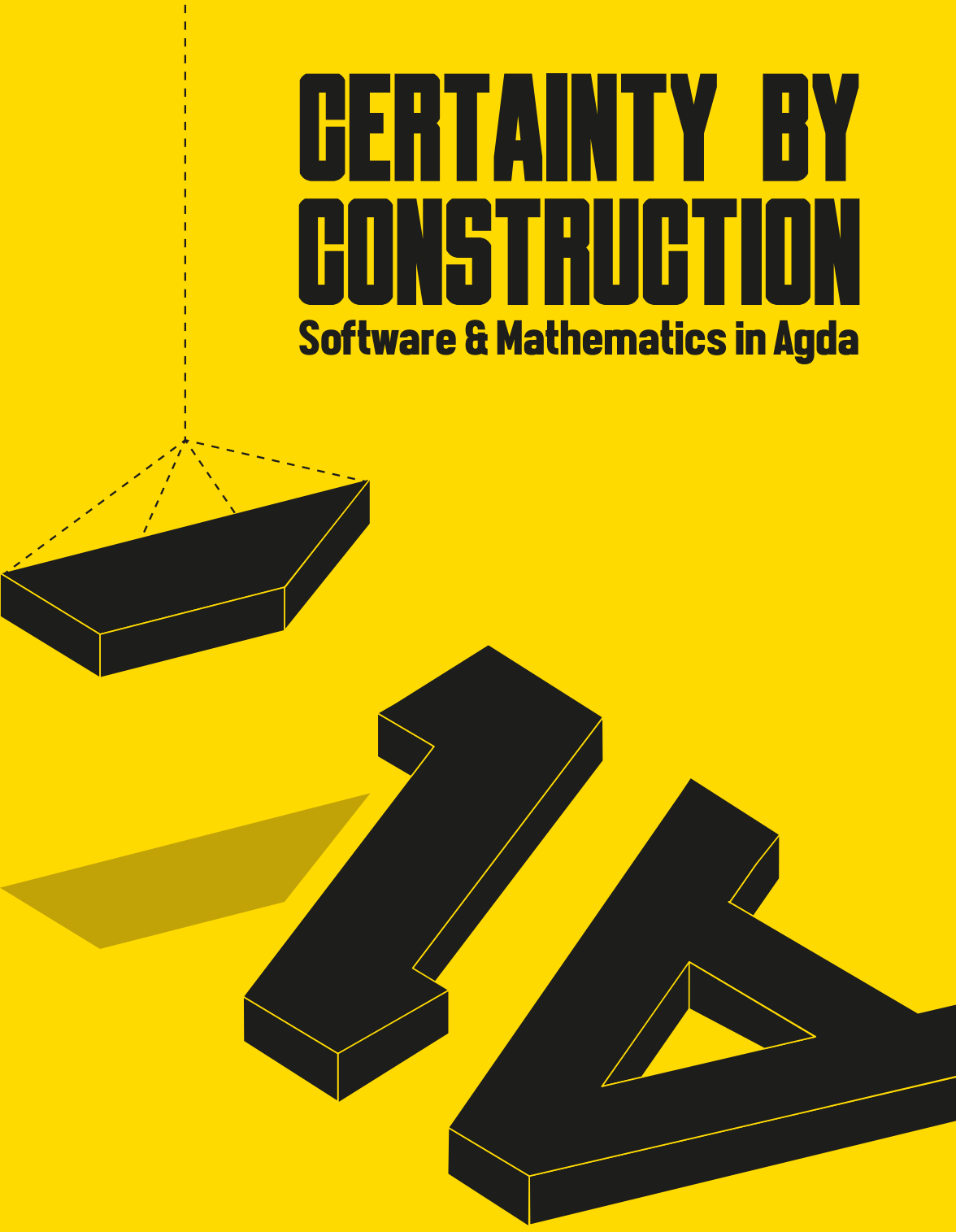
CERTAINTY BY CONSTRUCTION

Software & Mathematics in Agda



Sandy Maguire

Certainty by Construction

Software & Mathematics in Agda

Sandy Maguire

Cofree Press

First published 2023

Version 1.0.8 / 2024-01-09

To Erin Jackes,
who knows the true meaning of equality

In science if you know what you are doing you should not be doing it.

In engineering if you do not know what you are doing you should not be doing it.

RICHARD HAMMING

Contents

Preface

It was almost ten years ago when I stumbled across the idea that in order to truly understand something, one should write themselves a textbook to really flesh out the concepts. The reasoning, it goes, is that when you're just forced to articulate something from the ground up, the holes in your understanding become immediately obvious. As Richard Feynman says, "the first principle is that you must not fool yourself and you are the easiest person to fool."

The first textbook project I ever attempted in earnest was one on category theory, an alternative foundation for mathematics, as opposed its more traditional set theoretic foundation. Category theory has much to recommend it; while set theory is very good at get-the-answer-by-any-means-necessary sorts of approaches, category theory instead gives good theoretical underpinnings to otherwise-nebulous concepts like "abstraction" and "composition." The argument I'd heard somewhere was that doing math in sets was like writing programs in assembly code, while doing it in categories was comparative to writing them in a modern programming language.

While writing a textbook was indeed helpful at identifying holes in my understanding, it was never a particularly good tool for *building* that understanding in the first place. My mathematics education is rather spotty—I was good at "running the maze" in school, which is to say, I could reliably compute answers. At university I received an engineering degree, which required lots more running of the maze, but I did horrendously in all of my *actual* math courses. I had grown up writing software, and it felt extraordinarily vulnerable to need to write the technical solutions required by mathematics, without having tooling to help me. I was looking for some sort of compiler or runtime to help troubleshoot my proofs. As a self-taught programmer, I had developed a bad habit of brute-forcing my way to working programs. My algorithm for this was as follows:

1. write a program that seems to make sense
2. run it and pray that it works
3. insert some print statements to try to observe the failure
4. make random changes
5. go back to 2

Depending on the programming language involved, this can be an extremely tight feedback loop. But when it came to mathematics, the algorithm becomes much less effective. Not only is it meaningless to "run" a proof, but also as a non-mathematician thrust into the domain, I found it unclear what even constituted a proof. Which steps did I need to justify? How could I know when I was done? When is a proof sufficiently convincing?

At least in university, the answer to that last question is "when you get 100% on the assignment." In a very real sense, the feedback algorithm for mathematics is this:

1. write the proof
2. submit the assignment
3. wait for it to be marked
4. tweak the bits that have red underlines
5. go back to 2

Of course, this algorithm requires some sort of mythical teaching assistant with infinite time and understanding, who will continually mark your homework until you're satisfied with it. You might not be done in time to get the grade, but with perseverance, you'll eventually find enlightenment—intuiting the decision procedure that allows a theorem to pass through without any red underlines. I suppose this is how anyone learns anything, but the feedback cycle is excruciatingly slow.

Perhaps out of tenaciousness and math-envy more than anything else, I managed to keep at my category theory goal for seven years. I would pick up a new textbook every few years, push through it, get slightly further than the last time, and then bounce off anew. The process was extremely slow-going, but I did seem to be making sense of it. Things sped up immensely when I made friends with

kind, patient people who knew category theory, who would reliably answer my questions and help when I got stuck.

What a godsend! Having kind, patient friends sped up the feedback algorithm for mathematics by several orders of magnitude, from "wait until the next time I pick up a category theory textbook and identify some mistakes in my understanding elucidated by time" to "ask a friend and get a response back in less than an hour." Amazing stuff.

At some point, one of those kind, patient friends introduced me to proof assistants. Proof assistants are essentially programming languages meant for doing mathematics, and stumbling across them gave me a taste of what was possible. The selling point is that these languages come with a *standard library of mathematical theorems.* As a software guy, I know how to push programming knowledge into my brain. You bite off one module of the standard library at a time, learning what's there, inlining definitions, and reading code. If you ever need to check your understanding, you just code up something that seems to make sense and see if the compiler accepts it. Now, as they say, I was cooking with gas.

I spent about a year bouncing around between different proof assistants. There are several options in this space, each a descendant from a different family of programming languages. During that year, I came across Agda—a language firmly in the functional programming camp, with a type-system so powerful that several years later, I have still only scratched the surface of what's possible. Better yet, Agda comes with a massive standard library of mathematics. Once you can wrap your head around Agda programming (no small task), there is a delectable buffet of ideas waiting to be feasted upon.

Learning Agda has been slow going, and I chipped away at it for a year in the context of trying to prove things not about mathematics, but about my own programs. It's nice, for example, to prove your algorithm always gets the right answer, and to not need to rely on a bevy of tests that hopefully cover the input space (but how would you ever know if you'd missed a case?).

In the meantime, I had also been experimenting with formalizing my category theory notes in Agda. That is, I picked up a new textbook, and this time, coded up all of the definitions and theorems in Agda. Better, I wrote my answers to the book's exercises as *programs*, and Agda's compiler would yell at me if there was a flaw in my reasoning. WOW! What a difference this made! All of a sudden I had the feedback mechanism for mathematics that I'd always

wanted. I could get the red underlines on unconvincing (or wrong) parts of my argument automatically—all on the order of seconds!

Truly this is a marvelous technology.

When feedback becomes instant, tedious chores turn into games. I found myself learning mathematics into the late evenings on weekends. I found myself redoing theorems I'd already proved, trying to find ways of making them prettier, or more elegant, or more concise, or what have you. I've made more progress in category theory in the last six months than I had in a decade. I feel now that proof assistants are the best video game ever invented.

The idea to write this book came a few months later, when some friends of mine wanted to generalize some well-established applied mathematics and see what happened. I didn't know anything about the domain, but thought it might be fun to tag along. Despite not knowing anything, my experience with proving things in Agda meant I was able to help more than any of us expected.

I came away with the insight that there are a lot more people out there like me: people who want to be better at mathematics but don't quite know how to get there. People who are technically minded and have keen domain knowledge, but are lacking the *proof* side of things. And after all, what is mathematics without proof?

So we come now to this book. This book is the textbook I ended up writing—not to teach myself category theory as originally intended, but instead to teach myself mathematics at large. In the process of explaining the techniques, and the necessity of linearizing the narrative, I've been forced to grapple with my understanding of math, and it has become very clear in the places I was fooling myself. This book is itself a series of Agda modules, meaning it is a fully type-checked library. That is, it comes with the guarantee that I have told no lies; at least, none mathematically. I am not an expert in this field, but have stumbled across a fantastic method of learning and teaching mathematics.

This textbook is the book I wish I had found ten years ago. It would have saved me a great deal of wasted effort and false starts. I hope it can save you from the same. Good luck, godspeed, and welcome aboard.

The Co-blub Paradox

It is widely acknowledged that the languages you speak shape the thoughts you can think; while this is true for natural language, it is doubly so in the case of programming languages. And it's not hard to see why; while humans have dedicated neural circuitry for natural language, it would be absurd to suggest that we also have dedicated neural circuitry for fiddling with arcane, abstract symbols, usually encoded arbitrarily as electrical potentials on a conductive metal.

Programming—and mathematics more generally—does not come easily to us humans, and for this reason it can be hard to see the forest for the trees. We have no built-in intuition as to what should be possible, and thus, this intuition is built only by observing the work of more-established practitioners. In the more artificial human endeavors like programming, newcomers to the field must be constructivists—their art is shaped only by the patterns they have previously observed. Because different programming languages support different features and idioms, the imaginable shape of what programming *is* must be shaped by those languages we understand deeply.

In a famous essay, "Beating the Averages," Graham (2001) points out the so-called *Blub paradox.* This, Graham says, is the ordering of programming languages by powerfulness; a programmer who thinks in a middle-of-the-road language along this ordering (call it Blub) can identify less powerful languages, but not those which are more powerful. The idea rings true; one can arrange languages in power by the features they support, and subsequently check to see if a language supports all the features we feel to be important. If it doesn't, it must be less powerful. However, this technique doesn't work to identify more powerful languages—at best, you will see that the compared language supports all the features you're looking for, but you don't know enough to ask for more.

Quasi-formally, we can describe the Blub paradox as a semi-

decision procedure. That is, given an ordering over programming languages (here, by their relative "power",) we can determine whether a language is less than our comparison language, but not whether it is more than. We can determine when the answer is definitely "yes," but, not when it is "no!"

Over two decades of climbing this lattice of powerful languages, I have come to understand a lesser-known corollary of the Blub paradox, coining it the *Co-Blub paradox.* This is: knowledge of lesser languages is *actively harmful* when transposed into the context of a more powerful language. The hoops you unwittingly jumped through in Blub due to lacking feature X are *anti-patterns* in the presence of feature X. This is obviously true when stated abstractly, but insidiously hard to see when we are the ones writing the anti-patterns.

Let's look at a few examples over the ages, to help motivate the problem before we get into our introspection proper. In the beginning, people programmed directly in machine code. Not assembly, mind you, but in raw binary-encoded op-codes. They had a book somewhere showing them what bits needed to be set in order to cajole the machine into performing any given instruction. Presumably if this were your job, you'd eventually memorize the bit patterns for common operations, and it wouldn't be nearly as tedious as it seems today.

Then came assembly languages, which provided human-meaningful mnemonics to the computer's opcodes. No longer did we need to encode a jump as the number 1018892 (`11111000110000001100` in binary)—now it was simply `jl 16`. Still mysterious to be sure, but you must admit such a thing is infinitely more legible. When encoded directly in machine code, programs were, for the most part, write-only. But assembly languages don't come for free; first you need to write an assembler: a program that reads the mnemonics and outputs the raw machine code. If you were already proficient writing machine code directly, you can imagine the task of implementing an assembler to feel like make work—a tool to automate a problem you don't have. In the context of the Co-Blub paradox, knowing the direct encodings of your opcodes is an anti-pattern when you have an assembly language, as it makes your contributions inscrutable to your peers.

Programming directly in assembly eventually hit its limits. Every computer had a different assembly language, which meant if you wanted to run the same program on a different computer you'd have to completely rewrite the whole thing; often needing to translate

between extremely different concepts and limitations. Ignoring a lot of history, C came around with the big innovation that software should be portable between different computers: the same C program should work regardless of the underlying machine architecture—more or less. If you were an assembly programmer, you ran into the anti-pattern that while you could squeeze more performance and perform clever optimizations if you were aware of the underlying architecture, this fundamentally limited you *to that platform.*

By virtue of being in many ways a unifying assembly language, C runs very close to what we think of as "the metal." Although different computer architectures have minor differences in registers and ways of doing things, they are all extremely similar variations on a theme. They all expose storable memory indexed by a number, operations for performing basic logic and arithmetic tasks, and means of jumping around to what the computer should consider to be the next instruction. As a result, C exposes this abstraction of what a computer *is* to its programmers, who are thus required to think about mutable memory and about how to encode complicated objects as sequences of bytes in that memory.

But then (skipping much history) came along Java, whose contribution to mainstream programming was to popularize the idea that memory is cheap and abundant. Thus, Java teaches its adherents that it's OK to waste some bytes in order to alleviate the headache of needing to wrangle it all on your own. A C programmer coming to Java must unlearn the idea that memory is sacred and scarce, that one can do a better job of keeping track of it than the compiler can. The hardest thing to unlearn is that memory is an important thing to think about in the first place.

There is a clear line of progression here; as we move up the lattice of powerful languages, we notice that more and more details of what we thought were integral parts of programming turn out to be not particularly relevant to the actual task at hand. However, the examples thus discussed are already known to the modern programmer. Let's take a few steps further, into languages deemed esoteric in the present day. It's easy to see and internalize examples from the past, but those staring us in the face are much more difficult to spot.

Compare Java then to Lisp, which—among many things—makes the argument that functions, and even *programs themselves,* are just as meaningful objects as are numbers and records. Where Java requires the executable pieces to be packaged up and moved around with explicit dependencies on the data it requires, Lisp just lets you

write and pass around functions, which automatically carry around all the data they reference. Java has a *design pattern* for this called the "command pattern," which has required much ado and ink to be spilled. In Lisp, however, you can just pass functions around as first-class values and everything works properly. It can be hard to grok why exactly if you are used to thinking about computer programs as static sequences of instructions. Indeed, the command pattern is bloated and ultimately unnecessary in Lisp, and practitioners must first unlearn it before they can begin to see the way of Lisp.

Haskell takes a step further than Lisp, in that it restricts when and where side-effects are allowed to occur in a program. This sounds like heresy (and feels like it for the first six months of programming in Haskell) until you come to appreciate that *almost none* of a program needs to perform side-effects. As it happens, side-effects are the only salient observation of the computer's execution model, and by restricting their use, Haskell frees its programmers from needing to think about how the computer will execute their code—promising only that it will. As a result, Haskell code looks much more like mathematics than it looks like a traditional computer program. Furthermore, by abstracting away the execution model, the runtime is free to parallelize and reorder code, often even eliding unnecessary execution altogether. The programmer who refuses to acknowledge this reality and insists on coding with side-effects pays a great price, both on the amount of code they need to write, in its long-term reusability, and, most importantly, in the correctness of their computations.

All of this brings us to Agda, which is as far as I've personally come along the power lattice of programming languages. Agda's powerful type system allows us to articulate many invariants that are impossible to write down in other languages. In fact, its type system is so precise we can *prove* that our solutions are correct, which alleviates the need to actually *run* the subsequent programs. In essence, programming in Agda abstracts away the notion of execution entirely. Following our argument about co-Blub programmers, they will come to Agda with the anti-pattern that thinking their hard-earned, battle-proven programming techniques for wrangling runtime performance will come in handy. But this is not the case; most of the techniques we have learned and consider "computer science" are in fact *implementation ideas:* that is, specific realizations from infinite classes of solutions, chosen not for their simplicity or clarity, but for their *efficiency.*

Thus, the process of learning Agda, in many ways, is learning to separate the beautiful aspects of problem solving from the multitude of clever hacks we have accumulated over the years. Much like the fish who is unable to recognize the ubiquitous water around him, as classically-trained programmers, it is nigh-impossible to differentiate the salient points from the implementation details until we find ourselves in a domain where they do not overlap. Indeed, in Agda, you will often feel the pain of having accidentally conflated the two, when your proofs end up being much more difficult than you feel they deserve. Despite the pain and the frustration, this is in fact a feature, and not a bug. It is a necessary struggle, akin to the type-checker informing you that your program is wrong. While it can be tempting to blame the tool, the real fault is in the craftsmanship.

A World Without Execution?

It's worth stopping and asking ourselves in what way a non-executable programming language might be useful. If the end result of a coding endeavor is the eventual result, whether that be an answer (as in a computation) or series of side-effects (as in most real-world programs,) non-execution seems useless at best and masturbatory at worst.

Consider the case of rewriting a program from scratch. Even though we reuse none of the original source code, nor the compiled artifacts, the second time through writing a program is always much easier. Why should this be so? Writing the program has an effect on the world, completely apart from the digital artifacts that result—namely, the way in which your brain changes from having written the program. Programming is a creative endeavor, and every program leaves its mark on its creator. It is through these mental battle scars—accumulated from struggling with and conquering problems—that we become better programmers. Considering a program to be a digital artifact only ignores the very real groove it made on its author's brain.

