependent product type (see Section 5.3):

```
Sfree-comb : Set
Sfree-comb =  $\Sigma$  comb ( $\lambda$  a  $\rightarrow$  Sfree a  $\equiv$  tt)
```

An element of the type `Sfree-comb` is a pair (c, p) , where c is a combinator and p is a proof that the combinator is S -free. Now we can use `Sfree-comb` as the type A for `measure- \downarrow` from `termination.agda` (discussed in Section 9.1.3). The relation `_>A_` cannot be just `_>_`, however, because that relation is just on `comb`, not `Sfree-comb`. So we have to define a helper relation that is defined on `Sfree-comb`:

```
 $\sim\sim$ -Sfree-comb : Sfree-comb  $\rightarrow$  Sfree-comb  $\rightarrow$  Set
 $\sim\sim$ -Sfree-comb (a , _) (b , _) = a  $\sim\sim$  b
```

We also need to define our measure function on `Sfree-comb`:

```
size-Sfree-comb : Sfree-comb  $\rightarrow$   $\mathbb{N}$ 
size-Sfree-comb (a , _) = size a
```

The proof that `_>_` decreases size is easily used to show that the wrapper relation `$\sim\sim$ -Sfree-comb` reduces the wrapper measure function `size-Sfree-comb`:

```
decrease-size :  $\forall$  {x y : Sfree-comb}  $\rightarrow$   $\sim\sim$ -Sfree-comb x y  $\rightarrow$ 
  size-Sfree-comb x > size-Sfree-comb y  $\equiv$  tt
decrease-size {a , u} {b , _} p = Sfree- $\sim\sim$ -size > u p
```

The following statement from `combinators.agda` opens the module `measure` (from `termination.agda`):

```
open measure {A = Sfree-comb}  $\sim\sim$ -Sfree-comb
  ( $\lambda$  x y  $\rightarrow$  x > y  $\equiv$  tt) size-Sfree-comb
  decrease-size
```

The module's parameters are being instantiated this way:

- A is `Sfree-comb`
- `_>A_` is `$\sim\sim$ -Sfree-comb`
- B is \mathbb{N}
- `_>B_` is `_>_` (the natural-number greater-than relation, from `nat.agda`)
- m (the measure function, which should map A to B) is `size-Sfree-comb`

```

Sfree-terminatesh : ∀{a : comb}{p : Sfree a ≡ tt} →
  ↓ ~-Sfree-comb (a , p) → ↓ ~- a
Sfree-terminatesh{a}{p} (pf↓ f) = pf↓ h
  where h : {y : comb} → a ~ y → ↓ ~- y
        h{y} u = Sfree-terminatesh
          (f {y , ~-preserves-Sfree p u} u)

Sfree-terminates : ∀(a : comb) → Sfree a ≡ tt → ↓ ~- a
Sfree-terminates a p =
  Sfree-terminatesh (measure-decreases (a , p))

```

Figure 9.11: The concluding proof, that reduction of S-free combinators is terminating

- `decreasem` (the proof that the measure function is decreased by `_>A_`) is `decrease-size`

From this instantiation of `measure`, we get the following theorem, which shows that the wrapper reduction relation `~-Sfree-comb` is terminating for all elements of the type `Sfree-comb` of S-free combinators:

```

measure-decreases : ∀(a : Sfree-comb) → ↓ ~-Sfree-comb a
measure-decreases a = measure-↓ (↓-> (size-Sfree-comb a))

```

Here we must use the proof `↓->` of termination of `_>_`, which we considered in Section 9.1.2. From this, it is now easy to get the final desired theorem, whose proof is shown in Figure 9.11. There is one small matter that has to be dealt with, which is going from termination of the relation `~-Sfree-comb` on `Sfree-comb` to termination of `~-` from starting point `a` where `a` is S-free. This is done in the helper lemma `Sfree-terminatesh`. The key idea, which shows up in the definition of the subsidiary function `h`, is that if `a` is S-free and reduces to `y`, then by the lemma `~-preserves-Sfree` which we noted above, `y` must also be S-free. Hence, the pair `(y , ~-preserves-Sfree p u)` used in the body of `h` is an element of type `Sfree-comb`. This enables the proof that `~-Sfree-comb` terminates to be applied. The latter proof is supplied to `Sfree-terminatesh` by a call to the `measure-decreases` function which we defined above using `measure-↓` from the `measure` module.

9.2.4 Defining lambda abstractions using combinators

Combinators seem like a pretty peculiar language, hardly recognizable as something one could use for programming. In this section, we will see that at least in theory, combinators are a fully computationally powerful language. This is because we can translate anonymous functions, as declared with λ in Agda (see

```

data varcomb : Set where
  S : varcomb
  K : varcomb
  app : varcomb → varcomb → varcomb
  var : (s : string) → varcomb

```

Figure 9.12: Combinators containing variables

Section 4.2.5), into combinators. To do this, we must first define a datatype of combinators augmented with variables. It does not matter much what type we use for variables. Here we will just use strings. The datatype `varcomb` defined in Figure 9.12 is just like the datatype for combinators (Figure 9.8), except that it has a constructor `var` for representing variables. So a term like $S\ K\ x$ would be represented as the `varcomb` value `app (app S K) (var "x")`.

Now we will see how to translate lambda abstractions into combinators. More precisely, we are going to define a function which, given a variable x and a combinator c possibly containing that variable (and possibly not containing it), returns a combinator which will definitely not have the variable x in it, and will behave just like $\lambda x. c$. An algorithm with these properties is sometimes called a **bracket abstraction** algorithm. We will just consider the simplest such algorithm; more complicated versions aim to keep the size of the returned combinator smaller (the algorithm we are considering more than doubles its size). The code is shown in Figure 9.13. If we investigate this carefully, we will start to understand better why combinators are based on the primitive operations `K` and `S`. Let us look at the first defining equation. We are supposed to return a combinator which behaves like $\lambda s. S$. This is a function which given an argument for variable s , will just ignore that argument and return `S`. That function can be described by `app K S`. To see this, consider what happens when we apply `app K S` to an argument a . We have the following reduction, using the operational semantics for combinators:

$$\text{app (app K S)} \rightsquigarrow a$$

So indeed, `app K S` behaves just like $\lambda s. S$. Something similar happens if the combinator from which we are trying to abstract the variable is `K` or is a variable different from s . This leaves the application case to consider. The equation for this from Figure 9.13 is:

$$\lambda s. \text{app } c_1\ c_2 = \text{app (app S } (\lambda s. c_1))\ (\lambda s. c_2)$$

What happens when we apply the right-hand side of this equation to an argument a ? Remember, we are supposed to show that the right-hand side behaves exactly like $\lambda s. c_1\ c_2$.

$$\begin{aligned} &\text{app (app (app S } (\lambda s. c_1))\ (\lambda s. c_2))\ a} \rightsquigarrow \\ &\text{app (app } (\lambda s. c_1)\ a)\ (\text{app } (\lambda s. c_2)\ a) \end{aligned}$$

This is by the `S` reduction rule. We can see that `S` is routing the argument a to

```

λ* : (s : string) → varcomb → varcomb
λ* s S = app K S
λ* s K = app K K
λ* s (app c1 c2) = app (app S (λ* s c1)) (λ* s c2)
λ* s (var s') =
  if (s ==string s') then (app (app S K) K)
  else (app K (var s'))

```

Figure 9.13: The function for abstracting a variable from a combinator

```

contains-var : string → varcomb → B
contains-var s S = ff
contains-var s K = ff
contains-var s (app c1 c2) =
  contains-var s c1 || contains-var s c2
contains-var s (var s') = s ==string s'

```

Figure 9.14: Function for checking whether or not a `varcomb` contains a variable

its first and second arguments, and then applying the results. We will consider this in more detail just below, as part of proving that the λ^* operation satisfies its specification.

The variable is eliminated

Figure 9.14 defines a function for checking whether or not a `varcomb` contains a given variable. Using this definition, we can now state and prove the proposition that applying the λ^* to a variable s and a combinator v returns a combinator which does not contain s . The proof is straightforward. For the `app c1 c2` case, we must use our induction hypothesis for both $c1$ and $c2$, to simplify the goal, which simplifies to

$$\text{contains-var } s \ (\lambda^* s \ c1) \ || \ \text{contains-var } s \ (\lambda^* s \ c2) \equiv \text{ff}$$

For the variable case, we have to case-split on whether the variable in question (s') equals the variable s we are abstracting with λ^* . In either case, it is trivial to prove that the resulting combinator does not contain s . The `keep` idiom is needed here (see Section 4.3.4), to retain the knowledge about whether or not s equals s' , in the context.

The returned combinator behaves like the corresponding lambda-abstraction

Now we want to prove that if you take the combinator returned by λ^* and apply it to an argument, this will behave the same way as applying the corresponding


```

λ*-binds : ∀(s : string)(v : varcomb) →
           contains-var s (λ* s v) ≡ ff
λ*-binds s S = refl
λ*-binds s K = refl
λ*-binds s (app c1 c2)
  rewrite λ*-binds s c1 | λ*-binds s c2 = refl
λ*-binds s (var s') with keep (s =string s')
λ*-binds s (var s') | tt , p rewrite p = refl
λ*-binds s (var s') | ff , p rewrite p = p

```

Figure 9.15: The proof that the function we have defined eliminates the variable

```

subst : varcomb → string → varcomb → varcomb
subst c s S = S
subst c s K = K
subst c s (app c1 c2) = app (subst c s c1) (subst c s c2)
subst c s (var s') = if (s =string s') then c else var s'

```

Figure 9.16: Substitution function for combinators with variables

lambda-abstraction to an argument. Making this statement precise requires a little work. First of all, when one applies a lambda abstraction $\lambda x.t$ to an argument t' , the rules of lambda calculus dictate that the argument should be substituted for the variable x in the expression t . So we need to define a substitution function, in Figure 9.16. This is very straightforward: to compute $\text{subst } c \ s \ c'$ for substitution of combinator c in place of variable s in combinator c' , we just descend into applications and either leave c' unchanged or, if we encounter a use of the variable s , replace it with c .

In formalizing our idea that the result of λ^* behaves just like a lambda-abstraction, there are several more issues to attend to. First of all, to say that the result behaves a certain way, we intend to talk about the reduction behavior. But the relation $_ \rightsquigarrow _$ is defined on combinators (the type `comb`), not combinators with variables (`varcomb`). Now, we could have defined a single type of combinators that might or might not have variables, by using an index to the `comb` type that would control this. But for simplicity, we have two separate types. So we need to define another relation for reduction of elements of the `varcomb` type. This is somewhat regrettable duplication, but I felt it worthwhile here to avoid cluttering the previous development. The new type is named $_ \rightsquigarrow_{vc} _$ in `combinators.agda`, and its definition is just the same as that of $_ \rightsquigarrow _$, except for the different name for the relation and the use of `varcomb` everywhere instead of `comb`. So the definition is omitted here.

Even with this new relation $_ \rightsquigarrow_{vc} _$ for `varcomb`, there is still a further problem. We will be able to show that $(\text{app } (\lambda^* \ s \ v1) \ v2)$ reduces (using $_ \rightsquigarrow_{vc} _$) to

```

module basics {ℓ ℓ' : level}{A : Set ℓ}
  (_>A_ : A → A → Set ℓ') where

  data tc : A → A → Set (ℓ ⊔ ℓ') where
    tc-step : ∀{a b : A} → a >A b → tc a b
    tc-trans : ∀{a b c : A} → tc a b → tc b c →
      tc a c

```

Figure 9.17: The definition of the transitive closure of a relation, from `closures.agda`

(subst v2 s v1). But this reduction may require several steps, not just one. So to state the theorem, we need to define a multi-step version of \sim_{vc} . We will do this with help from code in another file from the IAL, `closures.agda`. The definition we will use from `closures.agda` is shown in Figure 9.17. There is a module `basics` parametrized by a type A and a binary relation $_>A_$ on A . Then the `tc` datatype is defined. The name stands for “transitive closure”. The transitive closure of a relation R , as one may recall from discrete math, is the smallest relation which relates all the same things as R does, but is additionally transitive. A simple example is the transitive closure of the parent relation. If A is a parent of B and B is a parent of C , then A is an ancestor of C . If we visualize the relation R as a graph (as we did above in Figure 9.1), then the transitive closure of R will add an edge from x to y if there is a path of length at least 1 from x to y . The definition in Figure 9.17 defines the transitive closure (`tc`) of the relation $_>A_$ by first saying, with the type of `tc-step`, that if $a >A b$, then also $tc\ a\ b$. This expresses the idea that the transitive closure of relation R posits all the same relationships as R . Additionally, however, $tc\ a\ c$ holds if $tc\ a\ b$ and $tc\ b\ c$. This is expressed by the type of `tc-trans`. This statement clearly makes `tc` a transitive relation, which is expressed by this very short proof (in `closures.agda`):

```

tc-transitive : transitive tc
tc-transitive = tc-trans

