Functional Programming: A Conceptual Guide

Will Johnson

June 24, 2021

Contents

Chapter 1

Introduction

1.1 Goals & Non-Goals

The primary goal of this document is to provide a conceptual overview and motiviation for functional programming concepts in a more or less language-agnostic way.

Writing a conceptual guide, rather than aiming to teach a particular language, is somewhat quirky. A cynical but not completely inaccurate reader might observe that this choice (a) absolves us from having to attend to messy practical matters (build systems, disagreeable syntax...), and (b) is a marketing decision because no one will pick up "Yet Another Haskell Book: This Time It'll Make Sense, We Promise!" However, we believe this choice is in the reader's interest too; to someone who is completely new to functional programming from even a conceptual standpoint, gaining intuition about how programs are organized and composed is difficult to do simultaneously with learning the practical parts like performance concerns and library management.

Ultimately this is still going to be about programming, so there will be "code" snippets. However, it is important to emphasize again that it is **not** a goal to teach any one programming language. Some examples will be written in languages such as Java or ECMAScript for comparison purposes, since they should be recognizable to most readers. That being said, Appendix A.1 provides a brief glance over some of the ways in which particular languages deviate from the names or styles introduced below.

For functional-style snippets, we will be using a syntax inspired by ML-family languages like Haskell because those languages were designed from the ground up to make programming in a functional style as efficient as possible. It is called "PseudoML" because it is sufficient to illustrate concepts, but it lacks many of the features that would make it suitable for parsing by a compiler; it is intended only for human readers. Section 2.1 goes over that syntax, but first we should introduce functional programming in general.

1.2 Functional Programming: What And Why

Let's begin at the beginning. Programming languages are an abstraction over instructions that can be run by your CPU. These instructions are fundamentally *imperative*: we are telling the machine to read and write data between memory registers. Low-level languages like C are designed to be very thin, platform-independent wrappers over these instructions, with minimal control structures (for loops, structs) to simplify the work of reading and writing code.

C also allowed giving a name to a repeatable set of instructions that can be called from other parts of the program. These are usually called functions, but *routine* might be a better word. Object-oriented languages like C++ or Java provide an additional level of abstraction over C, but keep the same imperative flavor: a *class* describes a set of data and routines to operate on that data.

By contrast, a (purely) functional programming language departs from the imperative style entirely. They are *declarative*: we describe *what things are* rather than *how to do them*, and the compiler takes care of translating what we write into imperative language. To illustrate the difference, Listing 1.1 shows some example code, written in imperative-style ECMAScript (1.1a) or in declarative pseudo-ML (1.1b).

Both examples illustrate breaking the problem down into smaller pieces, but that's where the similarities end. A loop, for instance, is an imperative construct: "repeatedly execute this block of instructions". In functional programming, iterating over a list is generally accomplished by doing something with the head of the list and then recursing through the remainder (as in `listRemainingTodos`). Pattern matching on function arguments takes the place of if blocks to direct execution. These and other quirks of functional programming are all due to the fundamental difference between imperative and declarative styles.

That takes care of the "what". Why do people like functional programming? What does it gain over other paradigms? As with all things, it comes down to a combination of personal preference and math.

1.2.1 Personal Preference

People who prefer functional programming tend to be the same set of people who like statically-typed languages with very smart compilers. This is where basically all of the room for different opinions resides; I'll start by going over some of the reasoning behind that preference (as well as some of the arguments against it). Later on, I'll go over how that has to do with functional programming.