It is for this reason that programmers, in their spare time, will also write other sorts of programs. Code that are is not necessarily useful, but code that allows its author to grapple with problems. Many open-source projects got started as a hobbyist project that someone created in order to learn more about the internals of databases, or to try their hand at implementing a new programming language. For programmers, code is a very accessible means of exploring new ideas,

which acts as a forcing function to prevent us from fooling ourselves into thinking we understand when we don't. After all, it's much easier to fool ourselves than it is to fool the computer.

So, programmers are familiar with writing programs, not for running the eventual code, but for the process of having built it in the first place. In effect, the real output of this process is the neural pathways it engenders.

Agda fits very naturally into this niche; the purpose of writing Agda is not to be able to run the code, but because Agda is so strict with what it allows. Having programmed it in Agda will teach you much more about the subject than you thought there was to know. Agda forces you to grapple with the hard, conceptual parts of the problem, without worrying very much about how you're going to make the whole thing go fast later. After all, there's nothing *to* go fast if you don't know what you're building in the first place.

Consider the universal programmer experience of spending a week implementing a tricky algorithm or data structure, only to realize upon completion that you don't need it—either that it doesn't do what you hoped, or it doesn't actually solve the problem you thought you had. Unfortunately, this is the rule in software, not the exception. Without total conceptual clarity, our software can never possibly be correct, if for no other reason than we don't know what correct *means.* Maybe the program will still give you an answer, but it is nothing but willful ignorance to assume this output corresponds at all to reality.

The reality is, conceptual understanding is the difficult part of programming—the rest is just coloring by numbers. The way we have all learned how to do programming is to attempt to solve both problems at once: we write code and try to decipher the problem as we go; it is the rare coder who stops to think on the whiteboard, and the rarer-still engineer who starts there.

But again recall the case of rewriting a program. Once you have worked through the difficult pieces, the rest is just going through the motions. There exists some order in which we must wiggle our fingers to produce the necessary syntax for the computer to solve our problem, and finding this order is trivial once we have conceptual clarity of what the solution is.

This, in short, is the value of learning and writing Agda. It's less of a programming language as it is a tool for thought; one in which we can express extremely precise ideas, propagate constraints around, and be informed loudly whenever we fail to live up to the

exacting rigor that Agda demands of us. While traditional programming models are precise only about the "how," Agda allows us to instead think about the "what," without worrying very much about the "how." After all, we're all very good at the "how"—it's been drilled into us for our entire careers.

CHAPTER 2

An Exploration of Numbers

In this chapter, we will get our hands dirty, implementing several different number systems in Agda. The goal is threefold: to get some experience thinking about how to model problems in Agda, to practice seeing familiar objects with fresh eyes, and to get familiar with many of the mathematical objects we'll need for the remainder of the book.

Before we start, note that this chapter has prerequisite knowledge from sec. 1. And, as always, every new chapter must start a new module:

```
0 module Chapter2-Numbers where
```

Prerequisites

```
0     import Chapter1-Agda
```

As you might expect, Agda already has support for numbers, and thus everything we do here is purely to enhance our understanding. That being said, it's important to get an intuition for how we can use Agda to solve problems. Numbers are simultaneously a domain you already understand, and, in most programming languages, they usually come as pre-built, magical primitives.

This is not true in Agda: numbers are *defined* in library code. Our approach will be to build the same number system exported by the standard library so we can peek at how it's done. Again, this is just an exercise; after this chapter, we will just use the standard library's implementation, since it will be more complete, and allow us better interopability when doing real work.

2.1 Natural Numbers

It is one thing to say we will "construct the numbers," but doing so is much more involved. The first question to ask is *which numbers?* As it happens, we will build all of them.

But that is just passing the buck. What do we mean by "all" the numbers? There are many different sets of numbers. For example, there are the numbers we use to count in the real world (which start at 1.) There are also the numbers we use to index in computer science (which begin at 0.) There are the *integers*, which contain negatives. And then there are the *rationals* which contain fractions, and happen to be all the numbers you have ever encountered in real life.

But somehow, not even that is all the numbers. Beyond the rationals are the *reals*, which are somehow bigger than all the numbers you have actually experienced, and in fact are so big that the crushing majority of them are completely inaccessible to us.

The party doesn't stop there. After the reals come the *complex numbers* which have an "imaginary" part—whatever that means. Beyond those are the *quaternions*, which come with three different varieties of imaginary parts, and beyond those, the *octonions* (which have *seven* different imaginaries!)

In order to construct "the numbers," we must choose between these (and many other) distinct sets. And those are just some of the number systems mathematicians talk about. But worse, there are the number systems that computer scientists use, like the *bits*, the *bytes*, the *words*, and by far the worst of all, the IEEE 754 "floating point" numbers known as *floats* and *doubles*.

You, gentle reader, are probably a programmer, and it is probably in number systems such as these that you feel more at home. We will not, however, be working with number systems of the computer science variety, as these are extremely non-standard systems of numbers, with all sorts of technical difficulties that you have likely been burned so badly by that you have lost your pain receptors.

Who among us hasn't been bitten by an integer overflow, where adding two positive numbers somehow results in a negative one? Or by the fact that, when working with floats, we get different answers depending on which two of three numbers we multiply together first.

One might make a successful argument that these are necessary limitations of our computing hardware. As a retort, I will only point to the co-Blub paradox (sec.), and remind you that our goal here

is to learn *how things can be,* rather than limit our minds to the way we perceive things must be. After all, we cannot hope to reach paradise if we cannot imagine it.

And so we return to our original question of which number system we'd like to build. As a natural starting point, we will pick the simplest system that it seems fair to call "numbers": the *naturals.* These are the numbers you learn as a child, in a simpler time, before you needed to worry about things like negative numbers, decimal points, or fractions.

The natural numbers start at 0, and proceed upwards exactly one at a time, to 1, then 2, then 3, and so on and so forth. Importantly to the computer scientist, there are *infinitely many* natural numbers, and we intend to somehow construct *every single one of them.* We will not placate ourselves with arbitrary upper limits, or with arguments of the form "X ought to be enough for anyone."

How can we hope to generate an infinite set of numbers? The trick isn't very impressive—in fact, I've already pointed it out. You start at zero, and then you go up one at a time, forever. In Agda, we can encode this by saying `zero` is a natural number, and that, given some number `n`, we can construct the next number up—its *successor*—via `suc n`. Such an encoding gives rise to a rather elegant (if *inefficient*, but, remember, we don't care) specification of the natural numbers. Under such a scheme, we would write the number 3 as `suc (suc (suc zero))`.

It is important to stress that this is a *unary* encoding, rather than the traditional *binary* encoding familiar to computer scientists. There is nothing intrinsically special about binary; it just happens to be an easy thing to build machines that can distinguish between two states: whether they be magnetic forces, electric potentials, or the presence or absence of a bead on the wire of an abacus. Do not be distraught; working in unary *dramatically* simplifies math, and if you are not yet sold on the approach, you will be before the end of this chapter.

But enough talk. It's time to conjure up the natural numbers. In the mathematical literature, the naturals are denoted by the *blackboard bold* symbol `ℕ`—a convention we too will adopt. You can input this symbol via `\bN`.

```
0 module Definition-Naturals where
    data ℕ : Set where
      zero : ℕ
      suc  : ℕ → ℕ    ❶
```

Here we use the `data` keyword to construct a type consisting of several different constructors. In this case, a natural is either a `zero` or it is a `suc` of some other natural number. You will notice that we must give explicit types to constructors of a `data` type, and at ❶ we give the type of `suc` as `ℕ → ℕ`. This is the precise meaning that a `suc` is "of some other natural number." You can think of `suc` as the mathematical function:

$$x \mapsto x + 1$$

although this is just a mental shortcut, since we do not yet have formal definitions for addition or the number 1.

2.2 Brief Notes on Data and Record Types

Before we play around with our new numeric toys, I'd like to take a moment to discuss some of the subtler points around modeling data in Agda.

The `data` keyword also came up when we defined the booleans in sec. 1.6, as well as for other toy examples in sec. 1.

Indeed, `data` will arise whenever we'd like to build a type whose values are *apart*—that is, new symbols whose purpose is in their distinctiveness from one another. The boolean values `false` and `true` are just arbitrary symbols, which we assign meaning to only by convention. This meaning is justified exactly because `false` and `true` are *distinct symbols*.

This distinctness is also of the utmost importance when it comes to numbers. The numbers are interesting to us only because we can differentiate one from two, and two from three. Numbers are a collection of symbols, all distinct from one another, and it is from their apartness that we derive importance.

As a counterexample, imagine a number system in which there is only one number. Not very useful, is it?

Contrast this apartness to the tuple type, which you'll recall was a type defined via `record` instead of `data`. In some sense, tuples exist only for bookkeeping. The tuple type doesn't build new things, it just lets you simultaneously move around two things that already exist. Another way to think about this is that records are made up of things that already exist, while `data` types create new things *ex nihilo*.

Most programming languages have a concept of `record` types (whether they be called *structures*, *tuples*, or *classes*), but very few

support `data` types. Booleans and numbers are the canonical examples of `data` types, and the lack of support for them is exactly why these two types are usually baked into to a language.

It can be tempting to think of types defined by `data` as enums, but this is a subtly misleading. While enums are indeed apart from one another, this comes from the fact that enums are just special names given to particular values of ints. This is an amazingly restricting limitation.

Note that in Agda, `data` types are strictly more powerful than enums, because they don't come with this implicit conversion to ints. As a quick demonstration, note that `suc` is apart from `zero`, but `suc` can accept any `ℕ` as an *argument!* While there are only 2^{64} ints, there are *infinitely many* `ℕ`s, and thus types defined by `data` in Agda must be more powerful than those defined as enums in other languages.

More generally, constructors in `data` types can take *arbitrary* arguments, and we will often use this capability moving forwards.

2.3 Playing with Naturals

Let's return now to our discussion of the naturals. Since we'd like to reuse the things we build in future chapters, let's first import the natural numbers from the standard library.

```
⇤─0 module Sandbox-Naturals where
      open import Data.Nat
        using (ℕ; zero; suc)
```

By repeated application of `suc`, we can build an infinite tower of natural numbers, the first four of which are built like this:

```
⇤─2 one : ℕ
    one = suc zero

    two : ℕ
    two = suc one

    three : ℕ
    three = suc two

    four : ℕ
    four = suc three
```

Of course, these names are just for syntactic convenience; we could have instead defined `four` thusly:

```
four : ℕ
four = suc (suc (suc (suc zero)))
```

It is tempting to use the traditional base-ten symbols for numbers, and of course, Agda supports this (although setting it up will require a little more effort on our part.) However, we will persevere with our explicit unary encoding for the time being, to really hammer-in that there is no magic happening behind the scenes here.

The simplest function we can write over the naturals is to determine whether or not the argument is equal to 0. For the sake of simplicity, this function will return a boolean, but note that this is a bad habit in Agda. There are much better techniques that don't lead to *boolean blindness* that we will explore in sec. 6. This function therefore is only provided to help us get a feel for pattern matching over natural numbers.

We can get access to the booleans by importing them from our exports from sec. 1:

```
open Chapter1-Agda
  using (Bool; true; false)
```

The function we'd like to write determines if a given `ℕ` is equal to `zero`, so we can begin with a name, a type signature, and a hole:

```
n=0? : ℕ → Bool
n=0? = ?
```

After `MakeCase` (`C-c C-c` in Emacs and VS Code), our argument is bound for us:

```
n=0? : ℕ → Bool
n=0? n = {! !}
```

and, like when writing functions over the booleans, we can immediately `MakeCase` with argument `n` (`C-c C-c` in Emacs and VS Code) to split `n` apart into its distinct possible constructors:

```
n=0? : ℕ → Bool
n=0? zero    = {! !}
```

```
n=0? (suc x) = {! !}    ❶
```

Interestingly, at ❶, Agda has given us a new form, something we didn't see when considering the booleans. We now have a pattern match of the form `suc x`, which after some mental type-checking, makes sense. We said `n` was a `ℕ`, but `suc` has type `ℕ → ℕ`. That means, `n` can only be a natural number of the `suc` form *if that function has already been applied to some other number.* And `x` is that other number.

The interpretation you should give to this expression is that if n is of the form `suc x`, then $x = n - 1$. Note that `zero` is not of the form `suc x`, and thus we don't accidentally construct any negative numbers under this interpretation.

Returning to `n=0?`, we care only if our original argument `n` is `zero`, which we can immediately solve from here—without needing to do anything with `x`:

```
n=0? : ℕ → Bool
n=0? zero    = true
n=0? (suc x) = false
```

It will be informative to compare this against a function that computes whether a given natural is equal to 2.

Exercise (Easy) Implement `n=2? : ℕ → Bool`

Solution

```
n=2? : ℕ → Bool
n=2? zero                = false
n=2? (suc zero)          = false
n=2? (suc (suc zero))    = true
n=2? (suc (suc (suc x))) = false
```

or, alternatively:

```
n=2? : ℕ → Bool
n=2? (suc (suc zero)) = true
n=2? _                = false
```

2.4 Induction

Unlike functions out of the booleans, where we only had two possibilities to worry about, functions out of the naturals have (in principle) an infinite number of possibilities. But our program can only ever be a finite number of lines, which leads to a discrepancy. How can we possibly reconcile an infinite number of possibilities with a finite number of cases?

Our only option is to give a finite number of interesting cases, and then a default case which describes what to do for everything else. Of course, nothing states that this default case must be constant, or even simple. That is to say, we are not required to give the *same* answer for every number above a certain threshold.

To illustrate this, we can write a function which determines whether its argument is even. We need only make the (mathematical) argument that "a number $n+2$ is even if and only if the number n is." This is expressed naturally by recursion:

```
even? : ℕ → Bool
even? zero          = true
even? (suc zero)    = false
even? (suc (suc x)) = even? x
```

Here, we've said that `zero` is even, `one` is not, and for every other number, you should subtract two (indicated by having removed two `suc` constructors from `x`) and then try again.

This general technique—giving some explicit answers for specific inputs, and recursing after refining—is known as *induction.*

It is impossible to overstate how important induction is. Induction is the fundamental mathematical technique. It is the primary workhorse of all mathematics.

Which makes sense; if you need to make some argument about an infinite number of things, you can neither physically nor theoretically analyze each and every possible case. You instead must give a few (usually very simple) answers, and otherwise show how to reduce a complicated problem into a simpler one. This moves the burden of effort from the theorem prover (you) to whomever wants an answer, since they are the ones who become responsible for carrying out the repetitive task of reduction to simpler forms. However, this is not so bad, since the end-user is the one who wants the answer, and they necessarily have a particular, finite problem that they'd like to solve.

Not being very creative, mathematicians often *define* the principle of induction as being a property of the natural numbers. They say all of mathematics comes from:

1. *a base case*—that is, proving something in the case that $n = 0$
2. *an inductive case*—that is, showing something holds in the case of n under the assumption that it holds under $n - 1$.

However, the exact same technique can be used for any sort of recursively-defined type, such as lists, trees, graphs, matrices, etc. While perhaps you could shoe-horn these to fit into the natural numbers, it would be wasted effort in order to satisfy nothing but a silly definition. That notwithstanding, the terminology itself is good, and so we will sometimes refer to recursive steps as "induction" and non-recursive steps as "base cases."

2.5 Two Notions of Evenness

We have now defined `even?` a function which determines whether a given natural number is even. A related question is whether we can define a type for *only* the even numbers. That is, we'd like a type which contains 0, 2, 4, and so on, but neither 1, nor 3, nor *any* of the odd numbers.

In a monkey-see-monkey-do fashion, we could try to define a new type called `Evenℕ` with a constructor for `zero`, but unlike `ℕ`, no `suc`. Instead, we will give a constructor called `suc-suc`, intending to be suggestive of taking two successors simultaneously:

```
data Evenℕ : Set where
  zero    : Evenℕ
  suc-suc : Evenℕ → Evenℕ
```

We can transform an `Evenℕ` into a `ℕ` by induction:

```
toℕ : Evenℕ → ℕ
toℕ zero        = zero
toℕ (suc-suc x) = suc (toℕ x)
```

This approach, however, feels slightly underwhelming. The reflective reader will recall that in a `data` type, the *meaning* of the constructors comes only from their types and the suggestive names we give them.

A slight renaming of `suc-suc` to `suc` makes the definition of `EvenℕN` look very similar indeed to that of `ℕ`. In fact, the two types are completely equivalent, modulo the names we picked.

As such, there is nothing stopping us from writing an incorrect (but not *obviously* wrong) version of the `toℕ` function. On that note, did you notice that the definition given above *was* wrong? Oops! Instead, the correct implementation should be this:

```
toℕ : EvenℕN → ℕ
toℕ zero        = zero
toℕ (suc-suc x) = suc (suc (toℕ x))
```

You might want to double check this new definition, just to make sure I haven't pulled another fast one on you. Double checking, however, is tedious and error prone, and in an ideal world, we'd prefer to find a way to get the computer to double check on our behalf.

Rather than trying to construct a completely new type for the even naturals, perhaps we can instead look for a way to filter for only the naturals we want. A mathematician would look at this problem and immediately think to build a *subset*—that is, a restricted collection of the objects at study. In this particular case, we'd like to build a subset of the natural numbers which contains only those that are even.

The high-level construction here is we'd like to build `IsEven : ℕ → Set`, which, like you'd think, is a function that takes a natural and returns a type. The idea that we can *compute* types in this way is rare in programming languages, but is very natural in Agda.

In order to use `IsEven` as a subset, it must return some sort of "usable" type when its argument is even, and an "unusable" type otherwise. We can take this function idea literally if we'd please, and postulate two appropriate types:

```
module Sandbox-Usable where
  postulate
    Usable   : Set
    Unusable : Set

  IsEven : ℕ → Set
  IsEven zero          = Usable
  IsEven (suc zero)    = Unusable
  IsEven (suc (suc x)) = IsEven x
```

You will notice the definition of `IsEven` is identical to that of `even?` except that we replaced `Bool` with `Set`, `true` with `Usable`, and `false` with `Unusable`. This is what you should expect, as `even?` was already a function that computed whether a given number is even!

While we could flesh this idea out in full by finding specific (non-postulated) types to use for `Usable` and `Unusable`, constructing subsets in this way isn't often fruitful. Though it occasionally comes in handy, and it's nice to know you can compute types directly in this way.

Let's drop out of the `Sandbox-Usable` module, and try defining `IsEven` in a different way.

The situation here is analogous to our first venture into typing judgments in sec. 1.6. While it's possible to do all of our work directly with postulated judgments, Agda doesn't give us any help in doing so. Instead, things became much easier when we used a more principled structure—namely, using the `data` type. Amazingly, here too we can use a `data` type to solve our problem. The trick is to add an *index* to our type, which you can think of as a "return value" that comes from our choice of constructor.