```

Armed with `tc`, we can now define the multi-step version of \sim_{vc} . We do this by instantiating the module parameter $_>A_$ of `closures.basics` with \sim_{vc} , and then defining \sim_{vc+} to be `tc`. Because we have instantiated $_>A_$ with \sim_{vc} , `tc` is now the transitive closure of \sim_{vc} , as desired:

```

open closures.basics ~>vc_

~>vc+_ : varcomb → varcomb → Set
~>vc+_ = tc

```

Mathematically, if we have a relation \sim , then its transitive closure is standardly denoted by \sim^+ . Finally we are in a position to state the theorem we want to prove,

about the behavior of the combinator returned by λ^* . This is $\lambda^* \multimap$, a little further on in `combinators.agda`:

```
 $\lambda^* \multimap$  :  $\forall$  (v1 v2 : varcomb) (s : string)  $\rightarrow$ 
  (app ( $\lambda^*$  s v1) v2)  $\sim_{vc+}$  (subst v2 s v1)
```

This says that the combinator we get by applying $(\lambda^* s v1)$ (i.e., the combinator returned by the λ^* function) to $v2$ will reduce, after one or more steps, to $(subst v2 s v1)$. This is exactly the behavior of a lambda abstraction $\lambda s.v1$. Let us look at this proof mathematically first (though we will use Agda syntax). The proof is by induction on $v1$. Suppose $v1$ is S . Then our goal is to prove

```
(app ( $\lambda^*$  s S) v2)  $\sim_{vc+}$  (subst v2 s v1)
```

By the definition of λ^* , the combinator returned by λ^* for input s and S is `app K S`. Also, since $v1$ does not contain the variable s (because $v1$ is just the combinator S), if we substitute any value for s in $v1$, we just get $v1$ again. So $(subst v2 s v1)$ is just equal to $v1$, which is S . So in the end, our goal is definitionally equal to just

```
(app (app K S) v2)  $\sim_{vc+}$  S
```

Now we have the following single-step reduction just by the \sim_K axiom:

```
(app (app K S) v2)  $\sim_{vc}$  S
```

And we can lift any single-step reduction to a multi-step reduction. So this case is completed.

This same reasoning applies if the combinator $v1$ is K or a variable not equal to s . If $v1$ is an application `app c1 c2`, we have already seen the first step of reasoning above. We have the following definitional equality for $\lambda^* s v1$:

```
 $\lambda^* s$  (app c1 c2) = app (app S ( $\lambda^* s$  c1)) ( $\lambda^* s$  c2)
```

And then we have this reduction:

```
app (app (app S ( $\lambda^* s$  c1)) ( $\lambda^* s$  c2)) v2  $\sim_{vc}$ 
app (app ( $\lambda^* s$  c1) v2) (app ( $\lambda^* s$  c2) v2)
```

Now our induction hypothesis gives us these two facts:

```
(app ( $\lambda^* s$  c1) v2)  $\sim_{vc+}$  (subst v2 s c1)
(app ( $\lambda^* s$  c2) v2)  $\sim_{vc+}$  (subst v2 s c2)
```

It seems that it should then be straightforward to get:

```
app (app ( $\lambda^* s$  c1) v2) (app ( $\lambda^* s$  c2) v2)  $\sim_{vc+}$ 
app (subst v2 s c1) (subst v2 s c2)
```

That latter term is exactly the value of `subst v2 s (app c1 c2)`, and so we would be done (though we must connect all the reductions into a single multi-step reduction). But there is a bit of a wrinkle. We have the following congruence rules for single-step reduction (essentially those from Figure 9.6):

$$\frac{a \rightsquigarrow_{\text{vc}} a'}{a \ b \rightsquigarrow_{\text{vc}} a' \ b} \quad \frac{b \rightsquigarrow_{\text{vc}} b'}{a \ b \rightsquigarrow_{\text{vc}} a \ b'}$$

But we do not have such rules if we replace $\rightsquigarrow_{\text{vc}}$ with $\rightsquigarrow_{\text{vc}}^+$ (its transitive closure). Fortunately, those can be derived as we will see shortly. So we can indeed connect up the reductions as proposed above, completing this case of the proof.

The last case of the proof is to show

`app (λ* s (var s)) v2 $\rightsquigarrow_{\text{vc}}^+$ (subst v2 s (var s))`

The definition of λ^* in this case returns `app (app S K) K`, which we argued above behaves like the identity function. So indeed

`app (app (app S K) K) v2 $\rightsquigarrow_{\text{vc}}^+$ v2`

and since `(subst v2 s (var s))` is definitionally equal to `v2`, that completes this case of the proof.

The above mathematical reasoning is formalized in Agda in the proof of Figure 9.18. The proofs for `id \rightsquigarrow` , `trans-Cong1`, and `trans-Cong2` are given in Figure 9.19. The `id \rightsquigarrow` lemma just formalizes the argument that `app (app S K) K` behaves like the identity function. For an intuitive explanation of `trans-Cong1`, suppose we have a sequence of single steps, glued together (by `tc-trans` in Agda), showing $c_1 \rightsquigarrow^+ c_n$:

$$c_0 \rightsquigarrow c_1 \quad c_1 \rightsquigarrow c_2 \quad \cdots \quad c_{n-1} \rightsquigarrow c_n$$

Then we can get a proof of $c_0 \ b \rightsquigarrow^+ c_n \ b$ just by adding a congruence step below each of the single steps:

$$\frac{c_0 \rightsquigarrow c_1}{c_0 \ b \rightsquigarrow c_1 \ b} \quad \frac{c_1 \rightsquigarrow c_2}{c_1 \ b \rightsquigarrow c_2 \ b} \quad \cdots \quad \frac{c_{n-1} \rightsquigarrow c_n}{c_{n-1} \ b \rightsquigarrow c_n \ b}$$

This reasoning is formalized in the proof of `trans-Cong1`; the proof of `trans-Cong2` follows the same idea.

9.3 Conclusion

We have seen an example of formalizing deductive systems of interest for Computer Science within Agda. We began by formalizing what it means for a relation to be terminating. We then defined the syntax and operational semantics of SK combinators, a minimalistic variable-free programming language. The operational semantics is formalized in Agda as an indexed datatype, where mathematical derivations showing that one combinator term reduces to another are formalized as values of the datatype. Using our formalization of termination, we were rather easily able to prove that reduction starting from combinator terms not

```

λ*~> : ∀ (v1 v2 : varcomb) (s : string) →
      (app (λ* s v1) v2) ~vc+ (subst v2 s v1)
λ*~> S v2 s = lift (~K S v2)
λ*~> K v2 s = lift (~K K v2)
λ*~> (app c1 c2) v2 s =
  (tc-trans (tc-step (~S (λ* s c1) (λ* s c2) v2))
   (tc-trans (trans-Cong1
              (app (λ* s c2) v2) (λ*~> c1 v2 s))
              (trans-Cong2 (subst v2 s c1) (λ*~> c2 v2 s)))))
λ*~> (var s') v2 s with s =string s'
λ*~> (var s') v2 s | tt = id~> v2
λ*~> (var s') v2 s | ff = lift (~K (var s') v2)

```

Figure 9.18: The proof that the returned combinator behaves like a lambda-abstraction

```

id~> : ∀ (a : varcomb) →
      app (app (app S K) K) a ~vc+ a
id~> a = (tc-trans
         (tc-step (~S K K a))
         (tc-step (~K a (app K a))))

trans-Cong1 : ∀{a a' : varcomb} (b : varcomb) →
      a ~vc+ a' → (app a b) ~vc+ (app a' b)
trans-Cong1 b (tc-trans d1 d2) =
  tc-trans (trans-Cong1 b d1) (trans-Cong1 b d2)
trans-Cong1 b (tc-step d) = tc-step (~Cong1 b d)

trans-Cong2 : ∀(a : varcomb) {b b' : varcomb} →
      b ~vc+ b' → (app a b) ~vc+ (app a b')
trans-Cong2 a (tc-trans d1 d2) =
  tc-trans (trans-Cong2 a d1) (trans-Cong2 a d2)
trans-Cong2 a (tc-step d) = tc-step (~Cong2 a d)

```

Figure 9.19: Proofs of some helper lemmas for the proof in Figure 9.18

containing the combinator S is terminating. We also defined the simplest bracket abstraction algorithm, showing how programs in pure lambda calculus can be translated to combinators, and proved that algorithm correct.

9.4 Exercises

1. Use the λ^* function from `combinators.agda` (possibly in nested fashion) to translate the following lambda terms to combinators:
 - (a) $\lambda x. \lambda y. x$
 - (b) $\lambda x. \lambda y. y$
 - (c) $\lambda s. \lambda z. (s\ z)$
 - (d) $\lambda x. (x\ x)$
2. Write down the two possible Agda expressions of type $c \rightsquigarrow c'$, where c is `app (app K K) ((app K K) K)`, and c' is up to you to determine.
3. Using the transitive closure operator defined in `closures.agda`, define a value `_~>+_` of type


```
comb → comb → Set
```

 and use it to prove that `app (app K K) ((app K K) K)` multi-step reduces to `K`.
4. Look up another bracket abstraction algorithm and implement it in Agda. You will likely need to extend the definition of `comb` with new combinators, as these are typically introduced for other bracket abstraction algorithms. Compare the sizes of the combinator terms you get using the algorithm you implement with the ones obtained with λ^* .

Chapter 10

Intuitionistic Logic and Kripke Semantics

As we have seen throughout this book, Agda is based on the Curry-Howard isomorphism: constructive proofs are identified with terminating functional programs. In logic, the paradigmatic constructive logic is **intuitionistic logic**. It has this name because of its historical connection to L. E. J. Brouwer, a mathematician whose antirealist philosophy of mathematics – he held that mathematical ideas are subjective, and grounded in human construction rather than an independently existing mathematical universe – led him to oppose certain principles of logic, which are now known as **classical** in contrast (as we considered also in Section 2.1.3). For example, $P \vee \neg P$, called the law of excluded middle, is the paradigmatic non-intuitionistic validity of classical logic. Does one of necessity have a construction for either P or $\neg P$, for any proposition P ? Brouwer believed not, and certainly from both a psychological and a foundational perspective, there is some plausibility to this view.

Philosophy aside, constructive logic has interesting properties that have made it very useful for Computer Science, as we have seen with Agda. In this section, we will study the theory of intuitionistic logic, as a topic in its own right, using Agda. We will define the syntax for formulas of a particular fragment of intuitionistic logic, namely **positive propositional intuitionistic logic**. Next, we will specify which formulas are valid using proof rules, and write an Agda datatype to represent proofs written with those rules. This is straightforward, and similar to what we have done already above for operational semantics of combinators (Figure 9.6).

The next step is something new (for us in this book): we will give what is known as a **denotational semantics for formulas**. We will explain what the meanings of formulas are by explaining what kind of statements they are making about the world. For this, naturally, we need to formalize some abstractions of “the world”, since the actual world is outside our formalization within Agda. These abstractions are called **structures**. For intuitionistic logic, the structures standardly used are called Kripke structures, after their inventor, Saul Kripke. The basic idea is to capture the different possible evolutions of the world over time. Propositions that are known to be true at one moment in time must remain true in the future. But Kripke structures allow for the possibility that a proposition which is not currently known to be true can become true in the future.

We will show that the proof rules for intuitionistic logic are **sound** with respect to Kripke structures. This means that if the proof rules imply that a formula is valid, then for any Kripke structure one could possibly have, the formula will be true also in that structure. This is not a difficult theorem to prove – indeed,

soundness theorems generally are not difficult in logic. More interestingly, we will be able to prove completeness: if a formula is true in all Kripke structures, then you can also prove it with the proof rules of intuitionistic logic. We will see an amazing twist on the Curry-Howard isomorphism related to this: the composition of our soundness proof with the completeness proof yields a normalizer. This is a computational function for simplifying proofs to a normal form. Since, under the Curry-Howard isomorphism, constructive proofs are programs, we will find that we have obtained an interpreter which can, in effect, run programs. We will see that this composition of soundness with completeness gives us what is known as **normalization by evaluation (NBE)**, a method for writing a normalizer by making use of evaluation in the language in which the normalizer is being written. Thus, the Curry-Howard connection between logic and programming thus runs deeper than just identifying proofs and programs. We will see that certain metalogical proofs, namely soundness and completeness, are strongly connected to a certain metaprogram, namely an interpreter. We will consider just positive logic, but the research literature also has examples with full intuitionistic logic [11].

The code for this chapter can be found in `kripke-semantics.agda` in the IAL.

10.1 Positive Propositional Intuitionistic Logic (PPIL)

Propositional logic generally has boolean operations and propositional variables. An example is a formula like $p \rightarrow (p \vee q)$, which says that if p (whatever state of affairs this corresponds to) is true, then it is also true that either p or q is true. This is a pretty trivial validity, but propositional logic is concerned with basic matters of implicational truth, so nothing too deep should be expected. Indeed, since intuitionistic logic is a subset of classical logic, we can expect to find even more trivial theorems than those of Chapter 2! But the proofs of those theorems will be more interesting.