First, some definitions. A *type system* is a way of assigning a property called *type* to a given *value* in the programming language. Types are a human construct that tell us something about what a program means: after all, inside the silicon we're just shuffling bits around. But when we have a function called `stringLength` and let the computer plug any random bits into it, there's a good

```javascript
function makeGreeting(name, todos) {
    const firstPart = `Hi, ${name}!`;
    let secondPart;
    if (todos && todos.length > 0) {
        let todoMsg = todos[0];
        for (let i = 1; i < todos.length; i++) {
            todoMsg += `; ${todos[i]}`;
        }
        secondPart = `Here are your tasks for the day: ${todoMsg}`;
    } else {
        secondPart = 'Congrats, you\'re all done for the day!';
    }
    return `${firstPart} ${secondPart}`
}
```

(a) An example of imperative style in ECMAScript.

```haskell
makeGreeting name todos =
    sayHello name ++ " " ++ todosMessage todos

sayHello name = "Hi, " ++ name ++ "!"

todosMessage []       = "Congrats, you're all done for the day!"
todosMessage (t :: ts) = todosIntro ++ todosList where
    todosIntro = "Here are your tasks for the day: "
    todosList  = t ++ listRemainingTodos ts

listRemainingTodos []       = "."
listRemainingTodos (t :: ts) = "; " ++ t ++ listRemainingTodos ts
```

(b) An example of declarative style in PseudoML.

Listing 1.1: Comparison of imperative and declarative styles.

chance that whatever comes out won't even make sense as an integer, let alone a measure of length!

To prevent this kind of faux pas, we tell the compiler (or the compiler infers from usage, if it's a smarty pants) that this function should only be called with one argument, a string, and the return value should only be used like an integer. We are in effect getting some documentation for free: clients can read the type signature of a function and understand quite a lot about its behavior, especially if we can guarantee that there are no "side effects" (more on that later). But that's not all! Static type checking can be considered a form of testing: not only does the compiler guarantee that the function is used correctly, but it can also guarantee that the *definition* of the function upholds the type signature we claimed it should!

Strong static typing isn't a universal solution, though. We are in effect front-loading the work of discovering funny edge cases to compile-time, rather than run-time. Dynamically-typed, interpreted languages like Python and ECMAScript derive a lot of their appeal from the fact that they make it easy to crack out a lot of code *fast*, and as long as it's syntactically correct it will run. With adequate tests, you can be reasonably sure that the code is correct; for small projects or scripts, this is frequently much more efficient.

But what happens when your project gets popular and starts being used by other people? Even with excellent documentation (which, of course, is always available) there's nothing preventing someone from accidentally passing a string into a function that expects a number. Can you guarantee that your code will fail quickly in such an event, without putting the client's system into a corrupted state? *Should* you be expected to verify that at runtime?

Statically-typechecked code *drastically* reduces the surface area for runtime errors, filtering out the noise from programmer errors or typos, and pulling them all the way to the front of the development cycle. For situations where that is desirable, like critical enterprise software that needs firm guarantees of correctness, the cost of the additional developer time fighting with the compiler is well worth avoiding potentially costly runtime issues. Even in less critical software, the ability of IDEs to typecheck code as you write it reduces the feedback loop even further, with the strong guarantee of correctness a happy side benefit.

Long story short, it comes down to: would you rather write possibly correct code really quickly, or really correct code possibly quickly? For various reasons that we'll explore presently, functional programming is an excellent choice for anyone who picks the latter option.

1.2.2 Math

Functional programming languages are designed to be very close to the language that mathematicians use to prove things like "does this algorithm terminate?" This makes it possible to write an *exceptionally* smart compiler. Some of the questions that mathematicians might ask are

- Can it be guaranteed that this program doesn't have an infinite loop?
- Can it be guaranteed that this program will run without an error?
- Can it be guaranteed that this program won't set my grandma on fire?

The language that mathematicians and logicians use to describe and (attempt to) answer these questions is called the *lambda calculus*, which is very much out of the scope of this paper. Atop that framework is a language of types, which lets us say things like "Here is a function called `stringLength`; if you plug in a string, this will return an integer representing the number of characters in that string; no other inputs are allowed." At this point, a compiler can check things like

- Reject any program that attempts to plug something other than a string into `stringLength`
- Reject any program that attempts to use the output of `stringLength` as anything other than an integer
- Reject the program if `stringLength` returns something other than an integer
- Reject the program if `stringLength` does not accept any valid string

and—importantly—it is possible to *mathematically prove* that the compiler answers those questions correctly. It is the ultimate in test technology: rather than relying on a mere finite number of example cases as in traditional testing, we can rely on **Mathematical Truth**$^{\text{TM}}$!