Don't worry, the idea will become much clearer in a moment after we look at an example. Let's begin just with the `data` declaration:

```
⇤ 2 data IsEven : ℕ → Set where    ❶
```

Every type we have seen so far has been of the form `data X : Set`, but at ❶ we have `ℕ → Set` on the right side of the colon. Reading this as a type declaration directly, it says that this type `IsEven` we're currently defining *is exactly* the function we were looking for earlier—the one with type `ℕ → Set`. Because of this parameter, we say that `IsEven` is an *indexed type*, and that the `ℕ` in question is its index.

Every constructor of an indexed type must fill-in each index. To a first approximation, constructors of an indexed type are *assertions* about the index. For example, it is an axiom that `zero` is an even number, which we can reflect directly as a constructor:

```
⇥ 4 zero-even : IsEven zero
```

Notice that this constructor is equivalent to the base case `even? zero = true`. We would like to exclude odd numbers from `IsEven`, so we can ignore the `suc zero` case for the moment. In the inductive case, we'd like to say that if n is even, then so too is $n + 2$:

```
4 suc-suc-even : {n : ℕ} → IsEven n → IsEven (suc (suc n))
```

In a very real sense, our indexed type `IsEven` is the "opposite" of our original decision function `even?`. Where before we removed two calls to `suc` before recursing, we now recurse first, and then *add* two calls to `suc`. This is not a coincidence, but is in fact a deep principle of mathematics that we will discuss later.

The concept of indexed types is so foreign to mainstream programming that it is prudent to spend some time here and work through several examples of what-the-hell-is-happening. Let's begin by showing that `four` is even. Begin with the type and a hole:

```
⇤2 four-is-even : IsEven four
   four-is-even = ?
```

Here's where things get cool. We can ask Agda to refine this hole via `Refine` (`C-c C-r` in Emacs and VS Code). Recall that refine asks Agda to fill in the hole with the only constructor that matches. Rather amazingly, the result of this invocation is:

```
2 four-is-even : IsEven four
  four-is-even = suc-suc-even {! !}
```

Even more impressive is that the new goal has type `IsEven two`—which is to say, we need to show that `two` is even in order to show that `four` is even. Thankfully we can ask Agda to do the heavy lifting for us, and again request a `Refine` (`C-c C-r` in Emacs and VS Code):

```
2 four-is-even : IsEven four
  four-is-even = suc-suc-even (suc-suc-even {! !})
```

Our new hole has type `IsEven zero`, which again Agda can refine for us:

```
2 four-is-even : IsEven four
  four-is-even = suc-suc-even (suc-suc-even zero-even)
```

With all the holes filled, we have now successfully proven that `four` is in fact even. But can we trust that this works as intended? Let's see what happens when we go down a less-happy path. Can we also prove `IsEven three`?

```
three-is-even : IsEven three
three-is-even = ?
```

Let's play the same refinement game. Invoking `Refine` (`C-c C-r` in Emacs and VS Code) results in:

```
three-is-even : IsEven three
three-is-even = suc-suc-even {! !}
```

Our new goal is `IsEven one`. But if we try to refine again, Agda gives us an error:

INFO WINDOW

```
No introduction forms found.
```

What's (correctly) going wrong here is that Agda is trying to find a constructor for `IsEven (suc zero)`, but no such thing exists. We have `zero-even` for `IsEven zero`, and we have `suc-suc-even` for `IsEven (suc (suc n))`. But there is no such constructor when we have only one `suc`! Thus neither `zero-even` nor `suc-suc-even` will typecheck in our hole. Since these are the *only* constructors, and neither fits, it's fair to say that *nothing can possibly fill this hole.* There is simply no way to give an implementation for `three-is-even`—it's impossible to construct an `IsEven n` whenever `n` is odd.

This is truly a miraculous result, and might give you a glimpse at why we do mathematics in Agda. The idea is to carefully construct types whose values are possible only when our desired property is *actually true.* We will explore this topic more deeply in sec. 3.

Exercise (Easy) Build an indexed type for `IsOdd`.

Solution

```
data IsOdd : ℕ → Set where
  one-odd     : IsOdd one
  suc-suc-odd : {n : ℕ} → IsOdd n → IsOdd (suc (suc n))
```

or, alternatively,

```
data IsOdd' : ℕ → Set where
  is-odd : {n : ℕ} → IsEven n → IsOdd' (suc n)
```

Exercise (Easy) Write an inductive function which witnesses the fact that every even number is followed by an odd number. This function should have type `{n : ℕ} → IsEven n → IsOdd (suc n)`.

Solution

```
evenOdd : {n : ℕ} → IsEven n → IsOdd (suc n)
evenOdd zero-even        = one-odd
evenOdd (suc-suc-even x) = suc-suc-odd (evenOdd x)
```

or, if you took the alternative approach in the previous exercise,

```
evenOdd' : {n : ℕ} → IsEven n → IsOdd' (suc n)
evenOdd' = is-odd
```

2.6 Constructing Evidence

When we originally implemented `even?`, I mentioned that functions which return booleans are generally a bad habit in Agda. You've done a lot of computation in order to get the answer, and then throw away all of that work just to say merely "yes" or "no." Instead of returning a `Bool`, we could instead have `even?` return an `IsEven`, proving the number really is even!

However, not *all* numbers are even, so we will first need some notion of failure. This is an excellent use for the `Maybe` type, which is a container that contains exactly zero or one element of some type `A`. We can define it as:

```
data Maybe (A : Set) : Set where
  just    : A → Maybe A
  nothing :     Maybe A
```

Here, `just` is the constructor for when the `Maybe` *does* contain an element, and `nothing` is for when it doesn't. `Maybe` is a good type for representing *partial functions*—those which don't always give back a result. Our desired improvement to `even?` is one such function, since there are naturals in the input which do not have a corresponding value in the output.

Our new function is called `evenEv`, to be suggestive of the fact that it returns *evidence* of the number's evenness. The first thing to study here is the type:

```
⇤ 2 evenEv : (n : ℕ) → Maybe (IsEven n)
    evenEv = ?
```

The type signature of `evenEv` says "for some `n : ℕ`, I can maybe provide a proof that it is an even number." The implementation will look very reminiscent of `even?`. First, we can do `MakeCase` (`C-c C-c` in Emacs and VS Code) a few times:

```
2 evenEv : (n : ℕ) → Maybe (IsEven n)
  evenEv zero          = {! !}
  evenEv (suc zero)    = {! !}
  evenEv (suc (suc n)) = {! !}
```

Then, in the `suc zero` case where we know there is not an answer, we can give back `nothing`:

```
2 evenEv : (n : ℕ) → Maybe (IsEven n)
  evenEv zero          = {! !}
  evenEv (suc zero)    = nothing
  evenEv (suc (suc n)) = {! !}
```

In the case of `zero`, there definitely *is* an answer, so we refine our hole with `just`:

```
2 evenEv : (n : ℕ) → Maybe (IsEven n)
  evenEv zero          = just {! !}
  evenEv (suc zero)    = nothing
  evenEv (suc (suc n)) = {! !}
```

...but a `just` of what? The type `IsEven zero` of the goal tells us, but we can also elicit an answer from Agda by invoking `Refine` (`C-c C-r` in Emacs and VS Code) on our hole:

```
2 evenEv : (n : ℕ) → Maybe (IsEven n)
  evenEv zero          = just zero-even
  evenEv (suc zero)    = nothing
  evenEv (suc (suc n)) = {! !}
```

At this step in `even?` we just recursed and we were done. However, that can't quite work here. The problem is that if we were to recurse,

we'd get a result of type `Maybe (IsEven n)`, but we need a result of type `Maybe (IsEven (suc (suc n)))`. What needs to happen then is for us to recurse, *inspect the answer,* and then, if it's `just`, insert a `suc-suc-even` on the inside. It all seems a little convoluted, but the types are always there to guide you should you ever lose the forest for the trees.

Agda does allow us to pattern match on the result of a recursive call. This is known as a `with` abstraction, and the syntax is as follows:

```
2 evenEv : (n : ℕ) → Maybe (IsEven n)
  evenEv zero       = just zero-even
  evenEv (suc zero) = nothing
  evenEv (suc (suc n)) with evenEv n    ❶
  ... | result = {! !}    ❷
```

At ❶, which you will note is on the *left* side of the equals sign, we add the word `with` and the expression we'd like to pattern match on. Here, it's `evenEv n`, which is the recursive call we'd like to make. At ❷, we put three dots, a vertical bar, and a name for the resulting value of the call we made, and then the equals sign. The important thing to note here is that `result` is a binding that corresponds to the result of having called `evenEv n`. This seems like quite a lot of ceremony, but what's cool is that we can now run `MakeCase` with argument `result` (`C-c C-c` in Emacs and VS Code) in the hole to pattern match on `result`:

```
2 evenEv : (n : ℕ) → Maybe (IsEven n)
  evenEv zero       = just zero-even
  evenEv (suc zero) = nothing
  evenEv (suc (suc n)) with evenEv n
  ... | just x  = {! !}
  ... | nothing = {! !}
```

In the case that `result` is `nothing`, we know that our recursive call failed, and thus that $n - 2$ is not even. Therefore, we too should return `nothing`. Similarly for the `just` case:

```
2 evenEv : (n : ℕ) → Maybe (IsEven n)
  evenEv zero       = just zero-even
  evenEv (suc zero) = nothing
  evenEv (suc (suc n)) with evenEv n
  ... | just x  = just {! !}
```

```
... | nothing = nothing
```

We're close to the end. Now we know that `x : IsEven n` and that our hole requires an `IsEven (suc (suc n))`. We can fill in the rest by hand, or invoke `Auto` (`C-c C-a` in Emacs and VS Code) to do it on our behalf.

```
evenEv : (n : ℕ) → Maybe (IsEven n)
evenEv zero       = just zero-even
evenEv (suc zero) = nothing
evenEv (suc (suc n)) with evenEv n
... | just x  = just (suc-suc-even x)
... | nothing = nothing
```

2.7 Addition

With the concept of induction firmly in our collective tool-belt, we are now ready to tackle a much more interesting function: addition over the naturals. Begin with the type, and bind the variables:

```
_+_ : ℕ → ℕ → ℕ
x + y = ?
```

At first blush, it's not obvious how we might go about implementing this. Perhaps we could mess about at random and see comes out, but while such a thing might be fun, it is rarely productive. Instead, we can go at this with a more structured approach, seeing what happens if we throw induction at the problem. Doing induction requires something to do *induction* on, meaning we can choose either `x`, `y` or both simultaneously. In fact, all three cases will work, but, as a general rule, if you have no reason to pick any parameter in particular, choose the first one.

In practice, doing induction means calling `MakeCase` (`C-c C-c` in Emacs and VS Code) on your chosen parameter, and then analyzing if a base case or an inductive case will help in each resulting equation. Usually, the values which are recursively-defined will naturally require recursion on their constituent parts. Let's now invoke `MakeCase` with argument `x` (`C-c C-c` in Emacs and VS Code):

```
2 _+_ : ℕ → ℕ → ℕ
  zero  + y = {! !}
  suc x + y = {! !}
```

Immediately a base case is clear to us; adding zero to something doesn't change it. In fact, that's the *definition* of zero. Thus, we have:

```
2 _+_ : ℕ → ℕ → ℕ
  zero  + y = y
  suc x + y = {! !}
```

The second case here clearly requires recursion, but it might not immediately be clear what that recursion should be. The answer is to squint and reinterpret `suc x` as $1 + x$, which allows us to write our left hand side as

$$(1 + x) + y$$

If we were to reshuffle the parentheses here, we'd get an $x + y$ term on its own, which is exactly what we need in order to do recursion. In symbols, this inductive case is thus written as:

$$(1 + x) + y = 1 + (x + y)$$

which translates back to Agda as our final definition of addition:

```
2 _+_ : ℕ → ℕ → ℕ
  zero  + y = y
  suc x + y = suc (x + y)
```

With a little thought, it's clear that this function really does implement addition. By induction, the first argument might be of the form `zero`, in which case it adds nothing to the result.

Otherwise, the first argument must be of the form `suc x`, in which case we assume `x + y` properly implements addition. Then, we observe the fact that $(1 + x) + y = 1 + (x + y)$. This is our first mathematical proof, although it is a rather "loose" one: argued out in words, rather than being *checked* by the computer. Nevertheless, it is a great achievement on our path towards mathematical fluency and finesse.

To wrap things up, we will add a fixity declaration for `_+_` so that it behaves nicely as an infix operator. We must choose a direction for

repeated additions to associate. In fact, it doesn't matter one way or another (and we used the fact that it doesn't matter in the inductive case of `_+_`.) But, looking forwards, we realize that subtraction *must* be left-associative in order to get the right answer, and therefore it makes sense that addition should also be left-associative. As a matter of convention, we will pick precedence 6 for this operator.

```
2 infixl 6 _+_
```

2.8 Termination Checking

There is a subtle point to be made about our implementation of `_+_`, namely that the parentheses are extremely important. Our last line is written as `suc x + y = suc (x + y)`, but if you were to omit the parentheses, the last line becomes `suc x + y = suc x + y`. Such a statement is unequivocally *true*, but is also computationally *unhelpful*. Since both sides of the equation are syntactically identical, Agda has no ability to make computational progress by rewriting one side as the other. In fact, if such a thing were allowed, it would let you prove anything at all! The only caveat would be that if you tried to inspect the proof, your computer would fall into an infinite loop, rewriting the left side of the equation into the right, forever.

Fortunately, Agda is smart enough to identify this case, and will holler, complaining about "termination checking" if you attempt to do it:

```
_+_ : ℕ → ℕ → ℕ
zero  + y = y
suc x + y = suc x + y
```

INFO WINDOW

```
Termination checking failed for the following functions:
  Sandbox-Naturals._+_
Problematic calls:
  suc x + y
```

By putting in the parentheses, `suc (x + y)` is now recursive, and, importantly, it is recursive on *structurally smaller* inputs than it

was given. Since the recursive call must be smaller (in the sense of there is one fewer `suc` constructor to worry about,) eventually this recursion must terminate, and thus Agda is happy.

Agda's termination checker can help keep you out of trouble, but it's not the smartest computer program around. The termination checker will only let you recurse on bindings that came from a pattern match, and, importantly, you're not allowed to fiddle with them first. As a quick, silly, illustration, we could imagine an alternative `ℕ'`, which comes with an additional `2suc` constructor corresponding to sucing twice:

```
module Example-Silly where
  open Chapter1-Agda
    using (not)

  data ℕ' : Set where
    zero : ℕ'
    suc  : ℕ' → ℕ'
    2suc : ℕ' → ℕ'
```

We can now write `even?'`, whose `2suc` case is `not` of `even?'` (`suc` n):

```
  even?' : ℕ' → Bool
  even?' zero     = true
  even?' (suc n)  = not (even?' n)
  even?' (2suc n) = not (even?' (suc n))
```

Tracing the logic here, it's clear to us as programmers that this function will indeed terminate. Unfortunately, Agda is not as smart as we are, and rejects the program:

INFO WINDOW

```
Termination checking failed for the following functions:
  Sandbox-Naturals.Example-Silly.even?'
Problematic calls:
  even?' (suc n)
```

The solution to termination problems like these is just to unwrap another layer of constructors:

```
⇤ 4 even?' : ℕ' → Bool
    even?' zero           = true
    even?' (suc n)        = not (even?' n)
    even?' (2suc zero)    = true
    even?' (2suc (suc n)) = not (even?' n)
    even?' (2suc (2suc n)) = even?' n
```

It's not the nicest solution, but it gets the job done.

2.9 Multiplication and Exponentiation

With addition happily under our belt, we will try our hand at multiplication. The approach is the same as with addition: write down the type of the operation, bind the variables, do induction, and use algebraic identities we remember from school to help figure out the actual logic. The whole procedure is really quite underwhelming once you get the hang of out!

After writing down the type and binding some variables, we're left with the following:

```
⇤ 2 _*_ : ℕ → ℕ → ℕ
    a * b = {! !}
```

We need to do induction on one of these bindings; because we have no reason to pick one or the other, we default to `a`:

```
2 _*_ : ℕ → ℕ → ℕ
    zero  * b = {! !}
    suc a * b = {! !}
```

The first case is immediately obvious; zero times anything is zero:

```
2 _*_ : ℕ → ℕ → ℕ
    zero  * b = zero
    suc a * b = {! !}
```

In order to solve what's left, we can dig into our mental cache of algebra facts. Recall that `suc a` is how we write $1 + a$ in Agda, thus:

$$\begin{aligned}(1 + a) \times b &= 1 \times b + a \times b \\ &= b + a \times b\end{aligned}$$

Therefore, our final implementation of multiplication is just:

```
_*_ : ℕ → ℕ → ℕ
zero  * b = zero
suc a * b = b + a * b
```

of course, we need to add a fixity definition for multiplication to play nicely with the addition operator. Since `_*_` is just a series of additions (as you can see from our implementation,) it makes sense to make multiplication also associate left. However, we'd like the expression `y + x * y` to parse as `y + (x * y)`, and so we must give `_*_` a higher precedence. Thus we settle on

```
infixl 7 _*_
```

Multiplication is just repeated addition, and addition is just repeated counting—as is made *abundantly* clear when working in our unary representation. We can repeat this pattern, moving upwards and building something that is "just repeated multiplication"—*exponentiation*:

Begin as always, with the type and the bound variables:

```
_^_ : ℕ → ℕ → ℕ
a ^ b = {! !}
```

We'd again like to do induction, but we must be careful here. Unlike addition and multiplication, exponentiation is not *commutative*. Symbolically, it is not the case that:

$$x^y \neq y^x$$

It's always a good habit to test claims like these. Because we're computer scientists we can pick $a = 2$, and because we're humans, $b = 10$. Doing some quick math, we see that this is indeed an inequality:

$$2^{10} = 1024 \neq 100 = 10^2$$

Due to this lack of commutativity, we must be careful when doing induction on `_^_`. Unlike in all of our examples so far, getting the right answer in exponentiation strongly depends on picking the right variable to do induction on. Think of what would happen if we were to do induction on `a`—we would somehow need to multiply smaller numbers together, each to the power of `b`.

Alternatively, doing induction on `b` means we get to multiply the same number together, each to a smaller power. That sounds much more correct, so let's pattern match on `b`:

```
2 _^_ : ℕ → ℕ → ℕ
  a ^ zero  = {! !}
  a ^ suc b = {! !}
```

The first case is a usual identity, namely that

$$a^0 = 1$$

while the second case requires an application of the exponent law:

$$a^{b+c} = a^b \times a^c$$

Instantiating this gives us:

$$\begin{aligned} a^{1+b} &= a^1 \times a^b \\ &= a \times a^b \end{aligned}$$

and thus:

```
2 _^_ : ℕ → ℕ → ℕ
  a ^ zero  = one
  a ^ suc b = a * a ^ b
```

As you can see, a judicious application of grade-school facts goes a long way when reasoning through these implementations. It makes sense why; algebraic identities are all about when are two expressions equal—and our Agda programs really and truly are defined in terms of *equations.*

2.10 Semi-subtraction

The natural numbers don't support subtraction, because we might try to take too much away, being forced to subtract what we don't have. Recall that there is no way to construct any negative naturals, and so this is not an operation we can implement in general.