For our formulation of positive propositional intuitionistic logic (PPIL), we will have logical operations for truth, conjunction (“and”), and implication, as well as propositional variables. The mathematical syntax for these formulas is given in Figure 10.1. We exclude disjunction, regarding it as a negative construct, since it significantly complicates the NBE completeness proof we will see below. Negation, of course, is negative and hence omitted. Figure 10.2 then gives the Agda definition of the set of formulas as a datatype (in `kripke-semantics.agda`). We do not try to replicate the mathematical infix syntax, as we are already making use of the mathematical operators in the IAL for other operations. To represent a propositional variable p , we will use `$ "p"`. So a formula like $p \rightarrow (p \wedge p)$ (which we will see shortly is provable in PPIL) is denoted with this value in Agda:

```
Implies ($ "p") (And ($ "p") ($ "p"))
```

Figure 10.3 gives the proof rules for PPIL. The rules derive statements of the form

$$\begin{array}{l} \text{variables } p \\ \text{formulas } f ::= p \mid f \wedge f' \mid f \rightarrow f' \mid \text{True} \end{array}$$

Figure 10.1: Mathematical syntax of PPIL formulas

```
data formula : Set where
  $ : string → formula
  True : formula
  Implies : formula → formula → formula
  And : formula → formula → formula
```

Figure 10.2: Agda definition of the set of PPIL formulas

$\Gamma \vdash f$. The symbol \vdash originates in the work of the 19th-century logician and philosopher Gottlob Frege, where it was used to indicate a judgment that a proposition is true. Here, we are judging that f is true, assuming that the formulas in the list Γ are true. If Γ is empty, we will just omit it. Γ is called a **context**. The notion of truth is constructive, of course, as we discussed above.

Let us walk through the proof rules of Figure 10.3. The first rule says that we can judge f to be true assuming Γ, f . That is, if f is the rightmost assumption in the list, then we can judge it to be true. If f is not the last assumption in the list, we can still judge it to be true, with some help from the fifth rule (discussed just below). The second rule says that if we separately judge f_1 to be true assuming Γ , and also f_2 to be true assuming Γ , then we can judge $f_1 \wedge f_2$ to be true assuming Γ . The third and fourth rules say that if we judge a conjunction $f_1 \wedge f_2$ to be true assuming Γ , then we can judge f_1 to be true assuming Γ , and also (separately) f_2 . The fifth rule says that if you can judge f to be true assuming Γ , then you are also allowed to judge it to be true using Γ, f' . That is, if you proved a fact with a smaller list of assumptions Γ , then you can just as well prove it if you add some additional assumption f' . This rule is usually called **weakening**, because by adding an assumption you are weakening your judgment (it is stronger to prove f from a smaller than a bigger list of assumptions). The sixth rule says that if we can judge f_2 to be true assuming Γ, f_1 – that is, assuming all the formulas in the context Γ are true, and also assuming f_1 is true – then we are entitled to judge that the implication $f_1 \rightarrow f_2$ is true assuming Γ . The seventh rule says that if we can judge an implication $f_1 \rightarrow f_2$ to be true assuming Γ , and we can also judge f_1 to be true assuming Γ , then we can judge f_2 to be true assuming Γ . Finally, the eighth rule says that we can always judge True to be true, no matter what Γ is. For a simple example derivation using these rules, see Figure 10.4. If $\Gamma \vdash f$ can be proved using the rules, we say that the judgment $\Gamma \vdash f$ is **provably valid**.

There are several different ways to write down rules that give this same notion of (constructive) provable validity. The particular style used here is known as **natural deduction**. There are some book-keeping rules, like the weakening rule we

$$\begin{array}{c}
\frac{}{\Gamma, f \vdash f} \quad \frac{\Gamma \vdash f_1 \quad \Gamma \vdash f_2}{\Gamma \vdash f_1 \wedge f_2} \quad \frac{\Gamma \vdash f_1 \wedge f_2}{\Gamma \vdash f_1} \quad \frac{\Gamma \vdash f_1 \wedge f_2}{\Gamma \vdash f_2} \\
\\
\frac{\Gamma \vdash f}{\Gamma, f' \vdash f} \quad \frac{\Gamma, f_1 \vdash f_2}{\Gamma \vdash f_1 \rightarrow f_2} \quad \frac{\Gamma \vdash f_1 \rightarrow f_2 \quad \Gamma \vdash f_1}{\Gamma \vdash f_2} \quad \frac{}{\Gamma \vdash \text{True}}
\end{array}$$

Figure 10.3: Proof rules for PPIL

$$\frac{\frac{\frac{}{p \vdash p} \quad \frac{}{p \vdash p}}{p \vdash p \wedge p}}{\vdash p \rightarrow (p \wedge p)}$$

Figure 10.4: A proof of $p \rightarrow (p \wedge p)$ with an empty context

discussed (which is also called a *structural* rule), and the rule for using an assumption from the context (the first rule). The other rules are divided into **introduction** and **elimination** rules for the different logical operators. An introduction rule for an operator introduces that operator in the conclusion of the rule (the part below the line). So the introduction rules from the figure are these:

$$\frac{\Gamma \vdash f_1 \quad \Gamma \vdash f_2}{\Gamma \vdash f_1 \wedge f_2} \quad \frac{\Gamma, f_1 \vdash f_2}{\Gamma \vdash f_1 \rightarrow f_2} \quad \frac{}{\Gamma \vdash \text{True}}$$

An elimination rule has the operator in a premise (the parts above the line in the rule), but not in the conclusion. So the elimination rules are these:

$$\frac{f_1 \wedge f_2}{\Gamma \vdash f_1} \quad \frac{f_1 \wedge f_2}{\Gamma \vdash f_2} \quad \frac{\Gamma \vdash f_1 \rightarrow f_2 \quad \Gamma \vdash f_1}{\Gamma \vdash f_2}$$

For our Agda formalization of the provable validity relation of PPIL, we are using an inductive datatype, just as we did for the operational semantics of combinators (Figure 9.9). This is shown in Figure 10.5. To type \vdash , you can use the following in emacs:

`\ | -`

One slight mismatch between our mathematical definition and the Agda version is that formulas are added to contexts on the right in Figure 10.3, while in our Agda formulation, since we model contexts as lists of formulas (`ℒ formula`), it is more convenient to add formulas on the left, using the `_ : : _` operation on lists. Sometimes functional programmers will introduce so-called *snoc*-lists (“snoc” is “cons” backwards), where `cons` has the tail on the left and head on the right, instead of head on left, tail on right as it is with `_ : : _`. But we will not trouble ourselves with this.

```

ctxt : Set
ctxt =  $\mathbb{L}$  formula

data  $\vdash$  : ctxt  $\rightarrow$  formula  $\rightarrow$  Set where
  assume :  $\forall\{\Gamma\ f\} \rightarrow (f :: \Gamma) \vdash f$ 
  weaken :  $\forall\{\Gamma\ f\ f'\} \rightarrow \Gamma \vdash f \rightarrow (f' :: \Gamma) \vdash f$ 
  ImpliesI :  $\forall\{\Gamma\ f1\ f2\} \rightarrow (f1 :: \Gamma) \vdash f2 \rightarrow$ 
              $\Gamma \vdash (\text{Implies } f1\ f2)$ 
  ImpliesE :  $\forall\{\Gamma\ f1\ f2\} \rightarrow \Gamma \vdash (\text{Implies } f1\ f2) \rightarrow$ 
              $\Gamma \vdash f1 \rightarrow \Gamma \vdash f2$ 
  TrueI :  $\forall\{\Gamma\} \rightarrow \Gamma \vdash \text{True}$ 
  AndI :  $\forall\{\Gamma\ f1\ f2\} \rightarrow \Gamma \vdash f1 \rightarrow \Gamma \vdash f2 \rightarrow \Gamma \vdash (\text{And } f1\ f2)$ 
  AndE :  $\forall(b : \mathbb{B})\{\Gamma\ f1\ f2\} \rightarrow \Gamma \vdash (\text{And } f1\ f2) \rightarrow$ 
          $\Gamma \vdash (\text{if } b \text{ then } f1 \text{ else } f2)$ 

```

Figure 10.5: Agda formalization of PPIL's provable validity, as an inductive datatype

The names for the rules are chosen to indicate whether they are introduction (I) or elimination (E) rules. One small optimization is that rather than having two elimination rules for conjunction, for whether one wants to deduce the left or the right conjunct, we instead have a single rule that uses a boolean argument b to determine which conjunct should be deduced. The example proof from Figure 10.4 is represented in Agda using this datatype as follows, where we have supplied just enough implicit arguments so that Agda can tell without further guidance that this proof proves $[] \vdash \text{Implies } (\$ "p") (\text{And } (\$ "p") (\$ "p"))$:

```

ImpliesI{ $\$ "p"$ } (AndI (assume{[]}) (assume))

```

10.2 Kripke structures

Next, we will consider a mathematical definition of Kripke semantics for PPIL, and how this semantics is formalized in Agda. As introduced above, the basic idea of Kripke semantics is to represent the world as evolving into different possible futures. Indeed, Kripke structures will consist of a set of worlds, connected by a binary relation representing something like future possibility. In any given world, some of our propositional variables may be designated as true. Variables which are true in a world must remain true when we pass to future possible worlds. But variables which are not designated as true can either remain so, or else become designated as true in a later possible world. For example, Figure 10.6 shows an example Kripke structure where there are three worlds, labeled 1, 2, and 3. In world 1, no variables are true. In world 2, p is true while in world 3, p remains untrue. Incidentally, for a full propositional intuitionistic logic including negation,

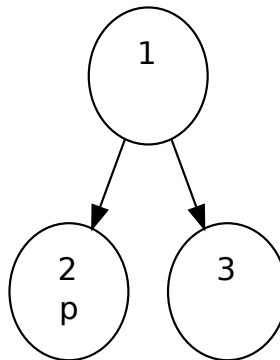


Figure 10.6: A simple Kripke structure

world 1 of this Kripke structure falsifies the classical principle $p \vee \neg p$. The meaning of $p \vee \neg p$ is that either p is true now, in which case it must be true in all future worlds; or else p is false and remains false in all future worlds (that is the semantics of negation). In world 1 of the structure in Figure 10.6, however, p is not true, so the first disjunct of $p \vee \neg p$ is not validated. But neither is the second disjunct validated, because p becomes true in world 2 (though it remains false in world 3).

Now we must give a precise mathematical definition of Kripke semantics. A Kripke structure consists of several mathematical objects. First, there is a set W of worlds. These can be any mathematical objects one wishes, and the set may be finite or infinite. Next, there is a binary relation R on W . This relation is required to be reflexive and transitive, expressing, respectively, the ideas that the current world is trivially a future possible world, and that a future world of a future world is a future world. The definitions of reflexivity and transitivity may be found in `relations.agda`; we considered them in some detail in Section 5.4.1. The next constituent part of a Kripke structure is a relation V which holds for a world w and a propositional variable p just in case p is designated as true in w . This V is required to be **monotonic** with respect to the ordering on worlds: if p is designated (by V) to be true in world w and if world w' is a future possible world of w , then p must also be designated to be true in w' . So, as we said above, variables which are designated as true in a world must remain true in all future possible worlds.

So for our example Kripke structure of Figure 10.6, the set W of worlds is just $\{1, 2, 3\}$, and the relation is the one depicted, except that the figure has left off edges from each node to itself. Such edges are required by reflexivity, but may be omitted in drawing Kripke structures, since they must always be there. Set-

```

record struct : Set1 where
  field W      : Set
        R      : W → W → Set
        preorderR : preorder R
        V      : W → string → Set
        monoV   : ∀ { w w' } → R w w' → ∀ { i } →
                  V w i → V w' i

  reflR : reflexive R
  reflR = fst preorderR
  transR : transitive R
  transR = snd preorderR

```

Figure 10.7: The definition of Kripke structure in Agda

theoretically, we can describe the relation for this example as

$$\{(1,1), (2,2), (3,3), (1,2), (1,3)\}$$

The relation V in this case can be described set-theoretically as just $\{(2,p)\}$, since 2 is the only world where any variable is designated true. This relation is trivially monotonic, as there are no future worlds for 2 in this structure.

Now let us see how to formalize Kripke structures in Agda. The definition comes next in `kripke-semantics.agda`, and is shown in Figure 10.7. We are using Agda’s record notation to represent a structure. A record in Agda is a type for values containing named fields. Here, the fields are W for the set of worlds, R for the binary relation expressing future possibility, `preorderR` for the proof that R is a preorder (a reflexive and transitive relation), V for which variables are designated true in which worlds, and `monoV` for the proof that V is monotonic with respect to R . Records in Agda can also contain definitions, and here we just project out proofs `reflR` of reflexivity and `transR` of transitivity, from `preorderR`. This is just for more convenient access later on. Agda views fields of records as functions which take an element of the record type, and return the value of the field for that record. For example, the W field is viewed as a function of type `struct → Set`. To use a member of a record – either a field or a defined value – one uses member-selection syntax $r.f$, where r is the record type and f is the member name. So `struct.W` is the function which, given a record of type `struct`, returns the type W for worlds in that record. A record in Agda can be opened (as is done for `struct` a little further in `kripke-semantics.agda`), much like a module. This makes the field names available without the “ r .” prefix.