Now, all that being said, software engineers shouldn't be expected to have math degrees! None of that background is required to actually *use* the FP toolkit, in the same way that we don't need to know the instruction set for the processors in our laptops. It is just a convenience that we can take for granted when we write our code and it compiles.

1.3 Functional Programming: Why Not?

The mathematical heritage of functional programming has given it a reputation for being difficult to understand, or just a research toy for mathematicians and academics. There is a reason why this image developed, but it is not really well-deserved.

Consider the languages most of us use today. From their earliest ancestors, they were developed by computer nerds, who just wanted to tinker around and play tetris and talk to others of their kind on message boards.

By contrast, as we saw above, many of the contributions that formed the foundation for functional programming languages came from mathematicians and logicians. This is actually a very useful thing for us, but it comes with a

price: they were there first, so they got to pick the names. Mathematicians are perfectly happy floating around in wizard robes and unironically saying things like "oh yes Veronica, monads are just monoids in the category of endofunctors!" Engineers live much closer to the real world and do not have time for such frippery, and have satisfied themselves with more normal-sounding terms like "class", "object", "singleton", or "factory".

So, yes, there will be some unfamiliar and mystical-sounding terms ahead. But fear not: they are just names, and the things they represent have solid programmer-friendly meanings.

Chapter 2

Basic Concepts

2.1 PseudoML Syntax

Type declarations and assignments look like this:

```
myInt : Int
myInt = 5

myString : String
myString = "Hello, world!"

type String = [Char]

myCharList : String
myCharList = ['H', 'e', 'l', 'l', 'o']
```

The `type` keyword creates a *type synonym*; this is just a different label, sometimes to save on keyboard work for a frequently-used but complex type name, or sometimes as a form of documentation (as in `type UserId = String`). Brand new types are declared with `data`:

```
data Optional a = Just a | None

justFive : Optional Int
justFive = Just 5

noInt : Optional Int
noInt = None
```

Functions are defined like so:

```
stringLength : String → Int
stringLength ""     = 0
stringLength (_::s) = 1 + stringLength s
```

Note the *pattern matching* on the left-hand side of the `=`. This should be read as "`stringLength` called with an empty string returns 0; otherwise, we know it isn't empty, so discard the head of the list and add 1 to the length of the remainder."

Function application (that is, "calling" a function) is achieved by simply listing the arguments after the function name, as in `f x y z` which calls a function `f` with three arguments. Arguments are always applied from left to right:

```
myValue = f x y z -- same as:  (((f x) y ) z)
```

and in general, it has higher precedence than any other operator:

```
myOtherValue = f x + g y -- same as:  (f x) + (g y)
```

2.2 Composition & Currying

Suppose we have

```
stringLength : String → Int
isOdd : Int → Bool
```

We could write

```
stringHasOddLength : String → Bool
stringHasOddLength s = isOdd (stringLength s)
```

but that gets tedious quickly if we are chaining several functions together. The idiomatic way to do this is to use *function composition*:

```
stringHasOddLength = isOdd . stringLength
```

The . in that definition is itself a function:

```
(.) : (b → c) → (a → b) → a → c
(f . g) x = f (g x)
```

It says "take the output of the function on the right, and plug it into the function on the left". Note that the new definition of `stringHasOddLength` does not actually bind a name for the argument! We *can* do that, as in

```
stringHasOddLength x = (isOdd . stringLength) x
```

but there is no need. We're just saying "`stringHasOddLength` is the result of composing these two functions." This is called, somewhat misleadingly, "point-free style."[1] You can do it in ES too:

```
const compose = (g, f) => x => g(f(x));
const stringLength = s => s.length;
const isOdd = i => i % 2 === 1;
const stringHasOddLength = compose(isOdd, stringLength);
```

If you were to take a static type analysis tool to this code, it would hopefully resolve the type of `stringHasOddLength` as a function from strings to booleans, despite not having actually used an explicit function definition that binds an argument name.