However, we have an operation closely related to subtraction, which instead *truncates* at zero. That is, if the result would have gone negative, we just return zero instead. This operation is called "monus", and given the symbol `_∸_`, input as `\.-`.

Exercise (Easy) Define `_∸_ : ℕ → ℕ → ℕ`

Solution

```
2       _∸_ : ℕ → ℕ → ℕ
        x     ∸ zero = x
        zero  ∸ suc y = zero
        suc x ∸ suc y = x ∸ y
```

Just to convince ourselves everything works, let's write a few unit tests:

```
2   module Natural-Tests where
      open import Relation.Binary.PropositionalEquality

      _ : one + two ≡ three
      _ = refl

      _ : three ∸ one ≡ two
      _ = refl

      _ : one ∸ three ≡ zero
      _ = refl

      _ : two * two ≡ four
      _ = refl
```

Looks good to me!

You can find all of these goodies, and significantly more, in the standard library's `Data.Nat` module. Additionally, you also get support for natural literals. No more `four : ℕ`; just use `4 : ℕ` instead!

By this point, you should be starting to get a good handle on the basics of Agda—both syntactically, as well as how we think about modeling and solving problems. Let's therefore ramp up the difficulty and put that understanding to the test.

2.11 Inconvenient Integers

In this section we will tackle the integers, which have much more interesting mathematical structure than the naturals, and subsequently, present many more challenges.

The integers extend the natural numbers by reflecting themselves onto the negative side of the axis. The number line now goes off to infinity in two directions simultaneously, both towards infinity and towards *negative* infinity.

Some of the integers, are $\ldots, -1000, \ldots, -2, -1, 0, 1, \ldots, 47, \ldots$, but of course, there are many, many more. The set of integers is often written in blackboard bold, with the symbol `ℤ`, input as `\bZ`. `ℤ` might seem like a strange choice for the integers, but it makes much more sense in German, where the word for "number" is *Zahl*.

Mathematically, the integers are an *extension* of the natural numbers. That is, every natural number can be thought of as an integer, but there are some (infinitely many) integers that do not correspond to any natural. When modeling this problem in Agda, it would be nice if we could reuse the machinery we just built for natural numbers, rather than needing to build everything again from scratch. But before building integers the right way, we will first take an intentional wrong turn, in order to highlight some issues when data modeling in Agda.

Rather than pollute our global module with this intentional dead-end, we'll start a new module which we can later use to "rollback" the idea. By analogy with `ℕ`, which contains `zero` and `suc`, perhaps `ℤ` also has a constructor `pred` which we will interpret as "one less than":

```
⇤ 0 module Misstep-Integers₁ where
      data ℤ : Set where
        zero : ℤ
        suc  : ℤ → ℤ
        pred : ℤ → ℤ
```

Perhaps we could make an honest go with this definition for `ℤ`, but it has a major problem—namely, that numbers no longer have a unique representation. For example, there are now infinitely many ways of representing the number zero, the first five of which are:

- `zero`
- `pred (suc zero)`
- `suc (pred zero)`
- `pred (suc (pred (suc zero)))`

This is not just a problem for zero; in fact, every number has infinitely many encodings in this definition of `ℤ`. We could plausibly try to fix this problem by writing a function `normalize`, whose job it is to cancel out `suc`s with `pred`s, and vice versa. An honest attempt at such a function might look like this:

```
⇤ 2 normalize : ℤ → ℤ
    normalize zero            = zero
    normalize (suc zero)      = suc zero
    normalize (suc (suc x))   = suc (normalize (suc x))
    normalize (suc (pred x))  = normalize x
    normalize (pred zero)     = pred zero
    normalize (pred (suc x))  = normalize x
    normalize (pred (pred x)) = pred (normalize (pred x))
```

It's unclear prima facie whether this function correctly normalizes all integers. As it happens, it doesn't:

```
2 module Counterexample where
    open import Relation.Binary.PropositionalEquality

    _ : normalize (suc (suc (pred (pred zero)))) ≡ suc (pred zero)
    _ = refl
```

I'm sure there is a way to make `normalize` work correctly, but I suspect that the resulting ergonomics would be too atrocious to use in the real world. The problem seems to be that we can't be sure that the `suc`s and `pred`s are beside one another in order to cancel out. Perhaps we can try a different type to model integers which doesn't have this limitation.

2.12 Difference Integers

Instead, this time let's see what happens if we model integers as a pair of two natural numbers—one for the positive count, and another for the negative count. The actual integer in question is thus the difference between these two naturals.

Because we'd like to use the natural numbers, we must import them. But we anticipate a problem—addition over both the natural numbers and the integers is called `_+_`, but in Agda, there can only be one definition in scope with a given name. Our solution will be to `import Data.Nat`, but not to `open` it:

```
⇤ 0 module Misstep-Integers₂ where
    import Data.Nat as ℕ
```

This syntax gives us access to all of `Data.Nat`, but allows us to use `ℕ` as the name of the module, rather than typing out `Data.Nat` every

time. However, not every definition in `ℕ` will conflict with things we'd like to define about the integers, so we can *also* `open ℕ` in order to bring out the definitions we'd like to use unqualified:

```
2 open ℕ using (ℕ; zero; suc)
```

We are now ready to take our second attempt at defining the integers.

```
2 record ℤ : Set where
    constructor mkℤ
    field
      pos : ℕ
      neg : ℕ
```

This new definition of `ℤ` also has the problem that there are infinitely many representations for each number, but we no longer need to worry about the interleaving problem. To illustrate this, the first five representations of zero are now:

- `mkℤ 0 0`
- `mkℤ 1 1`
- `mkℤ 2 2`
- `mkℤ 3 3`
- `mkℤ 4 4`

Because the positive and negative sides are tracked independently, we can now write `normalize` and be confident that it works as expected:

```
⇤ 2 normalize : ℤ → ℤ
    normalize (mkℤ zero neg)           = mkℤ zero neg
    normalize (mkℤ (suc pos) zero)     = mkℤ (suc pos) zero
    normalize (mkℤ (suc pos) (suc neg)) = normalize (mkℤ pos neg)
```

Given `normalize`, we can give an easy definition for `_+_` over our "difference integers," based on the fact that addition distributes over subtraction:

$$(p_1 - n_1) + (p_2 - n_2) = (p_1 + p_2) - (n_1 + n_2).$$

In Agda, this fact looks equivalent, after replacing $a - b$ with `mkℤ a b`:

```
2 _+_ : ℤ → ℤ → ℤ
  mkℤ p₁ n₁ + mkℤ p₂ n₂
    = normalize (mkℤ (p₁ ℕ.+ p₂) (n₁ ℕ.+ n₂))

  infixl 5 _+_
```

Subtraction is similar, but is based instead on the fact that subtraction distributes over addition—that is:

$$\begin{aligned}(p_1 - n_1) - (p_2 - n_2) &= p_1 - n_1 - p_2 + n_2 \\ &= p_1 + n_2 - n_1 - p_2 \\ &= (p_1 + n_2) - (n_1 + p_2).\end{aligned}$$

This identity is exactly what's necessary to implement subtraction in Agda:

```
2 _-_ : ℤ → ℤ → ℤ
  mkℤ p₁ n₁ - mkℤ p₂ n₂
    = normalize (mkℤ (p₁ ℕ.+ n₂) (n₁ ℕ.+ p₂))

  infixl 5 _-_
```

Finally we come to multiplication, which continues to be implemented by way of straightforward algebraic manipulation. This time we need to multiply two binomials, which we can do by distributing our multiplication across addition twice. In symbols, the relevant equation is:

$$\begin{aligned}(p_1 - n_1) \times (p_2 - n_2) &= p_1p_2 - p_1n_2 - n_1p_2 + n_1n_2 \\ &= p_1p_2 + n_1n_2 - p_1n_2 - n_1p_2 \\ &= (p_1p_2 + n_1n_2) - (p_1n_2 + n_1p_2).\end{aligned}$$

Again, in Agda:

```
2 _*_ : ℤ → ℤ → ℤ
  mkℤ p₁ n₁ * mkℤ p₂ n₂
    = normalize
        (mkℤ (p₁ ℕ.* p₂ ℕ.+ n₁ ℕ.* n₂)
             (p₁ ℕ.* n₂ ℕ.+ p₂ ℕ.* n₁))

  infixl 6 _*_
```

While each and every one of our operations here do in fact work, there is nevertheless something dissatisfying about them—namely,

our requirement that each of `_+_`, `_-_`, and `_*_` end in a call to `normalize`. This is by no means the end of the world, but it is *inelegant.* Ideally, we would like each of these elementary operations to just get the right answer, without needing to run a final pass over each result.

In many ways, what we have built with our difference integers comes from having "computer science brain" as opposed to "math brain." We built something that gets the right answer, but does it by way of an intermediary computation which doesn't correspond to anything in the problem domain. There are all these calls to `normalize`, but it's unclear what exactly `normalize` *actually means*– as opposed to what it *computes.*

Where this problem will really bite us is when we'd like to start doing proofs. What we'd really like to be able to say is that "these two numbers are the same," but, given our implementation, all we can say is "these two numbers are the same *after a call to* `normalize`." It is possible to work around this problem, as we will see in sec. 7.9, but the solution is messier than the problem, and is best avoided whenever we are able.

The crux of the matter is that we know what sorts of rules addition, subtraction and multiplication ought to abide by, but it's much less clear what we should expect of `normalize`. This function is a computational crutch—nothing more, and nothing less. If we rewind to the point at which we introduced `normalize`, we realize that this crutch was designed to work around the problem that there are non-unique representations for each number. If we could fix that problem directly, we could avoid `normalize` and all the issues that arise because of it.

2.13 Unique Integer Representations

The important takeaway from our last two wrong turns is that we should strive for unique representations of our data whenever possible.

Let's take one last misstep in attempting to model the integers before we get to the right tack. Our difference integers went wrong because they were built from two different naturals, which we implicitly subtracted. Perhaps we were on the right track using naturals, and would be more successful if we had only one at a time.

So in this attempt, we will again reuse the natural numbers, but now build integers merely by tagging whether that natural is postive

or negative:

```
⇤ 0
module Misstep-Integers₃ where
  open import Data.Nat

  data ℤ : Set where
    +_ : ℕ → ℤ
    -_ : ℕ → ℤ
```

This approach is much more satisfying than our previous attempt; it allows us to reuse the machinery we wrote for natural numbers, and requires us only to wrap them with a tag. The syntax is a little weird, but recall that the underscores correspond to syntactic "holes," meaning the following are both acceptable integers:

```
⇤ 2
_ : ℤ
_ = - 2

_ : ℤ
_ = + 6
```

Note that the spaces separating `-` from `2`, and `+` from `6` are *necessary*. Agda will complain very loudly—and rather incoherently—if you forget them.

While our second approach dramatically improves on the syntax of integers and eliminates most problems from `Misstep-Integers₂`, there is still one small issue: there is still a non-unique representation for zero, as we can encode it as being either positive or negative:

```
2
_ : ℤ
_ = + 0

_ : ℤ
_ = - 0
```

Perhaps there are some number systems in which it's desirable to have (distinct) positive and negative zeroes, but this is not one of them. We are stuck with two uncomfortable options—keep the two zeroes and insist that they are in fact two different numbers, or duplicate all of our proof effort and somehow work in the fact that the two zeroes are different encodings of the same thing. Such a thing can work, but it's inelegant and pushes our poor decisions down the line to every subsequent user of our numbers.

There really is no good solution here, so we must conclude that this attempt is flawed too. However, it points us in the right direction. Really, the only problem here is our *interpretation of the syntax.* Recall that the symbols induced by constructors are *just names,* and so we can rename our constructors in order to change their semantics.

This brings us to our and final (and correct) implementation for the integers:

```
⇤ 0 module Sandbox-Integers where
      import Data.Nat as ℕ
      open ℕ using (ℕ)

      data ℤ : Set where
        +_ : ℕ → ℤ
        -[1+_] : ℕ → ℤ
```

You'll notice this definition of ℤ is identical to the one from `Misstep-Integers`$_3$; the only difference being that we've renamed `-_` to `-[1+_]`. This new name suggests that `-[1+ n ]` corresponds to the number $-(1+n) = -n-1$. By subtracting this 1 from all negative numbers, we have removed the possibility of a negative zero.

Given this machinery, we can now name three particularly interesting integers:

```
⇤ 2 0ℤ : ℤ
    0ℤ = + 0

    1ℤ : ℤ
    1ℤ = + 1

    -1ℤ : ℤ
    -1ℤ = -[1+ 0 ]
```

Of course, we'd still like our `suc` and `pred` functions that we postulated our first time around. The constructors are already decided on our behalf, so we'll have to settle for functions instead:

```
  2 suc : ℤ → ℤ
    suc (+ x)          = + ℕ.suc x
    suc -[1+ ℕ.zero ]  = 0ℤ
    suc -[1+ ℕ.suc x ] = -[1+ x ]
```

If `suc`'s argument is positive, it makes it more positive. If it's negative, it makes it less negative, possibly producing zero in the process. Dually, we can define `pred` which makes its argument more negative:

```
pred : ℤ → ℤ
pred (+ ℕ.zero)  = -1ℤ
pred (+ ℕ.suc x) = + x
pred -[1+ x ]    = -[1+ ℕ.suc x ]
```

2.14 Pattern Synonyms

It might be desirable to negate an integer; turning it negative if it's positive, and vice versa. `-_` is a natural name for this operation, but its implementation is not particularly natural:

```
-_ : ℤ → ℤ
- (+ ℕ.zero)  = 0ℤ
- (+ ℕ.suc x) = -[1+ x ]
- -[1+ x ]    = + ℕ.suc x
```

When converting back and forth from positive to negative, there's an annoying `ℕ.suc` that we need to be careful to not forget. This irritant is an artifact of our encoding. We now have the benefit of unique representations for all numbers, but at the cost of the definition not being *symmetric* between positive and negative numbers.

Thankfully, Agda has a feature that can help us work around this problem. *Pattern synonyms* allow us to define new constructor syntax for types. While `ℤ` is and always will be made up of `+_` and `-[1+_]`, we can use pattern synonyms to induce other ways of thinking about our data.

For example, it would be nice if we could also talk about `+[1+_]`. This doesn't give us any new power, as it would always be equivalent to `+ ℕ.suc x`. Nevertheless, our definition of `-_` above *does* include a `+ (ℕ.suc x)` case, so this pattern does seem like it might be useful.

We can define a pattern synonym with the `pattern` keyword. Patterns look exactly like function definitions, except that they build constructors (highlighted red, and can be used in pattern matches) rather than (blue) function definitions.

```
pattern +[1+_] n = + ℕ.suc n
```

Let's also define a pattern for zero:

```
pattern +0 = + ℕ.zero
```

These two patterns give us the option to define functions symmetrically with respect to the sign of an integer. Where before in `-_` we had to pattern match on two cases, `+_` and `-[1+_]`, we can now instead choose to match into *three* cases: `+0`, `+[1+_]` and `-[1+_]`.

Let's use our new patterns to rewrite `-_`, leading to a significantly more elegant implementation:

```
-_ : ℤ → ℤ
- +0       = +0
- +[1+ x ] = -[1+ x ]
- -[1+ x ] = +[1+ x ]
```

What exactly is going on with these pattern synonyms? We haven't actually changed the constructors of `ℤ`; merely, we've extended our type with different ways of thinking about its construction. Behind the scenes, whats really happening when we write `+[1+ n ]` is that Agda simply rewrites it by the pattern equation—in this case, resulting in `+ (ℕ.suc n)`. It's nothing magic, but it does a lot in terms of ergonomics.

When should we use a `pattern` instead of a function definition? On the surface, they might seem quite similar. You could imagine we might define `+[1+_]` not as a pattern, but as a function:

```
+[1+_] : ℕ → ℤ
+[1+_] n = + ℕ.suc n
```

The problem is such a definition creates `+[1+_]` as a function definition, and note its blue color. In Agda, we're allowed to do pattern matching only on red values, which correspond to constructors and pattern synonyms. On the left side of the equality, Agda changes its namespace, and will only recognize known red identifiers. Anything it doesn't recognize in this namespace becomes a black binding, rather than a reference to a blue function. This is a reasonable limitation. In general, function definitions can do arbitrary computation, which would obscure—if not render uncomputable—Agda's ability to pattern match on the left side of an equality. Thus, blue bindings are strictly disallowed on the pattern matching side.

This is the reason behind the existence of pattern synonyms. Pattern definitions are required to be made up of only red constructors

and black bindings on both sides. In doing so, we limit their expressiveness, but *because* of that limitation, we have restricted them in such a way as to be usable in a pattern context.

As a rule of thumb, you should define a pattern synonym whenever you notice yourself frequently using the same series of constructors together. Pattern synonyms are valuable in providing a different lens into how you put your data together.

2.15 Integer Addition

With a satisfactory definition for the integers, and having completed our discussion of pattern synonyms, it is now time to implement addition over `ℤ`. As usual, we will intentionally go down the wrong (but obvious) path in order to help you develop antibodies to antipatterns.

Our particular misstep this time around will be to "bash" our way through the definition of addition—that is, match on all three of `+0`, `+[1+_]` and `-[1+_]` for both arguments of `_+_`. There are a few easy cases, like when one side is zero, or when the signs line up on both sides. After filling in the obvious details, we are left with:

```
2 module Naive-Addition where
  _+_ : ℤ → ℤ → ℤ
  +0       + y        = y
  +[1+ x ] + +0       = +[1+ x ]
  +[1+ x ] + +[1+ y ] = +[1+ 1 ℕ.+ x ℕ.+ y ]
  +[1+ x ] + -[1+ y ] = {! !}
  -[1+ x ] + +0       = -[1+ x ]
  -[1+ x ] + +[1+ y ] = {! !}
  -[1+ x ] + -[1+ y ] = -[1+ 1 ℕ.+ x ℕ.+ y ]
```

It's not clear exactly how to fill in the remaining holes, however. We must commit to a constructor of `ℤ`, which mathematically means committing to the sign of the result—but in both cases, the sign depends on whether `x` or `y` is bigger.

Of course, we could now do a pattern match on each of `x` and `y`, and implement integer addition inductively on the size of these natural numbers. That feels unsatisfactory for two reasons—the first is that this function is already doing a lot, and induction on the naturals feels like a different sort of thing than the function is already doing. Our second dissatisfaction here is that the two remaining

holes are symmetric to one another; since we know that $x + y = y + x$, we know that the two holes must be filled in with equivalent implementations. Both of these reasons point to the fact that we need a helper function.