For an example, consider the structure shown in Figure 10.8. It is a little tedious to define this as an element of type `struct` in Agda, but this can be done. The code, from `kripke-semantics.agda`, may be seen in Figures 10.9 and 10.10. This is a simple example of mathematical modeling in Agda. We declare a datatype `world` to represent the structure’s set of worlds. We next represent the structure’s

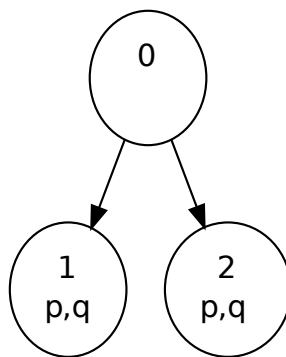


Figure 10.8: Kripke structure for an example of the semantics

future possibility relation by an indexed datatype `rel`. We just list the positive cases, where one world is indeed related to another; negative cases are implicitly negative in virtue of not having a `rel` constructor. So for example, we have a constructor `r00` expressing the fact that world 0 is related to itself. We do not have a constructor of type `rel w2 w0`; the absence of this constructor implicitly represents the negative fact that world 2 is not related to world 0. It is easy to prove by pattern-matching on the world that this relation `rel` is reflexive (this is the `rel-refl` proof of Figure 10.9). To prove transitivity, we are assuming we have world w related to world w' , which is in turn related to w'' . Pattern matching on the proofs of these relations, as is done in the definition of `rel-trans` (Figure 10.9) provides a concise and elegant way of considering all the cases for the relation, without explicitly considering negative cases.

Figure 10.10 uses another inductive datatype `val` to express which variables are true in which worlds of the structure. Again, we only have to list positive information. It is easy to prove that `val` is monotonic, if we case split on the proof of `rel w w'` (as done in `mono-val` of Figure 10.10). This `mono-val` function is given a proof of `rel w w'`, and a proof of `val w i`, where i is a variable (of type `string`), and required to produce a proof of `val w' i`. Since there are no variables true in world 0, we have two absurd cases in the definition of `mono-val`. The other cases are trivial since w' equals w in those cases. Finally, we can define the actual structure of interest, as a value of type `struct`. This uses Agda's syntax for record literals: we have the `record` keyword, and then in curly braces, semicolon-separated equations giving values for the fields.

```
data world : Set where
  w0 : world
  w1 : world
  w2 : world

data rel : world → world → Set where
  r00 : rel w0 w0
  r11 : rel w1 w1
  r22 : rel w2 w2
  r01 : rel w0 w1
  r02 : rel w0 w2

rel-refl : reflexive rel
rel-refl {w0} = r00
rel-refl {w1} = r11
rel-refl {w2} = r22

rel-trans : transitive rel
rel-trans r00 r00 = r00
rel-trans r00 r01 = r01
rel-trans r00 r02 = r02
rel-trans r11 r11 = r11
rel-trans r22 r22 = r22
rel-trans r01 r11 = r01
rel-trans r02 r22 = r02
```

Figure 10.9: Agda formalization of the Kripke structure of Figure 10.8, first part

```

data val : world → string → Set where
  v1p : val w1 "p"
  v1q : val w1 "q"
  v2p : val w2 "p"
  v2q : val w2 "q"

mono-val : ∀{w w'} → rel w w' →
           ∀{i} → val w i → val w' i
mono-val r00 p = p
mono-val r11 p = p
mono-val r22 p = p
mono-val r01 ()
mono-val r02 ()

k : struct
k = record { W      = world ;
             R      = rel ;
             preorderR = (rel-refl , rel-trans) ;
             V      = val ;
             monoV   = mono-val }

```

Figure 10.10: Agda formalization of the Kripke structure of Figure 10.8, second part

10.3 Kripke semantics for PPIL

Now that we have defined what Kripke structures are mathematically and in Agda, we can proceed to give a semantics for PPIL formulas, in terms of these structures. We will define what it means for a PPIL formula f to be true at a world w of a Kripke structure k . We will just give the definition in Agda, in Figure 10.11; the mathematical definition is so similar there is no need to write it out separately. We are defining a function $_, _ \models _$, which takes a `struct` (Kripke structure), an element of $\mathbb{W} \ k$, and a formula, and returns a type. So this is an example of **type-level computation**, as introduced in Chapter 7. The element of $\mathbb{W} \ k$ is the world with respect to which we are expressing that the formula is true. This $\mathbb{W} \ k$ notation is an example of the record member-selection syntax mentioned in the previous section: \mathbb{W} is a field of the record type `struct`, and since we have opened `struct` just above in `kripke-semantic.agda`, we can apply it (without having to write `struct.W`) to an element (i.e., a record) of the record type to get the value of that field of the record. The type that is returned by $_, _ \models _$ expresses that the formula is true in the world. If this type is provable (i.e., there is something which has that type), then the formula is considered true in that world of that structure; and otherwise it is not.

Let us walk through the definition. If we have a propositional variable x , it is true in world w just in case the V function of `struct` k says it is. Again, we are using member-selection syntax $V \ k$ to get the V function out of `struct` k ; we can then apply that function to its arguments w and x . The formula `True` is true, which we represent by the type `T` (from `unit.agda` in the IAL), which can be proved by the value `triv`, which is the sole constructor of the datatype `T`. The equation for the semantics of an implication `Implies f1 f2` is the most interesting part of the definition. It says that `Implies f1 f2` is true in world w iff for all future worlds w' , if $f1$ is true in w' , then so is $f2$. This is the only part of the definition where the future possibility relation R plays a role. The semantics is trying to capture the idea that for an implication to be true in the current world w , it must be necessarily true: in any future world, whenever the first part of the implication is true, the second part has to be true. Remember that the relation R is required to be reflexive, so w is also included under this quantification over future worlds w' . The semantics for conjunctions `And f1 f2` then just says that each conjunct is true separately.

As an example, we can prove, using this semantics, that the formula

```
Implies ($ "p") ($ "q")
```

is true in world $w0$ of the `struct` k which we defined in the previous section (Figure 10.9 and 10.10). This proof, which comes next in `kripke-semantic.agda` is shown in Figure 10.12. If we look at the depiction of this Kripke structure in Figure 10.8, it is easy to confirm that in every future possible world w of 0 (i.e., 0, 1, and 2), if p is true in that world, then so is q . If w is world 0, neither p nor q are designated true (by the V relation), so the statement in question is vacuously true. In w is world 1 or 2, p and q are both designated true, also validating the

```

_,_|=_ : ∀(k : struct) → W k → formula → Set
k , w |= ($ x) = V k w x
k , w |= True = T
k , w |= Implies f1 f2 = V {w' : W k} → R k w w' →
                                k , w' |= f1 → k , w' |= f2
k , w |= And f1 f2 = k , w |= f1 ∧ k , w |= f2

```

Figure 10.11: Agda definition of the semantics of PPIL formulas with respect to a Kripke structure

```

pf-test-sem : k , w0 |= Implies ($ "p") ($ "q")
pf-test-sem r00 ()
pf-test-sem r01 p = v1q
pf-test-sem r02 p = v2q

```

Figure 10.12: Agda proof that the sample formula $p \rightarrow q$ is true in world 0 of the structure from Figure 10.8 (this structure is formalized in Figures 10.9 and 10.10)

statement.

The first thing to understand about the Agda proof of Figure 10.12 is the type for `pf-test-sem`. Let us load `kripke-semantics.agda` and then request normalization (with `Control-c Control-n`) of the type of `pf-test-sem`:

```
k , w0 |= Implies ($ "p") ($ "q")
```

Agda will tell us this type normalizes to the following (although Agda will insert “`|=-example.`” throughout, since the definition is inside a module called `|=-example`):

```
{w' : world} → rel w0 w' → val w' "p" → val w' "q"
```

This is just the specialization of our general semantics for formulas from Figure 10.11, to the specific formula `Implies ($ "p") ($ "q")`. To prove this, the code in Figure 10.12 is taking in a proof that `w0` is related to implicit argument `w'`, and then a proof that “`p`” is designated true in world `w'`. It must produce a proof that “`q`” is also designated true in world `w'`. The code does a case split on the proof that `w` is related to world `w0`. So we have cases for if this proof is `r00`, `r01`, or `r02`. The vacuity we just considered in our informal confirmation shows up in the case for `r00`, as in that case, Agda can see that provability of `val w0 "p"` is absurd. So we can use the absurd pattern `()`. In the other two cases, we must just use the appropriate constructor, either `v1q` or `v2q`, to show that “`q`” is designated true in either world `w1` or `w2`, respectively.

In addition to the semantics of formulas, we need a simple and obvious extension of the semantics, to contexts. This is shown in Figure 10.13. A context Γ (which is just a list of formulas) is true in a world w of a Kripke structure k iff each of

```

_,_|=ctxt_ : ∀(k : struct) → W k → ctxt → Set
k , w |=ctxt [] = T
k , w |=ctxt (f :: Γ) = (k , w |= f) ∧ (k , w |=ctxt Γ)

```

Figure 10.13: Extending the semantics of PPIL formulas to contexts

the formulas in that context is. The notation is $k , w \models_{\text{ctxt}} \Gamma$. The code just recurses on the context, constructing a conjunctive Agda formula (i.e., a type) by calling $_,_ \models_{\text{ctxt}}$ for each formula in the context.

10.4 Soundness of PPIL

Now that we have defined the syntax and proof rules of PPIL (Section 10.1), formalized Kripke structures (Section 10.2), and given a semantics of PPIL formulas in terms of Kripke structures (Section 10.3), we are ready to study the relationship between provability on the one hand, and truth in a Kripke structure on the other. The first result is **soundness**: if $\Gamma \vdash f$ is provable in PPIL, then in any Kripke structure, in any world of that structure where all the formulas in Γ are true, f will also be true. This property is called soundness because it expresses the idea that whatever PPIL tells us is valid really does hold in any world of any Kripke structure. Soundness generally is the property that what is claimed to be true is, in fact, true. Soundness is usually not a hard theorem to prove in logic, and that is the case here for PPIL. We do need one helper lemma first, which is **monotonicity**: in any Kripke structure, if a formula is true in world w_1 and world w_2 is a future possible world of w_1 , then the formula is also true in w_2 . This extends monotonicity for propositional variables, which is required for all Kripke structures, to arbitrary formulas.

10.4.1 Monotonicity proof

The Agda proof of monotonicity is given in Figure 10.14. The proof is by induction on the formula f in question. The Kripke structure k and its worlds w_1 and w_2 are implicit arguments to $\text{mono}\models$. Let us walk through the cases. If f is just a use of a propositional variable x , then monotonicity holds by the monoV assumption which is part of the Kripke structure (and accessed, as discussed above, by applying the monoV field name to the record k). If f is `True`, then $k , w_2 \models f$ (which we are required to prove) is definitionally equal to `T`, so we can just return `triv`, the sole constructor for the `T` type. Now if f is `Implies f1 f2`, then our goal $k , w_2 \models f$ is definitionally equal to the following, by the definition of $_,_ \models_{\text{ctxt}}$:

$$\forall \{w' : W k\} \rightarrow R k w w' \rightarrow k , w' \models f_1 \rightarrow k , w' \models f_2$$

Agda allows us to include patterns for these additional inputs in the equation for $\text{mono}\models$. If we ask Agda for the types for all the local variables by putting a hole on the right-hand side of that equation and then typing “Control-c Control-,” we will see the following:

```

p'   : k , .w'  $\models$  f1
r'   : R k .w2 .w'
.w'  : W k
p    : {w' : W k}  $\rightarrow$  R k .w1 w'  $\rightarrow$ 
      k , w'  $\models$  f1  $\rightarrow$  k , w'  $\models$  f2
r    : R k .w1 .w2
f2   : formula
f1   : formula
.w2  : W k
.w1  : W k
k    : struct

```

We see that the implicit w' from the meaning of the `Implies` formula has been introduced to the context (as a dotted variable, as we have seen in earlier examples like in Section 3.8.1). We are trying to prove that $f2$ is true in world w' , and we have an assumption p that for any world related (by R) to $.w1$, if $f1$ is true in that world, then so is $f2$. We also have an assumption p' saying that $f1$ is true in world $.w'$. So as long as we can prove that $.w'$ is a future possible world of $.w1$, we can apply p . And the latter can be done, since we know from r that $.w1$ has $.w2$ as a future possible world, and from r' that $.w2$ has $.w'$ as a future possible world. So combining r and r' with transitivity (`transR`), which is part of the Kripke structure, we can conclude that case. Notice that in this case of $\text{mono}\models$, we do not make a recursive call to $\text{mono}\models$. The semantics of `Implies` statements has already included monotonicity, in the very meaning of an implicational formula.