Point-free style is related to the concept of "currying", which is named after a person named Haskell Curry, not the food. If a function takes two arguments, and you feed it only one, the result is a function that takes one argument:

[1] Mathematicians again. "Point" means "function argument"; "point-free" means "defined without binding a name to the arguments".

```
stringLengths = map stringLength
```

The `map` function is the usual: it takes a function `(a → b)` and a list `[a]`, and then returns the result of applying the given function to each element of the list. So if we stare at this definition, since `stringLength : String → Int`, we can deduce that `stringLengths : [String] → [Int]`.

2.3 Recursion

Recursion is much more important in functional programming than in imperative programming, because recursion is the primary way to implement loops (in addition to the various other uses that it has in common with non-functional code). We will have much more to say about recursion later on (see Part 5), but for now we can go over some basic examples to get us started.

Here's how we might implement `map` over lists:

```
map : (a → b) → [a] → [b]
map _ []       = []
map f (x::xs) = f x :: map f xs
```

The `::` constructor sticks an element on the head of a list. The first equation takes care of the base case (stop recursing once we hit the end of the list), and the second one says to apply the function to the head of the list, and then do the same thing on to the remainder.

Here's another example, which works rather like Python's `range()` with one argument:

```
range : Int → [Int]
range x = if x < 0 then [] else (x - 1) :: range (x - 1)
```

2.4 Purity

There are a couple of meanings of *pure*, depending on context:

2.4.1 Pure Functional Languages

A *pure functional language*, such as Haskell, is a language that only supports functional-style programming, with no way to represent other programming patterns like object-oriented code. These are usually contrasted with "functional-first" languages like F# or, depending on who you are talking to, Scala; in these languages, functional and object-oriented styles can coexist.

PseudoML is a pure functional language, because it was invented[2] to illustrate functional programming concepts, so attempting to support additional syntax would just be distracting.

[2] Well, perhaps it is more accurate to say "shamelessly cobbled together from bits of existing languages"

2.4.2 Pure Functions

A *pure function* is a function that does not have any "side effects", such as updating a global state, writing to (or reading from!) a file, and so on. You don't need a pure functional language to write a pure function; here's one in ES:

```
const pureAdd = (x, y) => x + y;
```

Pure functions are important for a couple of reasons. From a practical standpoint, they are easy to test; without any global state that can be corrupted by another process, or flaky I/O operations, we know that a pure function called with the same arguments will always produce the same result. From the standpoint of implementing a language, we can get a form of memoization for free: results of pure functions can be cached since the compiler can guarantee that there's no way for the result to change from one call to the next.

Of course, it doesn't make sense for a language to only support pure functions; the *whole point* of running a program is to get the side effects! Side-effectful operations belong to their own type, `IO a`. For instance, an `IO Int` represents an operation that does something unspecified and then returns an `Int`. They are, therefore, somewhat spooky; running an `IO` "action" is dangerous if you don't know where it's been, since even an innocuous-seeming `IO Int` could represent the action "wipe the hard drive and then return the number of dirty pictures that this program emailed your grandma".

To facilitate effective testing (and out of a desire to be tidy), functional programmers generally try to keep as much of their code pure as possible. Consider, for instance, parsing an image file and returning the number of red pixels present in the image. Side effects are only required when reading the file and then printing the result to the console; everything in between is pure operations on the contents of the file, which is just a series of bytes. Folks who are new to functional programming often find the restriction of side effects to `IO` grating, but that is just after a lifetime of being able to sprinkle side effects around anywhere. Once you get used to structuring your code appropriately, it becomes second nature, and eventually the cavalier attitude of other languages toward side effects starts to feel a bit rude!

```
function justAddIPromise(x, y) {
    window.open('http://downloadvirus.biz');
    sendDirtyEmails('grandma@oldfolks.net');
    console.log('ha ha you suck');
    return x + y;
}
```