Recall in sec. 2.10 when we implemented the monus operator, which performed truncated subtraction of natural numbers. The only reason it was required to truncate results was that we didn't have a satisfactory type in which we could encode the result if it went negative. With the introduction of ℤ, we now have room for all of those negatives. Thus, we can implement a version of subtraction whose inputs are the naturals, but whose output is an integer. We'll call this operation `_⊖_`, input like you'd expect as `\o--`

```
⇤ 2 _⊖_ : ℕ → ℕ → ℤ
    ℕ.zero  ⊖ ℕ.zero  = +0
    ℕ.zero  ⊖ ℕ.suc n = -[1+ n ]
    ℕ.suc m ⊖ ℕ.zero  = +[1+ m ]
    ℕ.suc m ⊖ ℕ.suc n = m ⊖ n
```

By implementing `_+_` in terms of `_⊖_`, we can factor out a significant portion of the logic. Note that all we care about is whether the signs of the arguments are the same or different, meaning we can avoid the pattern matches on `+0` and `+[1+_]`, instead matching only on `+_`:

```
2 infixl 5 _+_

  _+_ : ℤ → ℤ → ℤ
  + x      + + y      = + (x ℕ.+ y)
  + x      + -[1+ y ] = x ⊖ ℕ.suc y
  -[1+ x ] + + y      = y ⊖ ℕ.suc x
  -[1+ x ] + -[1+ y ] = -[1+ x ℕ.+ ℕ.suc y ]
```

This new definition of `_+_` shows off the flexibility of Agda's parser. Notice how we're working with `+_` and `_+_` simultaneously, and that Agda isn't getting confused between the two. In fact, Agda is less confused here than we are, as the syntax highlighting on the first line of the definition gives us enough to mentally parse what's going on. The blue identifier on the left of the equals sign is always the thing being defined, and its arguments must always be red constructors or black bindings. Practice your mental parsing of these definitions, as they will only get harder as we move deeper into abstract mathematics.

Having implemented addition is the hard part. We can implement subtraction trivially, via addition of the negative:

```
infixl 5 _-_
_-_ : ℤ → ℤ → ℤ
x - y = x + (- y)
```

Last but not least, we can define multiplication, again as repeated addition. It's a little trickier this time around, since we need to recurse on positive and negative multiplicands, but the cases are rather simple. Multiplication by zero is zero:

```
infixl 6 _*_
_*_ : ℤ → ℤ → ℤ
x * +0 = +0
```

Multiplication by either 1 or -1 merely transfers the sign:

```
x * +[1+ ℕ.zero ] = x
x * -[1+ ℕ.zero ] = - x
```

and otherwise, multiplication can just perform repeated addition or subtraction on one argument, moving towards zero:

```
x * +[1+ ℕ.suc y ] = (+[1+ y ] * x) + x
x * -[1+ ℕ.suc y ] = (-[1+ y ] * x) - x
```

Thankfully, our hard work is rewarded when the unit tests agree that we got the right answers:

```
module Tests where
  open import Relation.Binary.PropositionalEquality

  _ : - (+ 2) * - (+ 6) ≡ + 12
  _ = refl

  _ : (+ 3) - (+ 10) ≡ - (+ 7)
  _ = refl
```

2.16 Wrapping Up

Our achievements in this chapter are quite marvelous. Not only have we defined the natural numbers and the integers, but we've given their everyday operations, and learned a great deal about Agda in the process. Rather famously, in the Principia Mathematica, Whitehead and Russell took a whopping 379 pages to prove that $1+1=2$. While we haven't yet *proven* this fact, we are well on our way, and will do so in the next chapter when we reflect on the deep nature of proof.

Before closing, we will explicitly list out our accomplishments from the chapter, and export them from the standard library for use in the future. In sec. 2.1 we constructed the natural numbers `ℕ` and their constructors `zero` and `suc`. Addition comes from sec. 2.7, while multiplication and exponentiation come from sec. 2.9. The monus operator `_∸_` is from sec. 2.10.

```
⇤ 0 open import Data.Nat
      using (ℕ; zero; suc; _+_; _*_; _^_; _∸_)
      public
```

We also gave definitions for the first four positive naturals:

```
⇤ 0 open Sandbox-Naturals
      using (one; two; three; four)
      public
```

While discussing the natural numbers, we looked at two notions of evenness in sec. 2.5. We'd like to export `IsEven` and its constructors `zero-even` and `suc-suc-even`. For succinctness, however, we'll rename those constructors to `z-even` and `ss-even` by way of a `renaming` import modifier:

```
⇤ 0 open Sandbox-Naturals
      using (IsEven)
      renaming ( zero-even    to z-even
               ; suc-suc-even to ss-even
               )
      public
```

In sec. 2.6, we constructed the `Maybe` type, which we used to wrap functions' return types in case there is no possible answer. If the function can't return anything, we use the constructor `nothing`, but if it was successful, it can use `just`:

```
⇤ 0 open import Data.Maybe
      using (Maybe; just; nothing)
      public
```

Discussing the integers made for an interesting exercise, but we will not need them again in this book, and therefore will not export them. Nevertheless, if you'd like to use them in your own code, you can find all of our definitions under `Data.Int`.

⌨ UNICODE IN THIS CHAPTER

```
₀ U+2080 SUBSCRIPT ZERO (\_0)
₁ U+2081 SUBSCRIPT ONE (\_1)
₂ U+2082 SUBSCRIPT TWO (\_2)
₃ U+2083 SUBSCRIPT THREE (\_3)
ℕ U+2115 DOUBLE-STRUCK CAPITAL N (\bN)
ℤ U+2124 DOUBLE-STRUCK CAPITAL Z (\bZ)
→ U+2192 RIGHTWARDS ARROW (\to)
∸ U+2238 DOT MINUS (\.-)
≡ U+2261 IDENTICAL TO (\==)
⊖ U+2296 CIRCLED MINUS (\o-)
```

CHAPTER 6

Decidability

```
0 module Chapter6-Decidability where
```

In the last chapter, we thoroughly investigated the notion of doing proof work in Agda. We gave a propositional definition what it means for two things to be equal, derived relevant properties of equality, built a domain-specific language for doing equational reasoning, and proved many facts about the algebra of the natural numbers.

Perhaps now we can smugly think that we know all there is to know about proofs—but this is patently false. For example, we haven't the notion of falseness itself! Furthermore, everything we've built so far works only for *statically-known* values. But is it possible that we can use dependent types in contexts where we don't know all the values we need to work with? Maybe the values came from the user via an interactive system. Are we forced to throw away everything we've done and degrade our guarantees to those of a weaker programming language?

Thankfully the answer to all of these is "no," and we will explore each in this chapter. Additionally, we will also get our hands dirty modeling and proving facts about more computer-sciencey objects. After all, these proof techniques are applicable outside of the ivory tower, too!

Prerequisites

```
0     open import Chapter1-Agda
        using (Bool; true; false)
```

```
⇤ 0  open import Chapter2-Numbers
       using (ℕ; zero; suc; _+_)
```

```
⇤ 0  open import Chapter3-Proofs
       using ( _≡_; module PropEq; module ≡-Reasoning
             ; suc-injective)
     open PropEq
```

```
  0  open import Chapter4-Relations
       using (Level; _⊔_; lsuc; Σ; _,_)
```

6.1 Negation

Recall that we've now seen how to express that two values are (propositionally) equal, via the `_≡_` type, proven via `refl`. We'd now like to work out a means of discussing *inequality!*

Perhaps you might think we can make a slight tweak to the `_≡_` construction. While `refl : x ≡ x`, perhaps we could define `_≢_` (input as `\==n`) as something like:

```
⇤ 0  module Sandbox-Inequality where
       data _≢_ {A : Set} : A → A → Set where
         ineq : {x y : A} → x ≢ y
```

Unfortunately, this approach does not—and cannot—work. While we've attempted to assert that `x` and `y` are inequal by virtue of being different bindings of `A`, Agda gives us no guarantees that they are *distinct* values of `A`! It's as if were to write a function:

```
⇤ 2  f : ℕ → ℕ → ℕ
     f x y = x + y
```

Here, `x` and `y` are different *variables,* but that doesn't mean they must have different *values.* We can happily evaluate this at `f 2 2`. Thus, just because the variables are distinct doesn't mean the values must be. This attempt at modeling `_≢_` is therefore fundamentally flawed. Let's try something else.

Recall the *principle of explosion,* the property of a contradictory system, which states that "from false, anything is possible." This

gives us a hint as to how we might go about encoding an inequality. Rather than treating it as such, we can instead rewrite $x \neq y$ as "$x = y$ is false."

Let's try our hand at this principle of explosion. The idea being, in order to show that some claim `P` is false, we instead encode the problem as "from a proof of `P` we can derive anything else."

```
⇤ 0  module Sandbox-Explosion where
       _IsFalse : Set → Set₁    ❶
       P IsFalse = P → {A : Set} → A
```

You'll notice at ❶ that the type of `_IsFalse` is `Set → Set₁`, which is our first time seeing a `Level` in the wild. Such is necessary because the definition of `_IsFalse` expands out to a polymorphically-quantified `A : Set`. Whenever we have a `Set` parameter, the defining object itself must live in `Set₁`.

From here, We can now try to find a proof that two is not equal to three, as per:

```
2  2≠3 : (2 ≡ 3) IsFalse
   2≠3 = ?
```

Since `_IsFalse` expands to a function type, we can (rather unintuitively) do a `MakeCase` without any argument (`C-c C-c` in Emacs and VS Code) here to get a *parameter* of type `2 ≡ 3`:

```
2  2≠3 : (2 ≡ 3) IsFalse
   2≠3 x = {! !}
```

Of course, you and I know that there is no such proof that `2 ≡ 3`. Nevertheless, we can ask Agda to `MakeCase` with argument `x` (`C-c C-c` in Emacs and VS Code) here, which will produce a strange result:

```
2  2≠3 : (2 ≡ 3) IsFalse
   2≠3 ()
```

What happened is that Agda noticed that *there are no constructors* for the type `2 ≡ 3`. Therefore, this function `2≠3` can never be called, since there is argument that will typecheck. Since Agda notices that the whole thing is moot anyway, we aren't required to write any implementation, which explains the funny `()` notation, also known

as an *absurd pattern match*. Absurd pattern matches don't require an definition, and we illustrate that by not giving any equals sign.

By virtue of having written a definition of `2≠3` that typechecks, we have successfully proven the fact that two is not equal to three. Mission accomplished! Nevertheless, while this is an absolutely valid encoding and serves to illustrate the idea, it's not quite how we usually write negation in Agda. For that, we need the bottom type.

6.2 Bottom

While it's acceptable to use the principle of explosion directly, there is a simpler way of encoding the problem, namely by "factoring out" the pieces.

Rather than showing all contradictions lead to explosion, we can instead say all contradictions lead to a specific type, and then show that all values of that type lead to explosion. The resulting machinery means you can always use the principle of explosion if you'd like, but in practice allows for simpler-to-use types when doing negative proofs.

This "pre-explosive" type we'll work with is called the *bottom* type, written `⊥` and input as `\bot`. Its definition is short and sweet:

```
⇤ 0 module Definition-Bottom where
      data ⊥ : Set where
```

That's it. There are no constructors for `⊥`, meaning that *every* pattern match on `⊥` must be absurd as we saw in the last section. Besides a slightly different type signature, we can show `2≠3` again—with an identical proof, but this time using bottom:

```
2 2≠3 : 2 ≡ 3 → ⊥
  2≠3 ()
```

Why does this work? Recall the definition of a function: for any input in the domain, it must return an element in the codomain. But there are no elements in bottom, so any function whose codomain is bottom cannot possibly be a function—as it has nowhere to send inputs! The only possible workaround here is if the function's domain *also* has no elements.

By this argument, any function we can successfully define which maps into bottom necessarily constrains at least one of its parameters to also have no elements.

We still need to show that an element of `⊥` leads to the principle of explosion. This function is called "bottom elimination," and is itself another easy proof:

```
2 ⊥-elim : {A : Set} → ⊥ → A
  ⊥-elim ()
```

Mathematically, proofs of this flavor are called *vacuously true*. Which is to say, they are true only because they presuppose a contradiction. From another perspective, vacuously true proofs have already "baked in" the principle of explosion.

Nevertheless, we can give a vacuous proof that all elements of bottom—of which there are none—are equal, or, at least, that they would be equal if there were any:

```
2 ⊥-unique : {x y : ⊥} → x ≡ y
  ⊥-unique {x} = ⊥-elim x
```

We have now shown that bottom is a satisfactory definition of false, and that functions into bottom are therefore proofs that at least one of their parameters is a contradiction. Hence, we can define `¬_` (`\neg`), which states that a given mathematical statement is false:

```
2 ¬_ : {ℓ : Level} → Set ℓ → Set ℓ
  ¬ P = P → ⊥
  infix 3 ¬_
```

In this definition of `¬_`, we have used universe polymorphism in order to give negation for every possible `Level`, simultaneously. This is a common pattern in Agda—to make each and every `Set` you bind universe-polymorphic.

6.3 Inequality

With a satisfactory definition for propositional negation, we can define inequality (input as `\neq`):

```
2 _≠_ : {A : Set} → A → A → Set
  x ≠ y = ¬ x ≡ y
  infix 4 _≠_
```

When defining new relations, it's always a reward experience to investigate which properties hold of the new construction. As you might expect, inequality is symmetric:

```
2 ≠-sym : {A : Set} {x y : A} → x ≠ y → y ≠ x
  ≠-sym x≠y y=x = x≠y (sym y=x)
```

and it is *not* reflexive. Perhaps this is obvious to you, but if not, recall that reflexivity would require us to show `x ≠ x` for any `x`, which is exactly the *opposite* of what we're trying to encode here.

Interestingly however, we *can* prove that `_≠_` isn't reflexive by showing that if there were construction, it would lead immediately to contradiction. Let's show this in a few steps. First, we will define a type `Reflexive` which will state the reflexivity property. This isn't strictly necessary, but it lessens the cognitive burden later down the line.

A relation `_≈_ : A → A → Set ℓ` is *reflexive* if, for any `x : A`, we can construct `x ≈ x`. This definition can be encoded as a type:

```
2 Reflexive
    : {c ℓ : Level} {A : Set c}
    → (A → A → Set ℓ)
    → Set (ℓ ⊔ c)
  Reflexive {A = A} _≈_ = {x : A} → x ≈ x    ❶
```

Having a value of `Reflexive _≈_` states that `_≈_` is just such a reflexive relation. Note that, at ❶, `_≈_` is on the left side of the equals sign, and is thus just a parameter—allowing us to state *which* relation is reflexive.

To illustrate exactly what's happening here, we can show that propositional equality is reflexive:

```
2 ≡-refl : {A : Set} → Reflexive {A = A} _≡_
  ≡-refl = refl
```

Notice how the type `Reflexive _≡_` expands to the type of `refl`, that is, `{x : A} → x ≡ x`. We are required to explicitly bind `A : Set` here, so that we can use it to fill in the implicit `A` parameter of `Reflexive`. In principle, Agda could be inferred from the type of `_≡_`, but Agda has no way of knowing if we'd like to talk about `_≡_` in its fully-polymorphic type, or if we'd like it specialized to some particular type like `ℕ`. The distinction isn't extremely poignant in this example, but there do exist monomorphic relations which we might still want to say are reflexive.

Nevertheless, we have shown that `_≡_` is reflexive by giving a proof that `refl` always exists.

Contrasting against this example the proof that `_≠_` is *not* reflexive. The type of such a statement is a very subtle thing:

```
¬≠-refl : ¬ ({A : Set} → Reflexive {A = A} _≠_)    ❶
¬≠-refl = ?
```

Compare this type to the more "obvious" type:

```
¬≠-refl-bad : {A : Set} → ¬ Reflexive {A = A} _≠_    ❷
¬≠-refl-bad = ?
```

The difference between ❶ and ❷ is in the placement of the binder for `A : Set`. In the latter type, in the latter, we receive a specific `A`, and are then required to give back a proof that there is no `≠-refl` *for that specific type.* Contrast that against ❶, in which we are required to show that there does not exist a `≠-refl` that works for *every* type `A`.

End users of `¬≠-refl` don't care one way or another, since they only ever need one type at a time (or can use a variable if they'd like to show this for many types.) Nevertheless, the difference here is absolutely crucial when we'd actually like to *prove* the thing. Let's look at this more closely.

At a very high level, the proof here is "`≠-refl` must be false because it's the negation of `≡-refl`, which is easily shown to be true." While the type of the argument to `¬≠-refl` is `Reflexive _≠_`, we can use `TypeContext` (`C-u C-u C-c C-,` in Emacs and VS Code) to ask Agda to expand out this definition, resulting in:

INFO WINDOW

```
x ≡ x → ⊥
```

Under the lens of this expanded type, it seems reasonable to call the argument to our function `¬≡-refl`, as in the following:

```
¬≠-refl : ¬ ({A : Set} → Reflexive {A = A} _≠_)
¬≠-refl ¬≡-refl = {! !}
```

Of course, we do in fact have a proof of `x ≡ x`, namely `refl`, so we can try solving the hole as:

```
¬≠-refl : ¬ ({A : Set} → Reflexive {A = A} _≠_)
¬≠-refl ¬≡-refl = ¬≡-refl refl
```

Uh oh, something went wrong. This yellow background means that elaboration failed, which means that Agda was unable to determine some of the implicit arguments. We can help it out by explicitly introducing holes for each implicit argument and solving them ourselves:

```
¬≠-refl : ¬ ({A : Set} → Reflexive {A = A} _≠_)
¬≠-refl ¬≡-refl = ¬≡-refl { ? } { ? } refl
```

You'll notice that the yellow warning has disappeared, and we're left only with some holes to fill. The first hole has type `Set`, while the second has type `?0`, meaning its type depends on the answer to our first hole. We're free to choose any type we'd like for this first hole, so let's try `ℕ`:

```
¬≠-refl : ¬ ({A : Set} → Reflexive {A = A} _≠_)
¬≠-refl ¬≡-refl = ¬≡-refl {ℕ} { ? } refl
```

This refines our other hole to have type `ℕ`, and now all we need is to pick an arbitrary natural:

```
¬≠-refl : ¬ ({A : Set} → Reflexive {A = A} _≠_)
¬≠-refl ≠-refl = ≠-refl {ℕ} {0} refl
```

What was to be demonstrated has thus been proved. But how? We've shown that there can be no proof of `Reflexive _≠_` for *every type* `A`, because if there were such a proof, we could instantiate it at the naturals, and use it to prove that `0 ≠ 0`. Such is obviously bunk, because we can refute `0 ≠ 0`, and therefore the argument rests on false premises.