Finally, if f is a conjunction `And f1 f2`, then the semantics of `And` formulas says that $k , w1 \models f$ is

$$k , w1 \models f1 \wedge k , w1 \models f2$$

So the defining equation for $\text{mono}\models$ for this case has the pattern $(p1 , p2)$, since `_^_` in the IAL is the type for pairs. We then return a pair, constructed by making recursive calls to $\text{mono}\models$ to move the truth of $f1$ and $f2$ from world $w1$ to world $w2$, as required. This concludes our walkthrough of the proof $\text{mono}\models$.

Where we need monotonicity in the proof of soundness below, we actually need it for contexts, not just single formulas. But it is very easy to prove this now, as shown in Figure 10.15. We just recurse through the context, applying $\text{mono}\models$ for each formula.

```

mono|= : ∀{k : struct}{w1 w2 : W k}{f : formula} →
      R k w1 w2 →
      k , w1 |= f →
      k , w2 |= f
mono|={k} {f = $ x} r p = monoV k r p
mono|={k} {f = True} r p = triv
mono|={k} {f = Implies f1 f2} r p r' p' =
  p (transR k r r') p'
mono|={k} {f = And f1 f2} r (p1 , p2) =
  mono|={f = f1} r p1 , mono|={f = f2} r p2

```

Figure 10.14: Monotonicity theorem for the PPIL semantics

```

mono|=ctxt : ∀{k : struct}{Γ : ctxt}{w1 w2 : W k} →
      R k w1 w2 →
      k , w1 |=ctxt Γ →
      k , w2 |=ctxt Γ
mono|=ctxt{k}{[]} _ _ = triv
mono|=ctxt{k}{f :: Γ} r (u , v) =
  mono|={k}{f = f} r u , mono|=ctxt{k}{Γ} r v

```

Figure 10.15: Monotonicity for the truth of a context

10.4.2 Soundness proof

Monotonicity is an interesting property of the semantics in its own right, but our reason for proving it is that it is needed in the proof of soundness, found in Figure 10.16. To state the soundness property, it is convenient to introduce the definition of $_ \Vdash _$ (also found in Figure 10.16):

```

_⊧_ : ctxt → formula → Set1
Γ ⊧ f = ∀{k : struct}{w : W k} →
      k , w |=ctxt Γ → k , w |= f

```

$\Gamma \Vdash f$ holds iff for each Kripke structure k and every world w of k , if the context Γ is true in w , then the formula f is, too. In logic, this relation is standardly called **entailment**: if all the formulas in Γ are true (in a certain world), then f is also. With this definition, soundness of PPIL is easy to express:

```

Soundness : ∀{Γ : ctxt}{f : formula} → Γ ⊢ f → Γ ⊧ f

```

This says that assuming $\Gamma \vdash f$ is true – i.e., f is provable from assumptions Γ using the PPIL rules of inference from Figure 10.3 (formalized in Agda in Figure 10.5) – then Γ entails f .

Now let us walk through the proof, looking at the goal with its context for each right-hand side. We will omit some obvious typings from those contexts, to help

focus our attention on the essential aspects of the proof. For the `assume` case, the goal is

Goal: $.k, .w \models .f$

$g : \Sigma (.k, .w \models .f) (\lambda x \rightarrow .k, .w \models_{\text{ctxt}} .\Gamma)$

We have to prove that f is true in world $.w$. We have an assumption g saying that $k, w \models_{\text{ctxt}} \Gamma$, where Γ has been refined to $.f :: .\Gamma$, because the PPIL proof in this case is an `assume`. The type of g is then equal, by the definition of `_,_|=ctxt_to`

$(.k, .w \models .f) \wedge (k, .w \models_{\text{ctxt}} .\Gamma)$

which has been obscured a bit in the listing of the context, due to the fact that simple products (with `_&_`) are defined in terms of dependent products (with `Σ`) in the IAL. Just taking the first projection of the pair, with the `fst` function of `product.agda`, is sufficient to prove the goal. The case for `weaken` similarly involves manipulating the assumption g . The goal in this case is:

Goal: $.k, .w \models .f$

$g : \Sigma (.k, .w \models .f') (\lambda x \rightarrow .k, .w \models_{\text{ctxt}} .\Gamma)$

$.w : W .k$

$p : .\Gamma \vdash .f$

We have a proof p of $.\Gamma \vdash .f$, and we also can get a proof that $.\Gamma$ is true in world $.w$, just by applying `snd` to obtain the second component of the pair g . So we can now make a recursive call to `Soundness`, to obtain the desired proof of $.k, .w \models .f$.

As for the proof of monotonicity, the trickiest cases concern `Implies` formulas. Suppose the PPIL proof we are taking as input to `Soundness` is an `ImpliesI` proof.

Goal: $k, .w' \models .f2$

$u' : k, .w' \models .f1$

$r : R k .w .w'$

$g : k, .w \models_{\text{ctxt}} \Gamma$

$p : (.f1 :: \Gamma) \vdash .f2$

We need to prove that $.f2$ is true in world w' . We have a PPIL proof p of

$(.f1 :: \Gamma) \vdash .f2$

We will want to make a recursive call to `Soundness` on this proof, with w' as the world. This requires us to prove that $(.f1 :: \Gamma)$ is true in w' . By the definition of `_,_|=ctxt_`, that just means we need to prove that $.f1$ is true in w' , and also Γ . The assumption u' tells us that $.f1$ is true in world w' , so that requirement is met. The assumption g tells us that Γ is true, but in world $.w$, not world $.w'$. So

```

_⊨_ : ctxt → formula → Set1
Γ ⊨ f = ∀{k : struct}{w : W k} → k , w ⊨ctxt Γ → k , w ⊨ f

Soundness : ∀{Γ : ctxt}{f : formula} → Γ ⊢ f → Γ ⊨ f
Soundness assume g = fst g
Soundness (weaken p) g = Soundness p (snd g)
Soundness{Γ} (ImpliesI p) g r u' =
  Soundness p (u' , mono⊨ctxt r g)
Soundness (ImpliesE p p') {k} g =
  (Soundness p g) (reflR k) (Soundness p' g)
Soundness TrueI g = triv
Soundness (AndI p p') g = (Soundness p g , Soundness p' g)
Soundness (AndE tt p) g = fst (Soundness p g)
Soundness (AndE ff p) g = snd (Soundness p g)

```

Figure 10.16: Soundness theorem for PPIL

we seem to be stuck. But this is where monotonicity ($\text{mono}\models_{\text{ctxt}}$) comes in: we have an assumption r that $R\ k\ .w\ .w'$. So applying monotonicity for contexts to r and g gives us a proof of $k\ ,\ .w' \models_{\text{ctxt}} \Gamma$. Then we can make our recursive call to `Soundness` to complete the case.

For the `ImpliesE` case, the goal is:

Goal: $k\ ,\ .w \models .f$

```

-----
g   : k , .w ⊨ctxt .Γ
p'  : .Γ ⊢ .f1
p   : .Γ ⊢ Implies .f1 .f

```

To understand how the proof of `Soundness` is working in this case, let us ask Agda what the type of `Soundness p g` is (by putting that expression in a hole on the right-hand side of this equation, and typing “Control-c Control-.”):

```

{w' : W k} → R k .w w' → k , w' ⊨ .f1 → k , w' ⊨ .f

```

This says that for any future possible world w' of w , if we can prove `.f1` is true in w' , we can conclude that `.f` is true in that world. This is good, because if we apply `Soundness` to the assumption p' (and also g), we will get a proof that `.f1` is true in world $.w$. We just need to use reflexivity of R to get $R\ k\ .w\ .w$, and then we will have values for the inputs to `Soundness p g` shown above. This concludes the `ImpliesE` case. The cases for `TrueI`, `AndI`, and `AndE` are easy applications of the semantics, so we skip walking through the details.

$$\frac{}{\overline{\Gamma \leq \Gamma}} \quad \frac{\Gamma \leq \Gamma', f}{\Gamma \leq \Gamma'}$$

Figure 10.17: The relation of context extension

10.5 Completeness

The soundness theorem we just proved says that if a formula \mathfrak{f} is provable in PPIL from assumptions Γ , then \mathfrak{f} is true in any world where Γ is also true, for any Kripke structure. Now we will prove the converse of this: if \mathfrak{f} is true in all worlds where Γ is, for all Kripke structures, then there is a PPIL proof of $\Gamma \vdash \mathfrak{f}$. The method we will use for this proof comes from **normalization by evaluation (NBE)**, first explicitly used for a proof of completeness of an intuitionistic logic by Catarina Coquand [5] (see [6] for a survey of related results). First we will look at the completeness proof in detail, and then (Section 10.5.4) explore how the proof method is related to normalization.

10.5.1 A universal structure

As just noted, our goal is to show that if \mathfrak{f} is true in all worlds where Γ is, for all Kripke structures, then there is a PPIL proof of $\Gamma \vdash \mathfrak{f}$. The proof of completeness uses a clever construction to prove a statement that appears stronger. We will construct a single special structure \mathbb{U} , and prove what we can call completeness for \mathbb{U} : if \mathfrak{f} is true in a world where Γ is, in the structure \mathbb{U} , then there is a PPIL proof of $\Gamma \vdash \mathfrak{f}$. In other words, our general statement of completeness talks about truth in all structures, while the seemingly stronger statement of completeness for \mathbb{U} talks just about truth in \mathbb{U} . We will see that in fact, truth in \mathbb{U} coincides with truth in all structures! So completeness for \mathbb{U} only appears stronger than completeness. In fact, they are equivalent. Following Ilik, we will call \mathbb{U} a **universal** structure [11]. In the study of modal logic, a similar situation arises with what are called *canonical* structures.

This universal structure \mathbb{U} we will construct has a strong connection to PPIL proofs built in to the structure itself. The worlds of the structure are contexts Γ . Then Γ' is defined to be a future possible world for Γ if Γ' extends Γ – denoted $\Gamma \leq \Gamma'$ – in the sense defined by the mathematical rules of Figure 10.17. These rules are formalized as an Agda datatype in Figure 10.18, where we grow the context (in the type of `\leq -cons`) on the left instead of on the right, as we did for the formalization of PPIL proofs above. It is very easy to prove two simple properties of the `\leq` which we will need shortly. The properties are transitivity, and a form of weakening; see Figure 10.19.

We are now ready to define our universal structure \mathbb{U} , in Figure 10.20. \mathbb{U} is ascribed


```

data _≤_ :  $\mathbb{L}$  formula  $\rightarrow$   $\mathbb{L}$  formula  $\rightarrow$  Set where
  ≤-refl :  $\forall \{ \Gamma \} \rightarrow \Gamma \leq \Gamma$ 
  ≤-cons :  $\forall \{ \Gamma \Gamma' f \} \rightarrow \Gamma \leq \Gamma' \rightarrow \Gamma \leq (f :: \Gamma')$ 

```

Figure 10.18: Agda definition of context extension

```

≤-trans :  $\forall \{ \Gamma \Gamma' \Gamma'' \} \rightarrow \Gamma \leq \Gamma' \rightarrow \Gamma' \leq \Gamma'' \rightarrow \Gamma \leq \Gamma''$ 
≤-trans u ≤-refl = u
≤-trans u (≤-cons u') = ≤-cons (≤-trans u u')

weaken≤ :  $\forall \{ \Gamma \Gamma' \} \{ f : \text{formula} \} \rightarrow \Gamma \leq \Gamma' \rightarrow$ 
           $\Gamma \vdash f \rightarrow \Gamma' \vdash f$ 
weaken≤ ≤-refl p = p
weaken≤ (≤-cons d) p = weaken (weaken≤ d p)

```

Figure 10.19: Transitivity of context extension, and weakening with context extension

the type `struct` (defined in Figure 10.7 above), and defined using Agda's syntax for record literals. The type `W` of worlds is the type of contexts (lists of formulas). The future possibility relation `R` is the context extension relation `_≤_`. We can prove that relation is a preorder because it is reflexive by definition (`≤-refl`), and we proved transitivity already (`≤-trans`). The next field of the record is `V`, which designates the propositional variables that are true in a world. In `U`, a propositional variable is true in a world Γ iff it is provable from assumptions Γ . The reason for this definition of `V` is that it makes provability in PPIL and truth in the structure `U` coincide when the formula in question is a propositional variable. This will form the base case for an inductive proof that provability and truth in `U` coincide for any formula. Given that this is the definition of `V`, we must now prove a monotonicity lemma, which becomes a special case of `weaken≤` which we proved in Figure 10.19. Agda is able to infer this specialized type for the lambda abstraction of `weaken≤` given as the definition of `monoV` (but Agda cannot infer the correct type if we just put `weaken≤` as the definition for `monoV`).

```

U : struct
U = record { W = ctxt ;
             R = _≤_ ;
             preorderR = ≤-refl , ≤-trans ;
             V =  $\lambda \Gamma n \rightarrow \Gamma \vdash \$ n$  ;
             monoV =  $\lambda d p \rightarrow \text{weaken} \leq d p$  }