2.5 Strictness & Laziness

Strictness refers to whether or not a given expression in the source code is actually evaluated by the processor. One famous example of "non-strict semantics" is what we call short-circuiting in boolean operators:

```
const everythingIsFine = true || fireAllMissiles();
```

The expression on the left-hand side of the `||` is always evaluated, but the right-hand side may not be, if the processor knows by then what the overall expression will evaluate to. On the other hand, most other operations, such as function calls, follow "strict semantics":

```
function uhoh(stuff, things) {
    console.log(`here is some stuff: ${stuff}`);
}

uhoh('my stuff', fireAllMissiles());
```

Whenever you call a function, the arguments are always evaluated *first*, and then they are passed to the body of the function—regardless of whether the function body even refers to every argument it's given.

A term that is frequently used alongside "non-strict" is "lazy". Laziness is a way to *implement* non-strictness. In a lazy language, all expressions are implicitly replaced by zero-argument functions that *return* the expression's value, called a "thunk":

```
const two = 2;
const twoThunk = () => 2;
```

This is done behind the scenes, or else the code would be unacceptably cluttered. Although it makes it somewhat hard to decide whether some code will execute before or after another, the only times where that usually matters (namely, executing side-effectful actions) are wrapped up in the `IO` monad (more on that later!) which has a sense of "do this before that" built-in to the structure.

PseudoML as used here will in general be non-strict, though in the few places where it matters we will point that out. Specific languages have different ways to achieve strictness/non-strictness when that is not the default behavior, so we will leave it up to the reader to determine how to implement that in the wild.

2.6 Typeclasses

A typeclass is a set of functions that can be overloaded to work with any type. Defining how those functions work on a particular type is called *implementing* that typeclass. The most basic typeclasses are `Eq`, `Ord`, and `Show`, which we will go over here. In the next section, we'll start getting into some of the meatier examples.

Languages with this concept usually include several functions in the typeclass definition, many of which may be given a "default" definition in terms of some minimal set that must be implemented. This is entirely for practical purposes; in specific cases, there may be a more efficient way to implement one of the "extra" functions. Such considerations are an implementation detail outside our scope, so we will limit our typeclasses to the minimal set of functions, and define the other ones separately when they are needed.

2.6.1 Eq

A data type can be made an instance of `Eq` if its values can be compared as equal or not equal. It is defined like this:

```
typeclass Eq a
    (==) : a → a → Bool
```

This says "`a` is an instance of `Eq` if there is an implementation for the `(==)` function." As an example, consider a data type representing the three primary colors:

```
data PrimaryColor = Red | Blue | Green

instance Eq PrimaryColor
    Red   == Red   = True
    Blue  == Blue  = True
    Green == Green = True
    _     == _     = False
```

For "obvious" cases like this, the compiler can frequently implement this sort of thing on its own, but that functionality is generally language-dependent.

Polymorphic functions are written like this:

```
elem : Eq a ⇒ a → [a] → Bool
elem _ []       = False
elem e (x::xs) = e == x || elem e xs
```

The ⇒ notation says that `a` can be any type, as long as it has an `Eq` instance. This is a function of two arguments: something to look for in a list, and the list in which to look. The second line says "nothing is in an empty list." The third line says "check the first element in the list; if it is equal to what you're looking for, return `True`; otherwise, keep looking in the rest of the list."

Incidentally, how do we compare lists? Two lists are equal if they have the same elements in the same order. This means that we need a way to compare the elements to see if they're equal too. So we might write `Eq [a]` like:

```
instance Eq a ⇒ Eq [a]
    []      == []      = True
    (x::xs) == (y::ys) = x == y && xs == ys
    _       == _       = False
```

This says "two empty lists are equal; two nonempty lists are equal if their heads and tails are equal; otherwise, they are never equal." Note that the `Eq a` constraint is what lets us use `x == y`.