Let's see what goes wrong if we try the same approach on `¬≠-refl-bad`. Since `A` is now a top-level parameter, we can bind it in the function body, meaning we have one fewer implicit to fill in:

```
¬≠-refl-bad : {A : Set} → ¬ Reflexive {A = A} _≠_
¬≠-refl-bad {A} ¬≡-refl = ¬≡-refl { ? } refl
```

But we run into problems trying to fill this last hole. It unfortunately has type `A`, *for some unknown type* `A`*!* We can't possibly fill this in, because we don't know what type `A` is. In fact, `A` could be `⊥`, in which case there aren't even any values we *could* put here.

The takeaway from this argument is the importance of where you put your quantifiers. When the necessary parameters are *inside* the negation, we are able to instantiate them at our convenience when looking for a counterexample. But if they are outside the negation, we are at the mercy of the proof-caller. Such might dramatically change how we'd go about showing a counterexample, and often precludes it entirely.

6.4 Negation Considered as a Callback

There is an apt computational perspective to this problem of negating quantifiers, which exposes itself when we expand all the definitions. After fully normalizing the types of both `¬≠-refl` and `¬≠-refl-bad`, we are left with these:

```
¬≠-refl     : ({A : Set} → {x : A} → x ≡ x → ⊥) → ⊥
¬≠-refl-bad : {A : Set} → ({x : A} → x ≡ x → ⊥) → ⊥
```

The salient feature of these two types is that they both take functions as arguments. Whenever you see a function as an argument, you can interpret this argument as a *callback.* Viewed as such, the question becomes *who is responsible for providing what?* In the first case, it is the *implementer* of `¬≠-refl` who gets to pick `A`, while in the second, the *caller* of `¬≠-refl-bad` is responsible. And true to the name of this function, the caller might very well pick an antagonistic `A` in an attempt to thwart us!

6.5 Intransitivity of Inequality

In the same vein as `¬≠-refl`, we can also show that `_≠_` doesn't satisfy transitivity. The pen and paper proof is straightforward, since if inequality were transitive we could show:

$$\begin{aligned} 2 &\neq 3 \\ &\neq 2 \end{aligned}$$

This is known in the business as a "whoopsie daisy." Such a counterexample shows inequality cannot be transitive, because if it were, we could show reflexivity, which we already know is false. We can prove the non-transitivity of inequality more formally in Agda by giving a definition for transitivity:

```
Transitive
  : {c ℓ : Level} {A : Set c}
  → (A → A → Set ℓ)
  → Set (c ⊔ ℓ)
Transitive {A = A} _≈_ = {x y z : A} → x ≈ y → y ≈ z → x ≈ z
```

Spend a moment to parse and understand this type for yourself. Do you agree that this corresponds to the definition of transitivity (sec. 3.8)?

Showing the counterexample is easy—following our exact pen and paper proof above—since we already have a proof `2≠3`. Given a hypothetical `≠-trans`, we could combine this with `sym 2≠3`, to get a proof that `2 ≠ 2`. Such a thing is in direct contradiction with `refl : 2 ≡ 2`, which acts as the refutation:

```
¬≠-trans : ¬ ({A : Set} → Transitive {A = A} _≠_)
¬≠-trans ≠-trans = ≠-trans {ℕ} 2≠3 (≠-sym 2≠3) refl
```

6.6 No Monus Left-Identity Exists

To illustrate using negation in the real world, let's prove a claim made back in sec. 3.6: that there is no left-identity for the monus operation `_∸_`. First, let's import it.

```
open import Relation.Nullary
  using (¬_)

module Example-NoMonusLeftIdentity where
  open Chapter2-Numbers using (_∸_)
```

In order to prove this, let's first think about what we'd like to say, and what the actual argument is. We'd like to say: "it is not the case that there exists a number `e : ℕ` such that for any `x : ℕ` it is the case that `e ∸ x ≡ x`." A bit of a mouthful certainly, but it encodes rather directly. Replace "there exists a number `e : ℕ` such that" with `Σ ℕ (λ e → ?)`—recalling `Σ` from sec. 4.2. After replacing the "for any `x : ℕ`" with `(x : ℕ) → ?`, we're ready to go:

```
¬∸-identity¹ : ¬ Σ ℕ (λ e → (x : ℕ) → e ∸ x ≡ x)
¬∸-identity¹ = ?
```

Proving this is straightforward; we take the refutation at its word, and therefore get a function `e-is-id : (x : ℕ) → e ∸ x ≡ x`. If we instantiate this function at $x = 0$, it evaluates to a proof that `e ≡ 0`. Fair enough. But we can now instantiate the same function at $x = 1$, which then produces a proof that `e ∸ 1 ≡ 1`. However, we already know that $e = 0$, so we can replace it, getting `0 ∸ 1 ≡ 1`, which itself simplifies down to `0 ≡ 1`. Such is an obvious contradiction, and therefore there is no such identity for monus.

We can write the same argument in Agda by binding and splitting our argument, and then using a `with`-abstraction (sec. 2.6) on `e-is-id` at both 0 and 1:

```
¬∸-identity¹ : ¬ Σ ℕ (λ e → (x : ℕ) → e ∸ x ≡ x)
¬∸-identity¹ (e , e-is-id)
  with e-is-id 0 | e-is-id 1
... | e≡0 | e-1≡1 = ?
```

If we now pattern match (`MakeCase` (`C-c C-c` in Emacs and VS Code)) on `e≡0` and `e-1≡1` *in that order*, Agda will follow the same line of reasoning as we did, and determine that the second pattern match is necessarily absurd:

```
¬∸-identity¹ : ¬ Σ ℕ (λ e → (x : ℕ) → e ∸ x ≡ x)
¬∸-identity¹ (e , e-is-id)
  with e-is-id 0 | e-is-id 1
... | refl | ()
```

6.7 Decidability

It is now time to return to the question of how does one use dependent types in the "real world." That is, the proofs we've seen so far work fine and dandy when everything is known statically, but things begin to fall apart when you want to get input from a user, read from a file, or do anything of a dynamic nature. What can we do when we don't have all of our necessary information known at compile time?

The answer is unsurprising: just compute it!

Recall that every concrete proof we've given is represented by a value. That means it has a representation in memory, and therefore, that it's the sort of thing we can build at runtime. The types obscure the techniques that you already know how to do from a career of

computing things with elbow grease, but everything you know still works.

In a language with a less powerful system, how do you check if the number the user input is less than five? You just do a runtime check to see if it's less than five before going on. We can do exactly the same thing in Agda, except that we'd like to return a *proof* that our number is equal to five, rather than just a boolean. As a first attempt, you can imagine we could implement such a thing like this:

```
⇤ 0 module Sandbox-Decidability where
      open Chapter2-Numbers
        using (Maybe; just; nothing)

      n=5? : (n : ℕ) → Maybe (n ≡ 5)
      n=5? 5 = just refl
      n=5? _ = nothing
```

Given `n=5?`, we can call this function and branch on its result. If the result is a `just`, we now have a proof that `n` was indeed 5, and can use it as we'd please. Of course, if the argument *wasn't* five, this definition doesn't allow us to learn anything at all—all we get back is `nothing`!

Instead, it would be much more useful to get back a proof that `¬ (n ≡ 5)`, in case we'd like to do something with that information. From this little argument, we conclude that returning `Maybe` isn't quite the right choice. More preferable would be a type with slightly more structure, corresponding to a *decision* about whether `n ≡ 5`:

```
2 data Dec {ℓ : Level} (P : Set ℓ) : Set ℓ where
    yes :   P → Dec P
    no  : ¬ P → Dec P
```

The type `Dec P` state that either `P` definitely holds, or that it definitely *doesn't* hold. Of course, only one of these can ever be true at once, and thus `Dec P` corresponds to an answer that we know one way or the other.

As an anti-example, given two numbers, it's not too hard to determine if the two are equal:

```
⇤ 0 open import Relation.Nullary using (Dec; yes; no)

    module Nat-Properties where
      _==_ : ℕ → ℕ → Bool
```

```
  zero  == zero  = true
  zero  == suc y = false
  suc x == zero  = false
  suc x == suc y = x == y
```

While `_==_` *is* a decision procedure, it doesn't give us back any proof-relevant term. We'd like instead to get back a proof that the two numbers were or were not equal. A better name for such a function is `_≟_` (input as `\?=`).

```
_≟_ : (x y : ℕ) → Dec (x ≡ y)
_≟_ = ?
```

The goal is slightly modify the definition of `_==_` such that whenever it returns `true` we instead get back a `yes`, and likewise replace `false` with `no`. Giving back the `yes`es is easy enough, but the `no`s take a little more thought:

```
_≟_ : (x y : ℕ) → Dec (x ≡ y)
zero  ≟ zero  = yes refl
zero  ≟ suc y = no ?
suc x ≟ zero  = no ?
suc x ≟ suc y with x ≟ y
... | yes refl = yes refl
... | no x≢y   = no ?
```

The first hole here has type `zero ≡ suc y → ⊥`, which we can `Refine` (`C-c C-r` in Emacs and VS Code) to a lambda:

```
_≟_ : (x y : ℕ) → Dec (x ≡ y)
zero  ≟ zero  = yes refl
zero  ≟ suc y = no λ { x → {! !} }
suc x ≟ zero  = no ?
suc x ≟ suc y with x ≟ y
... | yes refl = yes refl
... | no x≢y   = no ?
```

Inside our lambda we have a term `x` whose type is `zero ≡ suc y`. We know this can never happen, since `zero` and `suc` are different constructors. Therefore, we can solve this (and the next) hole with absurd pattern matches inside of the lambda:

```
2 _≟_ : (x y : ℕ) → Dec (x ≡ y)
  zero  ≟ zero  = yes refl
  zero  ≟ suc y = no λ ()
  suc x ≟ zero  = no λ ()
  suc x ≟ suc y with x ≟ y
  ... | yes refl = yes refl
  ... | no x≢y   = no ?
```

We are left with only one hole, but it has type `suc x ≡ suc y → ⊥`, and thus our absurd pattern trick can't work here. However, we do have a proof that `x ≢ y`, from which we must derive a contradiction. The idea is that if refine our hole to a lambda, it will have a parameter of type `suc x ≡ suc y`, which if we pattern match on, Agda will learn that `x ≡ y`. From there, we can invoke the fact that `x ≢ y`, and we have the contradiction we've been looking for. It's a bit of a brain buster, but as usual, the types lead the way:

```
2 _≟_ : (x y : ℕ) → Dec (x ≡ y)
  zero  ≟ zero  = yes refl
  zero  ≟ suc y = no λ ()
  suc x ≟ zero  = no λ ()
  suc x ≟ suc y with x ≟ y
  ... | yes refl = yes refl
  ... | no x≢y   = no λ { refl → x≢y refl }
```

Take a moment to reflect on this. Where before `_==_` simply returned `false`, we are now responsible for *deriving a contradiction.* Alternatively said, we must now *reify* our reasoning, and *prove* that our algorithm does what it says. The advantage of returning a proof of the negation is that downstream callers can use it to show their own impossible code-paths.

We can package up decidable equality into its own type:

```
2 DecidableEquality : {ℓ : Level} (A : Set ℓ) → Set ℓ
  DecidableEquality A = (x y : A) → Dec (x ≡ y)
```

and give a more "semantically-inclined" type to our old function:

```
2 _≟ℕ_ : DecidableEquality ℕ
  _≟ℕ_ = _≟_
```

6.8 Transforming Decisions

Often times, you will be trying to decide a problem, but can instead only directly decide some closely-related problem. In cases like these, it can be helpful to have some machinery for transforming one decision into another.

Doing so is trickier than you might expect; at first blush, it seems like if we have some function `P → Q` we should be able to write a related function `Dec P → Dec Q`. But alas we cannot! Doing so also requires a function `Q → P`, because in the `no` case, we also need to transform `Q` back into a `P` in order to refute it!

We can wrap up this logic once and for all as `map-dec`:

```
map-dec : {ℓ₁ ℓ₂ : Level} {P : Set ℓ₁} {Q : Set ℓ₂}
  → (P → Q) → (Q → P)
  → Dec P   → Dec Q
map-dec to from (yes p)  = yes (to p)
map-dec to from (no ¬p) = no  (λ q → ¬p (from q))
```

which will come in handy in the future.

6.9 Binary Trees

Perhaps you are tired of always working with numbers. After all, isn't this supposed to be a book about applying mathematical techniques to computer science? As it happens, all the techniques we have explored so far are applicable to data structures just as much as they are to numbers and the more mathematical purviews.

It is a matter of tradition to begin every exploration of data structures in functional programming with lists. I'm personally exhausted of this tradition, and suspect you are too. Therefore, we will instead touch upon the binary trees: how to construct them, and how to prove things about them.

A binary tree is either `empty`, or it is a `branch`ing node containing a value and two subtrees. We can codify this as follows:

```
open import Relation.Binary using (DecidableEquality)

module BinaryTrees where
  data BinTree {ℓ : Level} (A : Set ℓ) : Set ℓ where
    empty  : BinTree A
    branch : BinTree A → A → BinTree A → BinTree A
```

To illustrate how this type works, we can build a binary tree as follows:

```
tree : BinTree ℕ
tree =
  branch
    (branch (branch empty 0 empty) 0 (branch empty 2 empty))
    4
    (branch empty 6 empty)
```

corresponding to this tree:

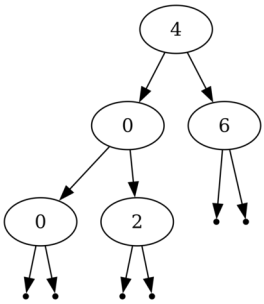


Figure 6.1: `tree`

where the little points are each an instance of the `empty` constructor.

For convenience, it's often helpful to be able to talk about a single-element tree, which we can encode as a `branch` with two `empty` children. This is another good use-case for a `pattern` synonym. The following defines a new pattern called `leaf : A → BinTree A`:

```
pattern leaf a = branch empty a empty
```

Having written `leaf`, we're now able to pattern match on singleton trees, as in:

```
is-singleton : {A : Set} → BinTree A → Bool
is-singleton (leaf _) = true
is-singleton _        = false
```

In addition, we can use patterns as expressions, as in:

```
five-tree : BinTree ℕ
five-tree = leaf 5
```

It's now possible to write `tree` more succinctly:

```
tree : BinTree ℕ
tree =
  branch
    (branch (leaf 0) 0 (leaf 4))
    2
    (leaf 6)
```

6.10 Proving Things about Binary Trees

With a suitable definition under our belt, it's time to suit up and enter the proof mines. The first thing we might want to do with a binary tree is determine whether it contains a particular element. Note that our `BinTree` type doesn't necessarily represent binary *search* trees (BSTs.)

We can show an element is in a `BinTree` inductively by looking at three cases:

1. the thing we're looking for is at the root of the tree, or
2. it's in the left subtree, or
3. it's in the right subtree.

Of course, in cases 2 and 3 we will work inductively, eventually finding a base case 1. We can encode these three cases directly in the type `_∈_` (input as `\in`):

```
data _∈_ {ℓ : Level} {A : Set ℓ} : A → BinTree A → Set ℓ where
  here  : {a : A}   {l r : BinTree A}        → a ∈ branch l a r
  left  : {a b : A} {l r : BinTree A} → a ∈ l → a ∈ branch l b r
  right : {a b : A} {l r : BinTree A} → a ∈ r → a ∈ branch l b r
```

This definition works perfectly well, but it's rather wordy. Notice that *over half* of it is just bringing implicit bindings into scope. There's nothing wrong with it, but it does somewhat obscure exactly what's going on.

This is a perfect use-case for Agda's `variable` block—already seen above when we were discussing `Level`s—which allows us to define implicit bindings that should exist for the remainder of the scope.

Variable blocks are started with the keywords `private variable`, and then begin a new layout. We can create a few variables:

```
⇤ 2 private variable
      ℓ : Level
      A : Set ℓ
      a b : A
      l r : BinTree A
```

Variable bindings can depend on one another. Here we've introduced a `Level` called `ℓ`, a type `A` of that level, two variables `a` and `b` of that type, and two binary trees called `l` and `r`.

Having put these variables into scope, we can rewrite `_∈_` in a style that better highlights the three cases:

```
⇤ 2 data _∈_ : A → BinTree A → Set where
      here  :         a ∈ branch l a r
      left  : a ∈ l → a ∈ branch l b r
      right : a ∈ r → a ∈ branch l b r
```

Much, much tidier.

As a demonstration of the `_∈_` type, we can give a proof that 6 is indeed in `tree`. Six is the root of the right-subtree, which makes our proof delightfully declarative:

```
⇤ 2 6∈tree : 6 ∈ tree
    6∈tree = right here
```

As usual, see what happens if you give the wrong proof, or change 6 in the type. Agda will correctly yell at you, indicating we've done the right thing here.

6.11 Decidability of Tree Membership

We've given decidability proofs for equality; can we also give one for `_∈_`? Certainly we can: given a decidable procedure for `A`, we can just try it at every node in the tree and see what shakes out.

But before writing it, however, the following definitions will be useful to help corral the types:

```
  2 Decidable : {c ℓ : Level} {A : Set c}
                → (A → Set ℓ) → Set (c ⊔ ℓ)
    Decidable {A = A} P = (a : A) → Dec (P a)

    Decidable₂ : {c ℓ : Level} {A : Set c}
```

```
                → (A → A → Set ℓ) → Set (c ⊔ ℓ)
Decidable₂ {A = A} _~_ = (x y : A) → Dec (x ~ y)
```

☢ CAUTION

Notice that we needed to give local definitions here for `ℓ` and `A`, rather than just use our `variable`s like before. This is a limitation in how Agda sorts out variables; written with `A` and `ℓ` bound to the `variable`s, Agda for some reason considers the `ℓ` in our type to be *different* than the `ℓ` in `A : Set ℓ`!

The `Dec` type corresponds to a particular *decision*, while `Decidable` states that we can make a decision for *every input*. It's the same difference between saying that you, personally, have an age, and saying that *everyone* has an age.