```

Figure 10.20: The universal structure `U`

10.5.2 Completeness and soundness with respect to \mathcal{U}

We are now ready for the intricate definitions at the heart of the completeness proof, shown in Figure 10.21. We are simultaneously proving a form of soundness and a form of completeness, with respect to \mathcal{U} . `CompletenessU` says that if f is true in world Γ of \mathcal{U} (remember that worlds in \mathcal{U} are actually contexts Γ), then f is provable from assumptions Γ . This is the property we were discussing above: we will prove truth in \mathcal{U} implies provability. But in order to prove this, it turns out, rather surprisingly, that we must simultaneously prove a form of soundness. `SoundnessU` says that if f is provable from assumptions Γ , then f is true in world Γ in \mathcal{U} . It is important to notice how this statement of soundness differs from the property we proved in Section 10.4.2. The form of soundness proved there is the following, where we are expanding the definition of $_, _ \models_{\text{ctxt}} _$:

$$\begin{aligned} \text{Soundness} : \forall \{\Gamma : \text{ctxt}\} \{f : \text{formula}\} \rightarrow \Gamma \vdash f \rightarrow \\ \forall \{k : \text{struct}\} \{w : \mathcal{W} \ k\} \rightarrow k, w \models_{\text{ctxt}} \Gamma \rightarrow \\ k, w \models f \end{aligned}$$

If we specialized that to our structure \mathcal{U} and Γ as the particular world w , this would say:

$$\begin{aligned} \text{Soundness} : \forall \{\Gamma : \text{ctxt}\} \{f : \text{formula}\} \rightarrow \Gamma \vdash f \rightarrow \\ \mathcal{U}, \Gamma \models_{\text{ctxt}} \Gamma \rightarrow \\ \mathcal{U}, \Gamma \models f \end{aligned}$$

This is similar to the `SoundnessU` property we are proving here (Figure 10.21), but different in one critical respect: in order for it to imply `SoundnessU`, `Soundness` would require a proof of

$$\mathcal{U}, \Gamma \models_{\text{ctxt}} \Gamma$$

We will, in fact, prove this below, as a lemma `ctxt-id`. But that lemma will rely on `SoundnessU`. It turns out to be quite simple, if subtle, to define `SoundnessU` directly.

Let us now walk through the proofs of `CompletenessU` and `SoundnessU` in detail. One of the most important aspects of these proofs is that they proceed by induction on the structure of the formula in both cases. So we have cases for variables $\$ \ x$, `True`, conjunctions `And $f \ f'$` , and implications `Implies $f \ f'$` . This is different from the proof of `Soundness` of Section 10.16, which was by induction on the structure of the proof of $\Gamma \vdash f$. For `CompletenessU`, the only reasonable target for induction is the formula f , because the type $\mathcal{U}, \Gamma \models f$ is computed by type-level computation from f . We will consider the significance of this structure for the induction in Section 10.5.4 below.

Let us look at the base cases for `CompletenessU` and `SoundnessU` first together. As noted above, the definition for the \vee component of \mathcal{U} , stating when a propositional variable is true in a world, has been tailored exactly to make both base cases trivial. If f is a propositional variable $\$ \ p$, then $\mathcal{U}, \Gamma \models f$ is definitionally equal to $\Gamma \vdash \$ \ p$. For `CompletenessU` we have an input of type $\mathcal{U}, \Gamma \models \$ \ p$

```

CompletenessU : ∀{f : formula}{Γ : W U} →
  U , Γ ⊨ f → Γ ⊢ f
SoundnessU : ∀{f : formula}{Γ : W U} →
  Γ ⊢ f → U , Γ ⊨ f
CompletenessU {$ x} u = u
CompletenessU {True} u = TrueI
CompletenessU {And f f'} u =
  AndI (CompletenessU{f} (fst u))
      (CompletenessU{f'} (snd u))
CompletenessU {Implies f f'}{Γ} u =
  ImpliesI
    (CompletenessU {f'}
      (u (≤-cons ≤-refl) (SoundnessU {f} (assume {Γ})))))
SoundnessU {$ x} p = p
SoundnessU {True} p = triv
SoundnessU {And f f'} p =
  SoundnessU{f} (AndE tt p) , SoundnessU{f'} (AndE ff p)
SoundnessU {Implies f f'} p r u =
  SoundnessU (ImpliesE (weaken≤ r p) (CompletenessU {f} u))

```

Figure 10.21: Soundness and completeness with respect to the structure U

and must produce an output $\Gamma \vdash \$ p$, whereas for `SoundnessU` it is the other way around. In either case, the output is just equal to the input, because of this fact that $U , \Gamma \models \$ p$ is definitionally equal to $\Gamma \vdash \$ p$.

The cases for when f is `True` are trivial for both `CompletenessU` and `SoundnessU`. In the former case we are required to prove $\Gamma \vdash \text{True}$, which is done just with `TrueI` (Figure 10.5). In the latter, we have to prove $U , \Gamma \models \text{True}$, which is definitionally equal to \top , and hence proved with the sole constructor `triv` for that type.

The cases for when f is a conjunction `And f f'` are also easy. For `CompletenessU`, the type $U , \Gamma \models (\text{And } f \ f')$ of input u is definitionally equal (by the definition of $_ \models _$) to

$$(U , \Gamma \models f) \wedge (U , \Gamma \models f')$$

So to obtain a proof of $\Gamma \vdash \text{And } f \ f'$, the `CompletenessU` proof just makes a recursive call on `fst u`, of type $U , \Gamma \models f$, and `snd u`, of type $U , \Gamma \models f'$; and glues the resulting proofs of $\Gamma \vdash f$ and $\Gamma \vdash f'$ together with the `AndI` proof rule (Figure 10.5). The recursive calls are legal, because the type has decreased from `And f f'` to `f` in the first call and `f'` in the second. The case for conjunction in the `SoundnessU` proof is similar. Here, our input p is a proof of $\Gamma \vdash \text{And } f \ f'$. By applying `AndE tt` and `AndE ff` to this p , we obtain proofs of $\Gamma \vdash f$ and $\Gamma \vdash f'$, respectively. Since the types have decreased, as they did for `CompletenessU`, we can make recursive calls to get proofs of $U , \Gamma \models f$

and $U, \Gamma \models f'$. These can be glued together with a comma (the constructor for $_ \wedge _$), to produce a proof of

$$(U, \Gamma \models f) \wedge (U, \Gamma \models f')$$

which, as noted above, is definitionally equal to $U, \Gamma \models \text{And } f \ f'$, the type of the result we are supposed to generate.

All these cases have been pretty straightforward. We now come to the two tricky cases, which are for implications $\text{Implies } f \ f'$ for both CompletenessU and SoundnessU . If we look carefully at those cases in Figure 10.21, we can notice that each case makes a recursive call to CompletenessU and to SoundnessU . This explains why the two proofs must be defined by mutual recursion: they cannot be separated, because each must call the other, though this happens only in the case for implications.

Let us look first at the case for implications $\text{Implies } f \ f'$ for CompletenessU . We have an input u of type $U, \Gamma \vdash \text{Implies } f \ f'$, which by the definitions of both U and $_ \vdash _$ is equal to

$$\forall \{\Gamma' : \text{ctxt}\} \rightarrow \Gamma \leq \Gamma' \rightarrow U, \Gamma' \vdash f \rightarrow U, \Gamma' \vdash f'$$

The code for the right-hand side makes the following call to this u , as part of what it is doing to return an output of the required type $\Gamma \vdash \text{Implies } f \ f'$:

$$(u \ (\leq\text{-cons} \ \leq\text{-refl}) \ (\text{SoundnessU } \{f\} \ (\text{assume } \{\Gamma\})))$$

There is an implicit instantiation of Γ' to $f :: \Gamma$ here. So the two arguments being given to u must have these two types, respectively:

- $\Gamma \leq (f :: \Gamma)$
- $U, (f :: \Gamma) \vdash f$

The first type is easily proved by $\leq\text{-cons} \ \leq\text{-refl}$ (see Figure 10.18). For the second, we can easily obtain a proof of $U, (f :: \Gamma) \vdash f$ using the assume proof rule for PPIL. Calling SoundnessU translates this from a PPIL proof (with $_ \vdash _$) to truth in U (with $_, _ \models _$), as required here. Note that the call is legal, because the formula f has decreased from $\text{Implies } f \ f'$ to just f here. The call to u has thus produced a result of type

$$U, (f :: \Gamma) \models f'$$

By recursively applying CompletenessU , we obtain a proof of

$$(f :: \Gamma) \vdash f'$$

The recursive call is legal, because the formula in question has decreased from $\text{Implies } f \ f'$ to f' . Finally, we just apply the ImpliesI proof rule to turn our derivation of $(f :: \Gamma) \vdash f'$ into one for the required $\Gamma \vdash \text{Implies } f \ f'$.

Now let us look at the case for implications $\text{Implies } f \ f'$ for SoundnessU . We take in a proof of $\Gamma \vdash (\text{Implies } f \ f')$, and we must produce a proof of

```

Goal: record
  { W =  $\mathbb{L}$  formula
  ; R =  $\_ \leq \_$ 
  ; preorderR =  $\leq$ -refl ,  $\leq$ -trans
  ; V =  $\lambda \Gamma n \rightarrow \Gamma \vdash \$ n$ 
  ; monoV =  $\lambda \{.w\} \{.w'\} d \{.i\} \rightarrow \text{weaken} \leq d$ 
  }
  , .w'  $\models$  f'
-----
u   : record
  { W =  $\mathbb{L}$  formula
  ; R =  $\_ \leq \_$ 
  ; preorderR =  $\leq$ -refl ,  $\leq$ -trans
  ; V =  $\lambda \Gamma n \rightarrow \Gamma \vdash \$ n$ 
  ; monoV =  $\lambda \{.w\} \{.w'\} d \{.i\} \rightarrow \text{weaken} \leq d$ 
  }
  , .w'  $\models$  f
r   : . $\Gamma \leq .w'$ 
p   : . $\Gamma \vdash \text{Implies } f f'$ 

```

Figure 10.22: The goal for the case for implications of SoundnessU

$U, \Gamma \models (\text{Implies } f f')$. By the definitions of $_ \models _$ and U , this means we must produce a function of the following type:

$$\forall \{w' : \text{ctxt}\} \rightarrow \Gamma \leq w' \rightarrow U, w' \models f \rightarrow U, w' \models f'$$

Agda lets us bring those extra explicit arguments into the left-hand side (but not, it seems, the implicit argument if we wish to give a different name to the context w'). That is why the pattern for this case of `SoundnessU` has additional explicit inputs `r` and `u`. If we replace the right-hand side of that equation with a hole and use “Control-c Control-,” to see the current goal, Agda will print the information in Figure 10.22 (plus omitted typings for some of the variables, including implicit $. \Gamma$ and $.w'$). The big record literal is just `U`. So we see we must prove $U, .w' \models f'$ from these assumptions:

- $u : U, .w' \models f$
- $r : .\Gamma \leq .w'$
- $p : .\Gamma \vdash \text{Implies } f f'$

We have some powerful tools we can apply to these assumptions, in the form of recursive calls to `SoundnessU` and `CompletenessU`. Using `CompletenessU` on `u`, we can get a proof of $.w' \vdash f$. This is allowed, because the type has decreased from `Implies f f'` to `f`. We should now be able to combine this, using the `ImpliesE` proof rule, with our proof of $. \Gamma \vdash \text{Implies } f f'$. The only issue is that the context is different: in one case we have $.w'$, and in the other $. \Gamma$.

But because we have a proof that $.w'$ extends $.\Gamma$ (i.e., a proof of $.\Gamma \leq .w'$), we can apply `weaken \leq` to transport our proof p in context $.\Gamma$ to the context $.w'$. That is why on the right-hand side of the equation for `SoundnessU` for this case (`Implies f f'`), we are applying `ImpliesE` to (`weaken \leq r p`) and then to (`CompletenessU {f} u`). This gives us a proof of $.w \vdash f'$. We need only call `SoundnessU` recursively on this to obtain the desired conclusion $U, .w' \models f'$. This recursive call is legal, because we have decreased the formula on which we are recursing from `Implies f f'` to f' . This completes the walk-through of the proofs in Figure 10.21

10.5.3 Concluding completeness and universality

The proofs in Figure 10.21 are the heart of the proof of completeness of PPIL inspired by normalization by evaluation. The final proof of `Completeness` is in Figure 10.23, using the entailment relation $_ \Vdash _$ we defined above in Figure 10.16. The proof is very easy. We first must prove the easy lemma `ctxt-id` we mentioned above, saying that in the universal structure U , Γ is true in world Γ . This is shown in Figure 10.23. `Completeness` takes in a proof p of $\Gamma \Vdash f$. This expression is definitionally equivalent to

$$\forall \{k : \text{struct}\} \{w : W\ k\} \rightarrow k, w \models_{\text{ctxt}} \Gamma \rightarrow k, w \models f$$

The proof of `Completeness` instantiates k with U and w with Γ , which gives us a value of type

$$U, \Gamma \models_{\text{ctxt}} \Gamma \rightarrow U, \Gamma \models f$$

This is applied to a proof of $U, \Gamma \models_{\text{ctxt}} \Gamma$, which we obtain from our helper lemma `ctxt-id`. So we get, from this application of p , a proof of

$$U, \Gamma \models f$$

We apply `CompletenessU` to this, to obtain $\Gamma \vdash f$, as required.