2.6.2 Ord

`Ord` types support a notion of "ordering". The class is defined like so:

```
data Ordering = LT | EQ | GT

typeclass Eq a ⇒ Ord a
    compare : a → a → Ordering
```

Similarly to its use in type signatures, the ⇒ at the top says that in order to be an `Ord`, the type must also implement `Eq`. The usual operators like `<` are then defined in terms of `compare`; for instance,

```
x <  y = compare x y == LT
x >= y = not (x < y)
max x y = if x >= y then x else y
```

One of the nifty examples using `Ord` is a recursive implementation of the QuickSort algorithm.

```
sort : (Ord a) ⇒ [a] ⇒ [a]
sort []      = []
sort (x::xs) = sort left ++ (x :: sort right) where
    left  = filter (<  x) xs
    right = filter (>= x) xs
```

This is a pretty popular canonical "look at how much better FP is!" example, but it emphasizes cuteness over performance. If you are tempted to compare it favorably against an in-place sort implemented in e.g. C, keep in mind that the in-place algorithm is going to be much more space-efficient, and almost certainly less obvious than this little toy algorithm.

2.6.3 Show

`Show` is for types that can be represented as a string:

```
typeclass Show a
    show : a → String
```

Most languages also have the ability to generate instances of `Show` for you. This is also simplified from the definition you might see in the wild, which is designed to support efficiently building the output string for nested structures. Just for fun, here's an example instance for lists whose elements are themselves `Show`able.

```
instance Show a ⇒ Show [a]
    show []      = "[]"
    show (x::xs) = "[" ++ show x ++ showNextElems xs ++ "]"

showNextElems : Show a ⇒ [a] → String
showNextElems []      = ""
showNextElems (x::xs) = ", " ++ show x ++ showNextElems xs
```

Chapter 3

Intermediate Typeclasses

So far, the typeclasses we've seen have obvious analogues in non-functional languages too. This section will look at a few of the typeclasses that are particularly important to functional programming design.

3.1 Functor

The `Functor` typeclass is for data types that can be treated like containers whose elements can be "mapped over." Specifically:

```
typeclass Functor f
    map : (a → b) → f a → f b
```

You might also think about a functor as a way to apply a function *through* a data structure. Lists are a common example; indeed, many object-oriented languages give a `map` method to their array or list class.

```
instance Functor []
    map _ []       = []
    map f (x::xs) = f x :: map f xs

-- this evaluates to [10, 4]
someInts = map stringLength ["functional", "peep"]
```

`Optional` is also a useful `Functor`:

```
instance Functor Optional
    map _ Nothing  = Nothing
    map f (Just x) = Just (f x)
```

3.2 Foldable

`Foldable` is the typeclass of data structures that can be traversed, accumulating some result at each point:

```
typeclass Foldable t
    foldl : (b → a → b) → b → t a → b
```

The `l` at the end of `foldl` indicates that this is a *left fold*[1]. The words "left" and "right" refer to the head and tail of a list, respectively; in general, left folds start at the "front" of a data structure (they are *breadth-first*), and right folds start at the "back" (*depth-first*). What this means is best illustrated by an example. Consider this definition of the `sum` function, which sums a list of integers:

```
instance Foldable []
    foldl _ acc []     = acc
    foldl f acc (x:xs) = foldl f (f acc x) xs

sum : [Int] → Int
sum = foldl (+) 0
```

Now let's consider what happens when this function is evaluated.

```
sum [1, 2, 4, 8]
foldl (+) 0 [1, 2, 4, 8]
foldl (+) (0 + 1) [2, 4, 8]
foldl (+) ((0 + 1) + 2) [4, 8]
foldl (+) (((0 + 1) + 2) + 4) [8]
foldl (+) ((((0 + 1) + 2) + 4) + 8) []
((((0 + 1) + 2) + 4) + 8)
```