Implementing `∈?` isn't hard, but has a lot of moving pieces, so let's work it through together. Begin with the type:

```
2 ∈? : DecidableEquality A → (t : BinTree A) → Decidable (_∈ t)
  ∈? = ?
```

What we're saying here is that if we have decidable equality for some type `A`, we can then give a decision procedure to check whether any element is in any tree. Expanding out the arguments helps to see this:

```
2 ∈? : DecidableEquality A → (t : BinTree A) → Decidable (_∈ t)
  ∈? _≟_ t a = ?
```

As usual, we can `MakeCase` (`C-c C-c` in Emacs and VS Code) on the only value with an inductive data type, that is, `t`:

```
2 ∈? : DecidableEquality A → (t : BinTree A) → Decidable (_∈ t)
  ∈? _≟_ empty a = {! !}
  ∈? _≟_ (branch l x r) a = {! !}
```

The `empty` case is easy, since we know there are no values in `empty`, and thus that the answer must be `no`.

```
2 ∈? : DecidableEquality A → (t : BinTree A) → Decidable (_∈ t)
  ∈? _≟_ empty a = no λ ()
```

```
∈? _≟_ (branch l x r) a = {! !}
```

Checking out the `branch` case requires us to determine whether `x` is equal to `a`, which we can do by invoking `x ≟ y` in a `with` abstraction.

```
2 ∈? : DecidableEquality A → (t : BinTree A) → Decidable (_∈ t)
  ∈? _≟_ empty a = no λ ()
  ∈? _≟_ (branch l x r) a
    with x ≟ a
  ... | x≡a? = {! !}
```

Pattern match on `x≡a?`—if the answer is `yes`, then pattern matching on the subsequent proof that `a ≡ x` will allow us to use the `here` constructor to fill the hole:

```
2 ∈? : DecidableEquality A → (t : BinTree A) → Decidable (_∈ t)
  ∈? _≟_ empty a = no λ ()
  ∈? _≟_ (branch l x r) a
    with x ≟ a
  ... | yes refl = yes here
  ... | no  x≢a  = {! !}
```

If `x` isn't what we were looking for, we can instead try to look down the left subtree and see if we get any hits. Change the `with` abstraction to add another binding:

```
2 ∈? : DecidableEquality A → (t : BinTree A) → Decidable (_∈ t)
  ∈? _≟_ empty a = no λ ()
  ∈? _≟_ (branch l x r) a
    with x ≟ a     | ∈? _≟_ l a
  ... | yes refl | _     = yes here
  ... | no  x≢a  | x∈l? = {! !}
```

We can play the same game here: pattern matching on `x∈l?` and rebuilding a `yes` if we were successful. Before you ask, you can type `∉` via `\inn`.

```
2 ∈? : DecidableEquality A → (t : BinTree A) → Decidable (_∈ t)
  ∈? _≟_ empty a = no λ ()
  ∈? _≟_ (branch l x r) a
    with x ≟ a     | ∈? _≟_ l a
  ... | yes refl | _       = yes here
```

```
... | no  _    | yes x∈l = yes (left x∈l)
... | no  x≠a  | no  x∉l = {! !}
```

If we didn't find what we were looking for, well, just try again in the right branch:

```
2 ∈? : DecidableEquality A → (t : BinTree A) → Decidable (_∈ t)
∈? _≟_ empty a = no λ ()
∈? _≟_ (branch l x r) a
  with x ≟ a    | ∈? _≟_ l a | ∈? _≟_ r a
... | yes refl | _          | _       = yes here
... | no  _    | yes a∈l    | _       = yes (left a∈l)
... | no  _    | no  _      | yes a∈r = yes (right a∈r)
... | no  x≠a  | no  a∉l    | no  a∉r = {! !}
```

Should the answer to *all* of our queries be `no`, we can be certain that `¬ a ∈ t`. All that's left is to say `no`, and then refute whichever piece of evidence we've got:

```
2 ∈? : DecidableEquality A → (t : BinTree A) → Decidable (_∈ t)
∈? _≟_ empty a = no λ ()
∈? _≟_ (branch l x r) a
  with x ≟ a    | ∈? _≟_ l a | ∈? _≟_ r a
... | yes refl | _          | _       = yes here
... | no  _    | yes a∈l    | _       = yes (left a∈l)
... | no  _    | no  _      | yes a∈r = yes (right a∈r)
... | no  x≠a  | no  a∉l    | no  a∉r
    = no λ { here       → x≠a refl
           ; (left  a∈l) → a∉l a∈l
           ; (right a∈r) → a∉r a∈r
           }
```

The resulting code compiles just fine, which, given the strength of our type system, means it *works* just fine. We can convince ourselves by asking Agda, leaving underscores in the `Dec` constructors—in essence, leaving Agda to fill in the underscores but confirm that we picked the right constructors:

```
⇤ 2 open Nat-Properties
      using (_≟ℕ_)

    _ : ∈? _≟ℕ_ tree 2 ≡ yes _
```

```
_ = refl

_ : ∈? _≟ℕ_ tree 7 ≡ no _
_ = refl
```

It's worth comparing and contrasting the implementation of `∈?` to that of `_≟ℕ_`, as this is the form that *every* decision problem takes—at least those concerned with the shape of a particular value.

6.12 The All Predicate

Something else we might want to be able to prove is that every element in a `BinTree` satisfies some property `P`—perhaps that it consists only of odd numbers, as an odd example. For a more enterprise sort of example, perhaps we'd like a proof that every employee in our tree has been properly connected to payroll.

Building `All` is easy enough. We can replace every instance of `A` with `P A`, and every instance of `BinTree` with `All`. Modulo a few indices at the end, we're left with a reminiscent definition:

```
data All {ℓ₁ ℓ₂ : Level} {A : Set ℓ₁} (P : A → Set ℓ₂)
       : BinTree A → Set (ℓ₁ ⊔ ℓ₂) where
  empty  : All P empty
  branch : All P l → P a → All P r → All P (branch l a r)
```

In the `branch` case, we must show that `P` holds for everything in both subtrees, as well as for the value at the root. By induction, we have thus covered every element in the tree.

Like we did above for `BinTree`, it will be convenient to make a `leaf` pattern here too. Because pattern synonyms don't come with a type signature and constructor names can be reused, pattern synonyms work over any type that has all of the necessary constructors. However, this is only the case if those constructors existed before the pattern was defined.

Since `All` reuses the constructor names `empty` and `branch`, we need only redefine `leaf` in order for it to work not only over `BinTree`, but `All` too.

```
pattern leaf a = branch empty a empty
```

We can show that `All` works as expected, by coming up with a quick predicate like the evenness of a number. Let's bring the machinery back into scope:

```
open Chapter2-Numbers
  using (IsEven; z-even; ss-even)
```

and now we would like to show that every element in `tree` is even:

```
tree-all-even : All IsEven tree
tree-all-even = ?
```

Thankfully, Agda is capable of proving this fact on our behalf, via `Auto` (`C-c C-a` in Emacs and VS Code):

```
tree-all-even : All IsEven tree
tree-all-even =
  branch
    (branch
      (branch empty z-even empty)
      z-even
      (branch empty (ss-even z-even) empty))
    (ss-even (ss-even z-even))
    (branch empty (ss-even (ss-even (ss-even z-even))) empty)
```

Notice the repeated use of `branch empty ? empty` here. Sometimes Agda can work out that it should use your pattern synonyms, and sometimes it can't. Unfortunately this is one of the times it can't, but we can clean it up ourselves by judiciously invoking `leaf`:

```
tree-all-even : All IsEven tree
tree-all-even =
  branch
    (branch
      (leaf z-even)
      z-even
      (leaf (ss-even z-even)))
    (ss-even (ss-even z-even))
    (leaf (ss-even (ss-even (ss-even z-even))))
```

Exercise (Medium) Show that `All P` is `Decidable`, given `Decidable P`.

Solution

↤ 2
```
all? : {P : A → Set} → Decidable P → Decidable (All P)
all? p? empty = yes empty
all? p? (branch l a r) with p? a | all? p? l | all? p? r
... | no ¬pa | _      | _      = no λ { (branch _ pa _)
                                       → ¬pa pa }
... | yes _  | no ¬al | _      = no λ { (branch al _ _)
                                       → ¬al al }
... | yes _  | yes _  | no ¬ar = no λ { (branch _ _ ar)
                                       → ¬ar ar }
... | yes pa | yes al | yes ar = yes (branch al pa ar)
```

As this exercise shows, decision procedures are often extremely formulaic. You decide each individual piece, and combine them together if possible. If not, you must find a way to project a contradiction out of the bigger structure.

6.13 Binary Search Trees

Binary search trees (BSTs) are a refinement of binary trees, with the property that everything in the left subtree is less than or equal to the root, and everything in the right subtree is greater than or equal to the root. This will be our next challenge to define in Agda.

You might be starting to see a pattern here. We always just look at the individual cases, figure out what we need for each, and then build an indexed type that ensures the properties are all met. In this case, we know that an `empty BinTree` is still a BST, so that's easy enough. In the case of a `branch`, however, we have several properties we'd like to check:

1. Everything in the left subtree is less than the root, and
2. everything in the right tree is greater than the root, furthermore
3. the left subtree is itself a BST, and finally
4. the right subtree is also a BST.

Having worked out the details, the type itself has a straightforward encoding, after realizing that we'd like to parameterize the whole thing by an ordering relation.

```
data IsBST {ℓ₁ ℓ₂ : Level} {A : Set ℓ₁} (_<_ : A → A → Set ℓ₂)
  : BinTree A → Set (ℓ₁ ⊔ ℓ₂) where
  bst-empty : IsBST _<_ empty
  bst-branch
    : All (_< a) l
    → All (a <_) r
    → IsBST _<_ l
    → IsBST _<_ r
    → IsBST _<_ (branch l a r)
```

Given the standard notion of ordering over the natural numbers:

```
open Chapter4-Relations
  using (_≤_; z≤n; s≤s; _<_)
```

we can again ask Agda for a proof that `tree` is a BST under the `_≤_` ordering relation, by invoking `Auto` (`C-c C-a` in Emacs and VS Code) to give us a definition of `tree-is-bst`:

```
tree-is-bst : IsBST _≤_ tree
tree-is-bst =
  bst-branch
    (branch (leaf z≤n) z≤n (leaf (s≤s (s≤s z≤n))))
    (leaf (s≤s (s≤s (s≤s (s≤s z≤n)))))
    (bst-branch
      (leaf z≤n)
      (leaf z≤n)
      (bst-branch empty empty bst-empty bst-empty)
      (bst-branch empty empty bst-empty bst-empty))
    (bst-branch empty empty bst-empty bst-empty)
```

Proofs like this are an awful lot of work, but thankfully we never need to write them by hand. For concrete values, we can always ask Agda to solve them for us, and for parameterized values, we can build them by induction.

Speaking of induction, can we decide whether a given tree is a BST? Sure! The pattern is exactly the same, decide each piece, derive contradictions on `no`, and assemble the final proof if everything is `yes`. This time, rather than doing all of our `with` abstraction at once, we will test one property at a time—leading to a more "vertical" definition:

```
⇤ 2 is-bst?
      : {_≤_ : A → A → Set}
      → Decidable₂ _≤_
      → Decidable (IsBST _≤_)
    is-bst? _≤?_ empty = yes bst-empty
    is-bst? _≤?_ (branch l a r)
      with all? (_≤? a) l
    ... | no  l≰ = no λ { (bst-branch l≤ _ _ _) → l≰ l≤ }
    ... | yes l≤
      with all? (a ≤?_) r
    ... | no  ≰r = no λ { (bst-branch _ ≤r _ _) → ≰r ≤r }
    ... | yes ≤r
      with is-bst? _≤?_ l
    ... | no  ¬bst-l = no λ { (bst-branch _ _ bst-l _)
                               → ¬bst-l bst-l }
    ... | yes bst-l
      with is-bst? _≤?_ r
    ... | no  ¬bst-r = no λ { (bst-branch _ _ _ bst-r)
                               → ¬bst-r bst-r }
    ... | yes bst-r  = yes (bst-branch l≤ ≤r bst-l bst-r)
```

You might be wondering whether there is an easier way to assemble these proofs. Unfortunately, there is not. Nothing in the theory precludes us from generating them, but at time of writing, there is alas no automatic means of deriving such.

6.14 Trichotomy

What we have done thus far is show that there exist things called binary search trees, and we have given a definition of what properties we mean when we say something is or isn't a binary search tree. This is a great start—prescriptively speaking—but this is not exactly where most computer science textbooks go when they discuss BSTs. Instead, they immediately dive into the meat of "what can you do with a BST."

Insertion is something you can do with a BST, and lest we be left behind by the traditional pedagogical material, let's turn our discussion in that direction. The algorithm is easy enough—if the tree is `empty`, add a `leaf`. Otherwise, compare the value you'd like to insert with the value at the root. If they're equal, you're done. Otherwise, recursively insert the value into the correct subtrees.

The implicit claim here is that this algorithm preserves the `IsBST` invariant, but that is never explicitly stated. For the record, this algorithm *does* indeed preserve the `IsBST` invariant. However, this poses some challenges for us, since in order to show this we must necessarily derive a proof, which is going to recursively depend on a proof that inserted into the correct subtree.

What we have to work with thus far is only the fact that `_<_` is decidable. But if we were to work directly with the decidability `_<_`, our algorithm would need to first check whether `a < x`, and if it isn't, check that `x < a`, and if it isn't check that `x ≡ a`, and if *that also isn't,* well, then we definitely have a problem.

However, it should be the case that exactly one of `a < x`, `x < a`, and `x ≡ a` should be true for any `x` and `a` you could possibly desire. We could attempt to prove this directly, but it doesn't sound like an enjoyable experience. As we have seen above, every invocation of decidability doubles the amount of work we need to do, since we need to use the proof or show a subsequent contradiction.

Instead, we can generalize the idea of decidability to a *trichotomy*, which is the idea that exactly one of three choices must hold. From this perspective, `Dec` is merely a type that states exactly one of `P` or `¬ P` holds, and so the notion of trichotomy shouldn't be earth-shattering. We can define `Tri` (analogous to `Dec`) as proof that exactly one of `A`, `B` or `C` holds:

```
2 data Tri {a b c : Level} (A : Set a) (B : Set b) (C : Set c)
         : Set (a ⊔ b ⊔ c) where
    tri< :   A → ¬ B → ¬ C → Tri A B C
    tri≈ : ¬ A →   B → ¬ C → Tri A B C
    tri> : ¬ A → ¬ B →   C → Tri A B C
```

Just as we defined $\texttt{Decidable}_2$, we will now lift this notion of `Tri` to an analogous `Trichotomous`: stating that for two relations, one equality-like, and one less-than-like, we can always determine which of the three options holds.

```
⇤ 2 Trichotomous
      : {ℓ eq lt : Level}
      → {A : Set ℓ}
      → (_≈_ : A → A → Set eq)
      → (_<_ : A → A → Set lt)
      → Set (lt ⊔ eq ⊔ ℓ)
    Trichotomous {A = A} _≈_ _<_ =
```

```
  (x y : A) → Tri (x < y) (x ≈ y) (y < x)
```

As a good exercise, we'd like to show that ℕ is trichotomous with respect to _≡_ and _<_. Doing so will require a quick lemma. If we know $x \not< y$ then we also know $x + 1 \not< y + 1$:

```
refute : {x y : ℕ} → ¬ x < y → ¬ suc x < suc y
refute x≮y (s≤s x<y) = x≮y x<y
```

Given these two lemmas, its not too much work to bash out <-cmp, the traditional name for the trichotomy of the natural numbers. The hardest part is simply massaging all six of the refutations in the recursive case:

```
<-cmp : Trichotomous _≡_ _<_
<-cmp zero zero = tri≈ (λ ()) refl (λ ())
<-cmp zero (suc y) = tri< (s≤s z≤n) (λ ()) (λ ())
<-cmp (suc x) zero = tri> (λ ()) (λ ()) (s≤s z≤n)
<-cmp (suc x) (suc y) with <-cmp x y
... | tri< x<y x≢y x≯y
    = tri< (s≤s x<y)
           (λ { sx≈sy → x≢y (suc-injective sx≈sy) })
           (refute x≯y)
... | tri≈ x≮y x≈y x≯y
    = tri≈ (refute x≮y)
           (cong suc x≈y)
           (refute x≯y)
... | tri> x≮y x≢y x>y
    = tri> (refute x≮y)
           (λ { sx≈sy → x≢y (suc-injective sx≈sy) })
           (s≤s x>y)
```

Working directly with Tri, rather than several invocations to Decidable₂ _<_ will dramatically reduce the proof effort necessary to define insert and show that it preserves IsBST, which we will do in the next section.

6.15 Insertion into BSTs

We are going to require a _<_ relation, and a proof that it forms a trichotomy with _≡_ in order to work through the implementation

details here. Rather than pollute all of our type signatures with the necessary plumbing, we can instead define an anonymous module and add the common parameters to that instead, as in:

```
⇤ 2 module _
      {ℓ : Level}
      {_<_ : A → A → Set ℓ}
      (<-cmp : Trichotomous _≡_ _<_) where
```

Anything defined in this module will now automatically inherit `_<_` and `Trichotomous _≡_ _<_` arguments, meaning we don't need to pass the (locally-bound) `<-cmp` function around by hand.

Defining `insert` follows exactly the same template as our in-writing algorithm above—recall, the plan is to replace `empty` with `leaf`, and otherwise recurse down the correct subtree in the case of `branch`. After everything we've been doing lately, `insert` turns out to be a walk in the park:

```
⇤ 4 insert
      : A
      → BinTree A
      → BinTree A
    insert a empty = leaf a
    insert a (branch l x r) with <-cmp a x
    ... | tri< _ _ _ = branch (insert a l) x r
    ... | tri≈ _ _ _ = branch l x r
    ... | tri> _ _ _ = branch l x (insert a r)
```

We would now like to show that `insert` preserves the `IsBST` invariant. That is, we'd like to define the following function:

```
4 bst-insert
      : (a : A)
      → {t : BinTree A}
      → IsBST _<_ t
      → IsBST _<_ (insert a t)
    bst-insert = ?
```

Before diving into this proper, we will do a little thinking ahead and realize that showing `IsBST` for a `branch` constructor requires showing that all elements in either subtree are properly bounded by the root. Therefore, before we show that `insert` preserves `IsBST`, we must first prove that `insert` preserves `All`! The type we need to show is that if `P a` and `All P t` hold (for any `P`), then so too must `All P (insert a t)`:

```
all-insert
  : {P : A → Set ℓ} → (a : A) → P a → {t : BinTree A}
  → All P t → All P (insert a t)
all-insert = ?
```

After binding our variables (including `t`, carefully positioned so that we can avoid binding `P`), our function now looks like:

```
all-insert
  : {P : A → Set ℓ} → (a : A) → P a → {t : BinTree A}
  → All P t → All P (insert a t)
all-insert a pa {t} all-pt = {! !}
```

Asking now for a `MakeCase` with argument `all-pt` (`C-c C-c` in Emacs and VS Code) results in:

```
all-insert
  : {P : A → Set ℓ} → (a : A) → P a → {t : BinTree A}
  → All P t → All P (insert a t)
all-insert a pa {.empty} empty = {! !}
all-insert a pa {.(branch _ _ _)} (branch l<x px x<r) = {! !}
```

Notice that when we pattern matched on `all-pt`, Agda realized that this fully determines the results of `t` as well, and it happily expanded them out into these funny .``ctor:empty .``ctor:branch patterns. These are called *dot patterns,* and they correspond to patterns whose form has been fully determined by pattern matching on something else. We will discuss dot patterns more thoroughly in sec. 4.10, but for the time being, it suffices to know that, whenever we have a dot pattern, we can simply replace it with a regular pattern match:

```
all-insert
  : {P : A → Set ℓ} → (a : A) → P a → {t : BinTree A}
  → All P t → All P (insert a t)
all-insert a pa {empty} empty = {! !}
all-insert a pa {branch l x r} (branch l<x xp x<r) = {! !}
```

Finally, there is no reason to pattern match on the implicit `empty`, since it doesn't bring any new variables into scope.

```
4 all-insert
    : {P : A → Set ℓ} → (a : A) → P a → {t : BinTree A}
    → All P t → All P (insert a t)
  all-insert a pa empty = {! !}
  all-insert a pa {branch l x r} (branch l<x px x<r) = {! !}
```

Filling the first hole is merely showing that we have `All` for a singleton, which is our `leaf` constructor:

```
4 all-insert
    : {P : A → Set ℓ} → (a : A) → P a → {t : BinTree A}
    → All P t → All P (insert a t)
  all-insert a pa empty = leaf pa
  all-insert a pa {branch l x r} (branch l<x px x<r) = {! !}
```

Attempting to `Refine` (`C-c C-r` in Emacs and VS Code) this last hole results in a funny thing. We know it must be filled with a `branch`, but Agda will refuses.If you ask for the type of the goal, we see something peculiar:

INFO WINDOW

```
Goal: All P (insert <-cmp a (branch l x r) | <-cmp a x)
```

The vertical bar is not anything that you're allowed to write for yourself, but the meaning here is that the result of `insert` is stuck until Agda knows the result of `<-cmp a`. Our only means of unsticking it is to also do a `with` abstraction over `<-cmp a x` and subsequently pattern match on the result:

```
4 all-insert
    : {P : A → Set ℓ} → (a : A) → P a → {t : BinTree A}
    → All P t → All P (insert a t)
  all-insert a pa empty = leaf pa
  all-insert a pa {branch l x r} (branch l<x px x<r)
    with <-cmp a x
  ... | tri< a<x _ _ = {! !}
  ... | tri≈ _ a=x _ = {! !}
  ... | tri> _ _ x<a = {! !}
```

Looking at this new set of goals, we see that they are no longer stuck and have now computed to things we're capable of implementing. By now you should be able to complete the function on your own:

```
all-insert
  : {P : A → Set ℓ} → (a : A) → P a → {t : BinTree A}
  → All P t → All P (insert a t)
all-insert a pa empty = leaf pa
all-insert a pa {branch l x r} (branch l<x px x<r)
  with <-cmp a x
... | tri< a<x _ _ = branch (all-insert a pa l<x) px x<r
... | tri≈ _ a=x _ = branch l<x px x<r
... | tri> _ _ x<a = branch l<x px (all-insert a pa x<r)
```

Now that we've finished the `all-insert` lemma, we're ready to pop the stack and show that `insert` preserves `IsBST`.