Figure 10.23 also includes proofs `Universality1` and `Universality2`, which express the idea that U is a universal structure. `Universality1` says if Γ entails f ($\Gamma \Vdash f$), then f is true in world Γ of U . This is easily proved by first applying `Completeness` to get a proof of $\Gamma \vdash f$, and then applying `SoundnessU`. `Universality2` is the converse: f is true in world Γ of U , then Γ entails f . This is proved by first using `CompletenessU` to get a proof of $\Gamma \vdash f$, and then `Soundness` to turn this into a proof of $\Gamma \Vdash f$.

10.5.4 Composing soundness and completeness

Consider the rather shocking definition in Figure 10.24. It seems both pointless and daring. Pointless because, unlike the proofs we have been studying in this chapter, the type is trivial from a logical perspective. It just says that if f is provable

```

ctxt-id : ∀{Γ : ctxt} → U , Γ ⊨ctxt Γ
ctxt-id{[]} = triv
ctxt-id{f :: Γ} =
  SoundnessU{f} assume ,
  mono⊨ctxt (≤-cons ≤-refl) (ctxt-id {Γ})

Completeness : ∀{Γ : ctxt}{f : formula} →
  Γ ⊨ f → Γ ⊢ f
Completeness{Γ} p = CompletenessU (p{U}{Γ} (ctxt-id{Γ}))

Universality1 : ∀{Γ : ctxt}{f : formula} →
  Γ ⊨ f → U , Γ ⊨ f
Universality1{Γ}{f} p = SoundnessU (Completeness{Γ}{f} p)

Universality2 : ∀{Γ : ctxt}{f : formula} →
  U , Γ ⊨ f → Γ ⊨ f
Universality2{Γ}{f} p = Soundness (CompletenessU{f}{Γ} p)

```

Figure 10.23: Proofs of completeness and universality

```

nbe : ∀ {Γ f} → Γ ⊢ f → Γ ⊢ f
nbe {Γ} p = Completeness (Soundness p)

```

Figure 10.24: The definition of the nbe function

from assumptions Γ , then so it is. The definition seems daring, because it is not treating the proofs of `Soundness` and `Completeness` as existing simply to establish theorems. Here we are calling them as functions to perform some operation that must be of only computational significance (as the type is logically trivial). So we are really computing with the soundness and completeness proofs. What is it, exactly, that we are computing? The answer is surprising: the function `nbe` is a normalizer for PPIL proofs. It takes in a PPIL proof as input, and returns one that has been simplified in certain natural ways. Let us walk through a few examples in detail. Doing so will help us understand better how composing soundness and completeness results in normalization by evaluation.

Example a

It is easy to prove `True` in PPIL: one just uses `TrueI`. But there are other ways to prove `True`, involving unnecessary detours in reasoning. For example, consider `a` defined in the `tests` module at the end of `kripke-semantic.agda`:

```
a : [] ⊢ True
a = AndE tt (AndI TrueI TrueI)
a' = nbe a
```

The mathematical version of this proof is:

$$\frac{\frac{\frac{}{\vdash \text{True}} \quad \frac{}{\vdash \text{True}}}{\vdash \text{True} \wedge \text{True}}}{\vdash \text{True}}$$

This is clearly an unnecessarily complex proof. There is no need to introduce a conjunction and then eliminate it. If we ask Agda what the value of `a'` is (which is defined, as we see above, as the result of calling `nbe` on `a`), we will get just `TrueI`. So `nbe` has taken the proof with the detour, and removed that detour to give the simplified proof `TrueI` of $\vdash \text{True}$.

Example b

Another example of a detour in proving $\vdash \text{True}$ is `b` just below that place in `kripke-semantic.agda`:

```
b : [] ⊢ True
b = ImpliesE (ImpliesE (ImpliesI (ImpliesI (assume)))
               TrueI)
               TrueI
b' = nbe b
```


The mathematical proof corresponding to this is:

$$\frac{\frac{\frac{\overline{\text{True}, \text{True} \vdash \text{True}}}{\text{True} \vdash \text{True} \rightarrow \text{True}}}{\vdash \text{True} \rightarrow (\text{True} \rightarrow \text{True})} \quad \frac{}{\vdash \text{True}}}{\vdash \text{True} \rightarrow \text{True}} \quad \frac{}{\vdash \text{True}}}{\vdash \text{True}}$$

Like our example a, this proof is unnecessarily complicated. We introduce two implications only to immediately eliminate them. The `nbe` function again returns just `TrueI` for this case. In fact, if we look more carefully at what `nbe` is doing in these two cases, we will see that for any proof of `True`, `nbe` will just return `TrueI`. This is because `nbe` calls `Completeness`, which in turn calls `CompletenessU` with `True` for the formula `f`. In this case, `CompletenessU` does not even look at the proof `u` it is given of $\mathcal{U}, \Gamma \models \text{True}$. It always just returns `TrueI`.

Example c

Since `nbe` always returns `TrueI` given any proof of `True` (with any set of assumptions Γ), let us look at a different example. Suppose we prove $p \rightarrow p$ via the following roundabout mathematical proof:

$$\frac{\frac{\frac{\overline{p, p \vdash p}}{p \vdash p \rightarrow p} \quad \frac{}{p \vdash p}}{p \vdash p}}{\vdash p \rightarrow p}$$

This is quite flabby reasoning, as we do not need to introduce and eliminate the implication $p \rightarrow p$ in the middle of the proof. The direct mathematical proof is simply:

$$\frac{\overline{p \vdash p}}{\vdash p \rightarrow p}$$

For example `c` towards the bottom of `kripke-semantics.agda`, we give the Agda form of the flabby proof, and then call `nbe` on it:

```
c : [] ⊢ (Implies ($ "p") ($ "p"))
c = ImpliesI (ImpliesE (ImpliesI assume) assume)
c' = nbe c
```

If we ask Agda for the value of `c'` (with `Control-c Control-n`), we get

```
ImpliesI assume
```

This is the Agda version of the direct proof! Remarkable. How did `nbe` produce this? The application of `Soundness` in `nbe` turns the `ImpliesI` inferences into Agda functions. This can be seen in the code for `Soundness` (Figure 10.16), in the

equation for if the proof is an `ImpliesI` proof. The `Soundness` proof takes in two extra arguments in this case:

```
Soundness{Γ} (ImpliesI p) g r u' =
  Soundness p (u' , mono|=ctxt r g)
```

We could just as well have written this in Agda as

```
Soundness{Γ} (ImpliesI p) g =
  λ r u' → Soundness p (u' , mono|=ctxt r g)
```

The latter makes it explicit that an `ImpliesI` proof is being turned into an Agda lambda-abstraction. And in the clause of `Soundness` for `ImpliesE`, we can see that `ImpliesE` inferences are being turned into Agda applications:

```
Soundness (ImpliesE p p') {k} g =
  (Soundness p g) (reflR k) (Soundness p' g)
```

We apply `(Soundness p g)` to `(Soundness p' g)` (with a proof about the future possibility relation interposed). So the PPIL proof `c` is being translated to an Agda expression that looks roughly like

$$\lambda.((\lambda.\text{assume}) \text{assume})$$

There is a function call `((λ.assume) assume)`, where we apply an Agda λ -abstraction to an argument. In fact, the uses of `assume` are translated into variables by `Soundness`. (If you are interested, you can look up *de Bruijn* indices, and you will see that where we use `assume` this is like de Bruijn index 0, and where we apply `weaken` it is like adding one to the de Bruijn index.) So the Agda lambda term will be roughly (ignoring additional arguments for the future possibility relation)

$$\lambda x.((\lambda y.y) x)$$

Here we see one crucial ingredient of normalization by evaluation. We have turned a proof with a detour into an Agda expression that can simplify by running Agda code. Indeed, it will simplify to just $\lambda x.x$. Now how does the rest of the code for `nbe` convert this back to `ImpliesI assume`? Well, the next thing that happens in the code for `nbe` is that we call `Completeness`. The context in this case is empty, so we are just calling `CompletenessU` on what is essentially $\lambda x.x$, with the type `(Implies ($ "p") ($ "p"))`. `Soundness` recurses on the proof, but `CompletenessU` recurses on the formula. So the case for `Implies` formulas will be used:

```
CompletenessU {Implies f f'} {Γ} u =
  ImpliesI
    (CompletenessU {f'}
      (u (≤-cons ≤-refl) (SoundnessU {f} (assume {Γ}))))
```

We see that an `ImpliesI` proof will be returned. Its subproof will be one generated by a sequence of calls. We first call `SoundnessU` on `assume`, with formula `$ "p"`. Looking at the code for `SoundnessU`, we can see that if the formula is a

```

d : [ $ "q" ] ⊢ (Implies ($ "p") ($ "q"))
d = ImpliesI (ImpliesE (ImpliesI (weaken (weaken assume)))
               assume)
d' = nbe d

```

Figure 10.25: Example d for normalization by evaluation

propositional variable like this, then the proof that is input to `SoundnessU` will be the one which is returned as output. So `(SoundnessU {f} (assume {Γ}))` just returns `assume`. Next `u` is applied to this (and an argument for the future possibility relation), but `u` is essentially just the identity function $\lambda x.x$, as we discussed above. So this application of `u` produces just `assume`. Finally, `CompletenessU` is called on that, with the formula being `$ "p"`. This is because `Implies f f'` for this call to `CompletenessU` that we are considering is instantiated to

```
Implies ($ "p") ($ "p")
```

so `f'` is `$ "p"`. And `CompletenessU` returns the given proof if the formula is a propositional variable, just as `SoundnessU` does. So we get `ImpliesI assume` from `nbe` for the example proof `c`.

Example d

Suppose our proof is making use of an assumption in the context. Our example d demonstrates this, for the following mathematical proof:

$$\frac{\frac{\frac{q, p, p \vdash q}{q, p \vdash p \rightarrow q} \quad \frac{}{q, p \vdash p}}{q, p \vdash q}}{q \vdash p \rightarrow q}$$

This is again quite unnecessarily complex. The direct proof is:

$$\frac{\frac{}{q, p \vdash q}}{q \vdash p \rightarrow q}$$

Once again, `nbe` can simplify the complicated proof to the simpler one. With the definitions in Figure 10.25, `d'` returns `ImpliesI (weaken assume)`. The proof `weaken assume` has type `($ "p") :: ($ "q") :: [] ⊢ ($ "q")`, since the `weaken` skips over the `($ "p")` in the context.

Example e

The `nbe` function will not always shorten PPIL proofs. Let us see an example:

```
e : [] ⊢ (Implies (And ($ "p") ($ "q"))
              (And ($ "p") ($ "q")))
e = ImpliesI assume
e' = nbe e
```

This corresponds to the following mathematical proof:

$$\frac{\overline{(p \wedge q) \vdash (p \wedge q)}}{\vdash (p \wedge q) \rightarrow (p \wedge q)}$$

If we ask Agda to normalize e' for us, we will see it is

```
ImpliesI (AndI (AndE tt assume) (AndE ff assume))
```

This is longer than `ImpliesI assume`, which we gave for e . It corresponds to this mathematical proof:

$$\frac{\frac{\overline{(p \wedge q) \vdash (p \wedge q)}}{(p \wedge q) \vdash p} \quad \frac{\overline{(p \wedge q) \vdash (p \wedge q)}}{(p \wedge q) \vdash p}}{\frac{(p \wedge q) \vdash (p \wedge q)}{\vdash (p \wedge q) \rightarrow (p \wedge q)}}$$

Now surely this proof exhibits more meandering reasoning than the one we gave for e . What is going on? The answer is that due to the way `SoundnessU` works, `nbe` puts PPIL proofs into a normal form where assumptions of compound formulas (i.e., formulas which are not just propositional variables) are expanded out in a certain way, into more complex proofs. `SoundnessU` is always called on `assume` to start with. For conjunctions, it applies `AndE` both ways to the assumption, and pairs up (with an Agda pair) the two results. For implications, it builds an Agda lambda-abstraction whose body applies `ImpliesE` to the assumption. So `SoundnessU` decorates `assume` with possibly several nested eliminations, following the structure of the formula. This decorated version of `assume` is then passed, where `SoundnessU` is called in the code for `CompletenessU`, to a function (something of type $U, \Gamma \models (\text{Implies } f \ f')$). For this example e , the function is just the identity function, since that is the lambda-abstraction which `Soundness` returns for the PPIL proof `ImpliesI assume` which we have given for e . Then `CompletenessU` is called on the result. In this case, `CompletenessU` is just building a `AndI` proof from the two components of the pair that `SoundnessU` produced. So we get an introduction (`AndI`) with eliminations (`AndE`) for the subproofs.

Example f

```
f : [] ⊢ (Implies (Implies ($ "p") ($ "q"))
              (Implies ($ "p") ($ "q")))
f = ImpliesI assume
f' = nbe f
```

Here we try the same example as `e`, but with implications instead of conjunctions. The `nbe` function returns a proof with similarly expanded uses of `assume` (here, the `assume` in question is modified with `weaken` because of the intervening second `ImpliesI`):

```
ImpliesI (ImpliesI (ImpliesE (weaken assume) assume))
```

10.6 Conclusion

In this chapter, we have seen an extended example of mathematical modeling in Agda. We defined positive propositional intuitionistic logic (PPIL), and gave its standard Kripke semantics. Kripke semantics envisions the mathematical universe as an unfolding evolution, where different future possible worlds are connected formally by a preorder to past worlds. We proved soundness and completeness for PPIL proofs with respect to this semantics. Every formula which is provable using the PPIL proof rules is true in all worlds of all Kripke structures (soundness), and vice versa (completeness). Astonishingly, composing the proofs of soundness and completeness gives a normalizer, which can put PPIL proofs into a certain canonical form. This method is known as normalization by evaluation. PPIL proofs can, by the Curry-Howard isomorphism, be seen as functional programs. So the normalizer is not just of interest for simplifying proofs. It is essentially an interpreter for lambda calculus. And this seems a fitting point to end the main material for this book.

10.7 Exercises

1. Write out proofs of some simple example PPIL theorems like

$$(p \wedge (q \wedge r)) \rightarrow ((p \wedge q) \wedge r)$$

Write mathematical proofs first, and then translate these to Agda using the datatype of Figure 10.5.

2. Call the `nbe` function on the example proofs you wrote for the previous exercise. Make sure you can see how the results resemble the ones we saw in Section 10.5.4.
3. Find a Kripke structure where the following formula is not true. First draw the Kripke structure, and then define it as a record of type `struct`, as defined in Figure 10.7.

$$((p \rightarrow q) \rightarrow p) \rightarrow p$$

This formula is known as Peirce's law. It is classically valid but not intuitionistically valid. It is an interesting example of such a formula, because it uses only implication (as opposed to formulas like $p \vee \neg p$).

Quick Guide to Symbols

Here is a list of some Unicode symbols we have encountered in this book, together with key sequences to type them in `emacs`. The table also tells which `.agda` file to find them in, and their meaning, in the library we have studied (or in Agda itself). For a few of these, you need to follow the directions given in the appendix section “Some Extra Emacs Definitions” to configure `emacs` to accept the key sequences. In some cases, you may need to hit the space bar to force the selection of the particular default symbol from a list of possibilities. `emacs` will display this list at the very bottom of the window, and you can also use numbers to select a symbol in the list. Note that if you select a number, that choice becomes the new default, until you restart `emacs` (just choose a different number to change the default again). For many characters c , the sequence `_c` will print a subscripted c .

Sequence	Symbol	File	Meaning
<code>\r</code>	\rightarrow	built-in	for function types
<code>\all</code>	\forall	built-in	for all (universal quantification)
<code>\Gl</code>	λ	built-in	for anonymous functions
<code>\ell</code>	ℓ		symbol usually used for type levels
<code>_1</code>	1		subscript used in symbols
<code>_2</code>	2		subscript used in symbols
<code>\bb</code>	\mathbb{B}	bool	type for booleans
<code>\bn</code>	\mathbb{N}	nat	type for natural numbers
<code>=\bn</code>	$= \mathbb{N}$	nat	equality test for \mathbb{N}
<code>\eq</code>	\equiv	eq	equality type
<code>\.-</code>	\div	nat	subtraction for \mathbb{N}
<code>\le</code>	\leq	nat	less-than or equal to for \mathbb{N}
<code>\div</code>	\div	nat-division	division for \mathbb{N}
<code>\bl</code>	\mathbb{L}	list	type constructor for lists
<code>[]</code>	$[]$	list	constructor for empty list
<code>::</code>	$::$	list	cons constructor for lists
<code>++</code>	$++$	list	append two lists

Sequence	Symbol	File	Meaning
\top	\top	unit	top type
\eqn	\neq	neq	disequality type
\bv	\mathbb{V}	vector	vector type
\bot	\perp	empty	empty type
\GS	Σ	product	dependent product type
\and	\wedge	product	nondependent product type
\x	\times	product	nondependent product type
\uplus	\uplus	sum	sum type
\or	\vee	sum	sum type
\d	\downarrow	termination	termination of a relation

Commonly Used Emacs Control Commands

Figure 10.26 shows some commonly used control commands for working with Agda in `emacs`. These are specific to Agda files. Figure 10.27 show some general `emacs` commands which are very useful for editing any text, including Agda code, in `emacs`. Files are viewed in `emacs` in **buffers**. Many of the commands reference the cursor, which is the single square character marking your current position in the current buffer.

Command	Action
Control-c Control-l	Check the current <code>.agda</code> file
Control-c Control-d	Enter an expression, then Agda will type check it
Control-c Control-n	Enter an expression, then Agda will normalize it
Control-c Control-,	In a hole, see the simplified goal and context
Control-c Control-.	Like Control-c Control-, but also check an expression in the hole
Control-c Control-r	Try to solve a hole with an expression in the hole
Control-c Control-c	In a hole, case-split on a variable (or several separated by spaces like <code>x y z</code>)
Control-c Control-k	See information on the character at the cursor (requires the extra emacs code in the next section)

Figure 10.26: Agda-specific emacs control commands

Command	Action
Control-x Control-f	Enter the name of a file, which <code>emacs</code> will then open
Control-x Control-s	Save the current buffer
Control-x 2	Split the view of the current <code>emacs</code> frame
Control-x 1	Make the current frame fill the whole window
Control-x o	Move your cursor to the next visible frame
Control-x b	View another open buffer in the current frame
Control-g	Cancel whatever <code>emacs</code> is currently doing
Control-p	Move up a line
Control-n	Move down a line
Control-f	Move right one character
Control-b	Move left one character
Alt-f	Move right one word
Alt-b	Move left one word
Control-a	Move to the start of the line
Control-e	Move to the end of the line
Control-l	Recenter around the cursor
Control-k	Cut from the cursor to the end of the line
Control-y	Paste the most recently cut text
Control-space	Set the “point” (an implicit mark in the file)
Control-w	Cut the text between the cursor and the point
Alt-w	Copy the text between the cursor and the point
Control-o	Insert a newline

Figure 10.27: General emacs control commands

Some Extra Emacs Definitions

Here is some emacs lisp (elisp) code I have found useful for programming in Agda in emacs. You can copy and paste this (with modifications as noted) into your .emacs file. You can open your .emacs file by doing “Control-x Control-f” in emacs, and then typing ~/.emacs.

```
; This defines backslash commands for some extra symbols.
(eval-after-load "quail/latin-ltx"
  '(mapc (lambda (pair)
            (quail-defrule (car pair) (cadr pair) "TeX"))
    '( ("\\bb" "B") ("\\bl" "L") ("\\bs" "S")
      ("\\bt" "T") ("\\bv" "V") ("\\cv" "Y")
      ("\\comp" "o") ("\\m" "→") ("\\om" "ω"))))

; This sets the Control-c Control-k shortcut to
; describe the character under your cursor.
(global-set-key "\C-c\C-k" 'describe-char)
```

[continued on the next page]

```

; Change Control-c Control-, and Control-c Control-.
; in Agda mode so they show the normalized rather
; than the "simplified" goals
(defun agda2-normalized-goal-and-context ()
  (interactive)
  (agda2-goal-and-context '(3)))
(defun agda2-normalized-goal-and-context-and-inferred ()
  (interactive)
  (agda2-goal-and-context-and-inferred '(3)))
(eval-after-load "agda2-mode"
  '(progn
    (define-key agda2-mode-map (kbd "C-c C-,")
      'agda2-normalized-goal-and-context)
    (define-key agda2-mode-map (kbd "C-c C-." )
      'agda2-normalized-goal-and-context-and-inferred)))

; This sets the Agda include path.
; Change YOUR-PATH to the path on your computer
; to the Iowa Agda Library directory. Use forward
; slashes to separate subdirectories.
(custom-set-variables
  '(agda2-include-dirs
    (quote (". " "YOUR-PATH/ial"))))
)

```

References

- [1] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, pages 135–150, 2011.
- [2] Lennart Augustsson. Cayenne - a Language with Dependent Types. In Matthias Felleisen, Paul Hudak, and Christian Queinnec, editors, *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 239–250. ACM, 1998.
- [3] Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement day. In J. Giesl and R. Hähnle, editors, *Automated Reasoning (IJCAR 2010)*, volume 6173 of *LNCS*, pages 107–121. Springer, 2010.
- [4] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941. Annals of Mathematics Studies, no. 6.
- [5] Catarina Coquand. A formalised proof of the soundness and completeness of a simply typed lambda-calculus with explicit substitutions. *Higher-Order and Symbolic Computation*, 15(1):57–90, 2002.
- [6] Peter Dybjer and Andrzej Filinski. Normalization and Partial Evaluation. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*, pages 137–192. Springer, 2002. available at <http://www.diku.dk/~andrzej/papers/NaPE.pdf>.
- [7] Timothy Griffin. A Formulae-as-Types Notion of Control. In Frances E. Allen, editor, *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 47–58. ACM Press, 1990.
- [8] D. Grune and C.J.H. Jacobs. *Parsing Techniques - A Practical Guide (Second Edition)*, 2007.
- [9] Roger Hindley. The Root-2 Proof as an Example of Non-constructivity. Available from the author’s web page, <http://www.users.waitrose.com/~hindley/>, 2014.

- [10] Joe Hurd. First-order proof tactics in higher-order logic theorem provers. In *Design and Application of Strategies/Tactics in Higher Order Logics*, number NASA/CP-2003-212448 in *NASA Technical Reports*, pages 56–68, 2003.
- [11] Danko Ilik. *Constructive Completeness Proofs and Delimited Control*. PhD thesis, Ecole Polytechnique X, 2010.
- [12] Oleg Kiselyov. Type-safe functional formatted IO. <http://okmij.org/ftp/typed-formatting/FPrintScan.html>. Accessed: July, 2015.
- [13] D. Kozen. *Automata and Computability*. Springer, 1997.
- [14] Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.
- [15] Albert R. Meyer and Mark B. Reinhold. "Type" is Not a Type. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 287–295. ACM, 1986.

Index

- absurd pattern, 39, 105
- anonymous function, 81
- automatic case split, 63
- binary relation, 181
- bool-relations.agda, 117
- bool-thms.agda, 25
- bool-thms2.agda, 15
- bool.agda, 11
- boolean type
 - ff, 11
 - tt, 11
- braun-tree.agda, 106
- bst.agda, 116
- closures.agda, 200
- combinators.agda, 190
- comments, 12
- completeness, 206, 221
- constructive logic, 28
- constructors, 13
- context, 58
- context-free grammars, 129
- Curry-Howard isomorphism, 26, 33, 57
- definitional equality, 28
- disjunction type, 107
- dot pattern, 43
- dotted variables, 70
- empty type, 187
- empty.agda, 187
- eq.agda, 28
- equality
 - definitional, 28
 - different kinds of, 71
 - propositional, 26
 - reflexivity of, 27
 - symmetry of, 42
- equality test, 71
- equality type, 26
- goal, 44, 55
- gratr, 129
- gratr parsing tool, 129
- head of a list, 76
- holes, 35, 58
- IAL, 8
- if_then_else_, 21
- implicit arguments, 34
- induction, 57
- inspect idiom, 91
- integer.agda, 147
- Iowa Agda Library, 8
- keep idiom, 91
- kripke-semantics.agda, 206
- lambda abstraction, 81
- let, 80
- level.agda, 12
- list-merge-sort.agda, 127
- list-simplifier.agda, 157
- list-thms.agda, 84
- list.agda, 77
- maybe.agda, 82
- minmax.agda, 122
- nat-nonzero.agda, 114
- nat-thms.agda, 53
- nat-to-string.agda, 141
- nat.agda, 50
- normalization by evaluation, 206

- polymorphism, 21
- positive propositional intuitionistic logic,
 - 206
- preorder, 119
- product.agda, 114

- record literals, 212
- records, 211
- reflection
 - proof by, 157
- reflexivity, 27, 72, 119, 210
- relations.agda, 117
- rewrite, 44, 85

- Sigma types, 114
- soundness, 72, 206, 216
- string-format.agda, 151
- sum.agda, 107
- symmetry, 42

- tail of a list, 76
- termination, 181
- termination.agda, 182
- top type, 148
- transitive closure, 200
- transitivity, 68, 119, 210
- trie data structure, 169
- type levels, 13
- type parameters, 76
- type refinement, 33

- unit.agda, 148
- universal quantifier, 32

- variable names, 20
- vector.agda, 100
- view, 154

- where clause, 113
- with construct, 87, 89