Again, when implementing this you will see that the type will get stuck on `insert x t | <-cmp a x`, and will require another pattern match on `<-cmp a x`. This will always be the case when proving things that pattern match on a `with` abstraction; it is for this reason that we say the proof has the same shape as the computation. It's like poetry: it rhymes.

Implementing `bst-insert` isn't any more of a cognitive challenge than `all-insert` was; just pattern match and then build a proof term via induction:

```
bst-insert
  : (a : A)
  → {t : BinTree A}
  → IsBST _<_ t
  → IsBST _<_ (insert a t)
bst-insert a bst-empty
  = bst-branch empty empty bst-empty bst-empty
bst-insert a {branch l x r} (bst-branch l<x x<r lbst rbst)
  with <-cmp a x
... | tri< a<x _ _ =
        bst-branch
          (all-insert a a<x l<x)
          x<r
          (bst-insert a lbst)
          rbst
... | tri≈ _ a=x _ = bst-branch l<x x<r lbst rbst
... | tri> _ _ x<a =
        bst-branch
          l<x
          (all-insert a x<a x<r)
```

```
        lbst
        (bst-insert a rbst)
```

With `bst-insert` finally implemented, we have now proven that inserting into a binary search tree results in another binary search tree. It's a bit of an underwhelming result for all the work we've done, isn't it? In the next section, we will look at alternative ways of phrasing the problem that can help.

6.16 Intrinsic vs Extrinsic Proofs

This style of proof we have demonstrated, is called an *extrinsic* proof. The idea here is that we have defined a type `BinTree A` and an operation `insert : A → BinTree A → BinTree A` in such a way that they are not guaranteed to be correct. In order to subsequently prove that they are, we added an additional layer on top, namely `IsBST` and `bst-insert` which assert our invariants after the fact. This is a very natural way of proving things, and is a natural extension of the way most people write code: do the implementation first, and tack the tests on afterwards.

However, extrinsic proofs are not our only option! We can instead make a `BST A` type which satisfies the binary search tree invariant *by construction.* By virtue of this being by construction, we are not required to tack on any additional proof: the existence of the thing itself is proof enough. We call this notion of proof *intrinsic*, as the proof is intrinsic to the construction of the object itself. In a sense, intrinsic proofs don't exist at all; they're just cleverly-defined objects of study.

Intrinsic proofs are desirable because they only require you to do the work once. Recall that when defining both `bst-insert` and `all-insert`, we essentially had to mirror the definition of `insert` modulo a few changes. It was quite a lot of work, and you can imagine that this effort would multiply substantially for each additional operations we'd like to define over BSTs. Extrinsic proofs make you shoulder this burden all on your own.

That's not to say that intrinsic proofs are without downsides. At the forefront, almost every algorithm you have ever seen is given with an extrinsic proof—if it comes with a proof at all. Not only are we as computer scientists better primed for thinking about extrinsic proof, but worse: the field itself is riddled with them. Almost every algorithm you have ever heard of isn't amenable to intrinsic proof,

as most algorithms violate the intrinsic invariant at some point. The invariant of most structures is a macro-level property, preserved by common operations, but rarely preserved *inside* of them.

For example, consider a heap data structure, which is a particular implementation of a priority queue—at any time, you can extract the highest-priority thing in the queue. Heaps are usually implemented as binary trees (or sometimes, clever encodings of binary trees as arrays) with the *heap property:* the root node is larger than everything else in the tree. The heap property is also satisfied recursively, so that we have a partial ordering over all elements in the heap.

Now imagine we'd like to insert something new into the heap. We don't know where it should go, so we insert it into the first empty leaf we can find, and then recursively "bubble" it up. That is to say, you put it at a leaf, and then check to see if it's larger than its parent. If so, you swap the two, and then recurse upwards. Eventually the newly-insert element is somewhere in the tree such that the heap invariant is met.

This algorithm works just fine, but it *cannot* be used in an intrinsic heap, because when we first insert the new element into a bottommost leaf, the heap property is immediately broken! It doesn't matter if we eventually fix the invariant; the intrinsic construction of heaps means it's *impossible* to insert an element somewhere it doesn't belong, and thus the bubble algorithm cannot be used.

It is for reasons like these that intrinsic proofs are *hard* for computer scientists. Fully embracing them requires unlearning a great deal of what our discipline has taught us, and that is a notoriously difficult thing to do.

6.17 An Intrinsic BST

Constructing an intrinsic BST is not the most straightforward construction, but thankfully McBride (2014) has done the hard work for us and we will re-derive his solution here.

In order to define an intrinsic binary search tree, we will proceed in two steps. First, we will define a BST indexed by its upper and lower bounds, which we can subsequently use to ensure everything is in its right place, without resorting to extrinsic techniques like `All`. We will then hide these indices again to get a nice interface for the whole thing.

Begin with a new module to sandbox our first step:

```
⇤ 0 module Intrinsic-BST-Impl
      {c ℓ : Level} {A : Set c} (_<_ : A → A → Set ℓ) where
```

As before, we make a `BST` type, but this time parameterized by a `lo` and `hi` bound. In the `empty` constructor we require a proof that `lo < hi`.

```
⇤ 2 data BST (lo hi : A) : Set (c ⊔ ℓ) where
      empty : lo < hi → BST lo hi
```

Our other constructor, `branch`, now restricts the bounds of its left and right subtrees so their top and bottom bounds are the root, respectively:

```
  4 xbranch   ❷
      : (a : A)   ❶
      → BST lo a
      → BST a hi
      → BST lo hi
```

Notice at ❶ that the root of the tree comes as the first argument! This is unlike our extrinsic `branch`, but is necessary here so that `a` is in scope when we build our subtrees. The discrepancy in argument order is why this constructor has been named `xbranch`, as indicated at ❷.

Fortunately, we can use a pattern synonym to shuffle our parameters back into the correct shape, as well as one to bring back `leaf`s:

```
⇤ 2 pattern branch lo a hi = xbranch a lo hi
    pattern leaf lo<a a a<hi = branch (empty lo<a) a (empty a<hi)
```

Returning to the issue of `insert`, we notice one big problem with putting the bounds in the type index: it means that `insert` could *change the type* of the `BST` if it is outside the original bounds! This is a bad state of affairs, and will dramatically harm our desired ergonomics. For the time being, we will sidestep the issue and merely require proofs that `a` is already in bounds.

The implementation of `insert` is nearly identical to our original, extrinsically-proven version:

```
  2 open BinaryTrees using (Trichotomous; Tri)
    open Tri
```

```
insert
  : {lo hi : A}
  → (<-cmp : Trichotomous _≡_ _<_)
  → (a : A)
  → lo < a
  → a < hi
  → BST lo hi
  → BST lo hi
insert <-cmp a lo<a a<hi (empty _) = leaf lo<a a a<hi
insert <-cmp a lo<a a<hi (branch l x r)
  with <-cmp a x
... | tri< a<x _ _ = branch (insert <-cmp a lo<a a<x l) x r
... | tri≈ _ a=x _ = branch l x r
... | tri> _ _ x<a = branch l x (insert <-cmp a x<a a<hi r)
```

This concludes our first step of the problem. We now have an intrinsically-proven BST—all that remains is to deal with the type-changing problem, putting an ergonomic facade in front.

A cheeky solution to the problem of `insert` possibly changing the type is to bound all BSTs by the equivalent of negative and positive infinity. This is, in essence, throwing away the bounds—at least at the top level. If we can hide those details, we will simultaneously have solved the problem of changing types and the ergonomics of needing to juggle the bounds. Let's do that now.

The first step is to define a type which *extends* `A` with the notions of positive and negative infinity. We'll put this in a new module, because we'd like to instantiate its parameters differently than those of `Intrinsic-BST-Impl`:

```
⇤ 0 open BinaryTrees using (Trichotomous)

module Intrinsic-BST
         {c ℓ : Level} {A : Set c}
         {_<_ : A → A → Set ℓ}
         (<-cmp : Trichotomous _≡_ _<_) where

  data A↑ : Set c where
    -∞ +∞ : A↑
    ↑     : A → A↑
```

We can type the ↑ symbol here as `\u-`, and ∞ as `\inf`.

The ↑ constructor lifts an A into A↑, and it is through this mechanism by which we are justified in saying that A↑ extends A with -∞ and +∞.

From here, it's not too hard to define a _<_ relationship over ∞:

```
⇤ 2 data _<∞_ : A↑ → A↑ → Set ℓ where
      -∞<↑  : {x   : A}       → -∞ <∞ ↑ x
      ↑<↑   : {x y : A} → x < y → ↑ x <∞ ↑ y
      ↑<+∞  : {x   : A}       → ↑ x <∞ +∞
      -∞<+∞ : -∞ <∞ +∞
```

This has all the properties you'd expect, that -∞ is less than everything else, +∞ is more than everything else, and that we can lift _<_ over A. The trick should be clear: we are not going to instantiate the Intrinsic-BST-Impl module at type A, but rather at ∞!

Before we can do that, however, we must show that our new _<∞_ is trichotomous. This is quite a lot of work, but the details are uninteresting to us by now, and indeed Agda can do the first several cases for us automatically:

```
⇤ 2 open BinaryTrees using (Tri)
    open Tri

    <∞-cmp : Trichotomous _≡_ _<∞_
    <∞-cmp -∞    -∞    = tri≈ (λ ()) refl   (λ ())
    <∞-cmp -∞    +∞    = tri< -∞<+∞ (λ ()) (λ ())
    <∞-cmp -∞    (↑ _) = tri< -∞<↑  (λ ()) (λ ())
    <∞-cmp +∞    -∞    = tri> (λ ()) (λ ()) -∞<+∞
    <∞-cmp +∞    +∞    = tri≈ (λ ()) refl   (λ ())
    <∞-cmp +∞    (↑ _) = tri> (λ ()) (λ ()) ↑<+∞
    <∞-cmp (↑ _) -∞    = tri> (λ ()) (λ ()) -∞<↑
    <∞-cmp (↑ _) +∞    = tri< ↑<+∞  (λ ()) (λ ())
```

All that's left is lifting <-cmp, which we must do by hand:

```
  2 <∞-cmp (↑ x) (↑ y)
      with <-cmp x y
    ... | tri< x<y ¬x=y ¬y<x =
            tri<
              (↑<↑ x<y)
              (λ { refl → ¬x=y refl })
              (λ { (↑<↑ y<x) → ¬y<x y<x })
```

```
... | tri≈ ¬x<x refl _ =
        tri≈
          (λ { (↑<↑ x<x) → ¬x<x x<x })
          refl
          (λ { (↑<↑ x<x) → ¬x<x x<x })
... | tri> ¬x<y ¬x=y y<x =
        tri>
          (λ { (↑<↑ x<y) → ¬x<y x<y })
          (λ { refl → ¬x=y refl })
          (↑<↑ y<x)
```

The end is nigh! We can now define our final `BST` as one bounded by `-∞` and `+∞`:

```
open module Impl = Intrinsic-BST-Impl _<∞_
  hiding (BST; insert)

BST : Set (c ⊔ ℓ)
BST = Impl.BST -∞ +∞
```

and finally, define insertion by merely plugging in some trivial proofs:

```
insert : (a : A) → BST → BST
insert a t = Impl.insert <∞-cmp (↑ a) -∞<↑ ↑<+∞ t
```

6.18 Wrapping Up

Decidability turns out to be quite a big topic! The field of computer science tends to hold up decidability as the gold standard of things to care about, but here we are, having not mentioned it until chapter 6.

In sec. 6.1, we discussed negation, falseness, and the principle of explosion. The last two of these can be found in the standard library under `Data.Empty`:

```
open import Data.Empty
  using (⊥; ⊥-elim)
  public
```

while negation can be found alongside `Dec` (from sec. 6.7) under `Relation.Nullary`:

```
⇤ 0 open import Relation.Nullary
      using (Dec; yes; no; ¬_)
      public
```

Despite `Dec` and `¬_` allegedly being nullary relations, for some reason the type `Decidable` itself comes from `Relation.Unary`:

```
⇤ 0 open import Relation.Unary
      using (Decidable)
      public
```

In sec. 6.3 we discussed inequality, which can be found in the same place as propositional equality:

```
⇤ 0 open import Relation.Binary.PropositionalEquality
      using (_≢_; ≢-sym)
      public
```

Many of the properties we could show inequality *didn't respect* come instead from `Relation.Binary`:

```
⇤ 0 open import Relation.Binary
      using (Reflexive; Transitive; DecidableEquality)
      using (Trichotomous; Tri)
      renaming (Decidable to Decidable₂)
      public
```

Alongside all this, there are a few miscellaneous re-exports that we must give, like `map-dec` from sec. 6.8 is really called `map′` in `Relation.Nullary.Decidable`:

```
⇤ 0 open import Relation.Nullary.Decidable
      renaming (map′ to map-dec)
      public
```

and the fact that `_<_` is `Trichotomous` comes again from `Data.Nat.Properties`:

```
⇤ 0 open import Data.Nat.Properties
      using (<-cmp)
      public
```

And finally, even though they aren't in the standard library, we will see our old binary tree friends from sec. 6.9 again soon.

```
⇤ 0 open BinaryTrees
      using (BinTree; empty; branch; leaf)
      public
```

⌨ UNICODE IN THIS CHAPTER

```
¬ U+00AC NOT SIGN (\neg)
× U+00D7 MULTIPLICATION SIGN (\x)
ˡ U+02E1 MODIFIER LETTER SMALL L (\^l)
Σ U+03A3 GREEK CAPITAL LETTER SIGMA (\GS)
λ U+03BB GREEK SMALL LETTER LAMDA (\Gl)
′ U+2032 PRIME (\')
₁ U+2081 SUBSCRIPT ONE (\_1)
₂ U+2082 SUBSCRIPT TWO (\_2)
ℓ U+2113 SCRIPT SMALL L (\ell)
ℕ U+2115 DOUBLE-STRUCK CAPITAL N (\bN)
ℤ U+2124 DOUBLE-STRUCK CAPITAL Z (\bZ)
↑ U+2191 UPWARDS ARROW (\u-)
→ U+2192 RIGHTWARDS ARROW (\to)
∈ U+2208 ELEMENT OF (\in)
∉ U+2209 NOT AN ELEMENT OF (\inn)
∞ U+221E INFINITY (\inf)
∸ U+2238 DOT MINUS (\.-)
≈ U+2248 ALMOST EQUAL TO (\~~)
≉ U+2249 NOT ALMOST EQUAL TO (\~~n)
≟ U+225F QUESTIONED EQUAL TO (\?=)
≡ U+2261 IDENTICAL TO (\==)
≢ U+2262 NOT IDENTICAL TO (\==n)
≤ U+2264 LESS-THAN OR EQUAL TO (\le)
≮ U+226E NOT LESS-THAN (\<n)
≰ U+2270 NEITHER LESS-THAN NOR EQUAL TO (\len)
≱ U+2271 NEITHER GREATER-THAN NOR EQUAL TO (\gtn)
⊔ U+2294 SQUARE CUP (\lub)
⊥ U+22A5 UP TACK (\bot)
```

About the Author

Sandy Maguire is a man of many facets, whose passion for software correctness has led him to become the author of three influential books in the field. He firmly believes that a combination of solid theoretical foundations and the practical wisdom to apply them is paramount in the world of software development.

In addition to theoretical software, Sandy's interests are as diverse as they are unique. He's an avid car-hater who finds solace from the busy city as a talented pianist. His enthusiasm doesn't stop there, as he's also a dedicated parkour enthusiast, constantly seeking new heights and challenges.

Not content with the status quo, Sandy stepped away from the software industry in 2018 to actively pursue change. He's a self-described author, programmer, musician, and trouble-maker who's been immersed in the world of coding for over two decades. With considerable time and experience, he feels like he's finally making progress on where good software comes from.

Currently residing in Vancouver with his wife Erin, you can find Sandy vigorously researching programming languages and delving into the intricacies of mathematics. When he's not busy advocating for purely functional programming, he can be found leaping over obstacles and attempting not to destroy coffee machines. Sandy Maguire's unique blend of expertise, creativity, and adventurous spirit make him a compelling and dynamic figure in the world of software.

Books by Sandy Maguire

- Thinking with Types, 2018, Cofree Press
  available at https://leanpub.com/thinking-with-types
- Algebra-Driven Design, 2020, Cofree Press
  available at https://leanpub.com/algebra-driven-design
- Certainty by Construction, 2023, Cofree Press
  available at https://leanpub.com/certainty-by-construction