As you can see, the first thing to be evaluated is $0+1$, and we proceed down the list, evaluating the "left-most" operations first. This implies the existence of a *right fold*:

```
-- see Appendix A.2 for a generic foldr in terms of foldl
foldrList : (a → b → b) → b → [a] → b
foldrList _ b [] = b
foldrList f b (x :: xs) = f x (foldr f b xs)

sumr : [Int] → Int
sumr = foldr (+) 0
```

When this is evaluated, we get

```
sumr [1, 2, 4, 8]
foldr (+) 0 [1, 2, 4, 8]
(1 + (foldr (+) 0 [2, 4, 8]))
(1 + (2 + (foldr (+) 0 [4, 8])))
(1 + (2 + (4 + (foldr (+) 0 [8]))))
(1 + (2 + (4 + (8 + (foldr (+) 0 [])))))
(1 + (2 + (4 + (8 + 0))))
```

Unsurprisingly, now we're starting on the right! This ends up evaluating to the same result, but that is only the case for *associative* operations. You may recall from math class that this has to do with how we group a series of operations; if we just write $0+1+2+4+8$ there are five different $+$s that we

[1] This is also called a *left-associative* fold, when there are mathematicians around.

could choose to evaluate first. Of course, with addition, it doesn't matter; any way we group the operations comes out to the same result. We call functions with this property associative. On the other hand, subtraction is definitely not associative:

$$((((0-1)-2)-4)-8) = -15$$
$$(1-(2-(4-(8-0)))) = -5$$

In this case, `foldl` and `foldr` give different results! This isn't actually that big of a deal though—if you know which side you're starting from, you can always define your folding function appropriately (and perhaps reverse your list) in order to get the result you want. It turns out, though, that sometimes it does matter which fold you choose!

The examples above with `(+)` are *reductions*: they collapse the list as they traverse it. Both reductions happen in linear time (since they traverse the input list exactly once), but `foldl` happens in constant space, while `foldr` uses linear space! For very long lists, this can easily overflow the stack. The reason is that when folding from the left, we're keeping a "running total" of the folded value; each rescursive call need not generate its own stack frame, so the fold only needs as much memory as is required to store the result value. On the other hand, folding from the right means that we must traverse the entire list before we can start evaluating stuff, and each time we recurse further into the list, we have to hold on to the current value while we wait for the evaluation to work its way back up the stack!

However, not all folds are reductions, and interestingly, the situation is reversed for non-reductive folds. Consider the two functions below, which implement `map` over a list, one with a left fold and the other with a right fold. You should be able to convince yourself that they both produce the same result as we saw for the `Functor` instance above:

```
mapl : (a → b) → [a] → [b]
mapl f = foldl mapAndAppend [] where
    mapAndAppend ys x = ys ++ [f x]

mapr : (a → b) → [a] → [b]
mapr f = foldr mapAndPrepend [] where
    mapAndPrepend x ys = f x :: ys
```

Note that when we're folding from the left, we put each successive result at the *end* of the new list. Likewise, when we're folding from the right, we start at the end of the list, so we append each result to the head of the new list. Evaluating these as we did before, we get

```
mapl stringLength ["Mrs", "Birdy", "says", "peep"]
-- (((([] ++ [3]) ++ [5]) ++ [4]) ++ [4])

mapr stringLength ["Mrs", "Birdy", "says", "peep"]
-- (3 ::  (5 ::  (4 ::  (4 ::  [])))
```

The problem here is that concatenation using `++` runs in time proportional to the length of the left-hand list, and each time we do a concatenation, the left-hand list gets bigger; suddenly our left fold is in *quadratic time*! We would therefore rather choose a right fold for this job, because it allows us to use the constant-time list constructor `::` rather than linear-time concatenation.

Now, depending on your language's evaluation rules, how it implements lists, and particularly how smart its optimizer is, your mileage may vary. The moral of this story is that you should choose your fold so that reductions are *strict and tail-recursive*, and non-reductive folds build the output structure efficiently, using only constant-time operations (if possible).

3.3 Monoid

First, a warning: monoids are to monads as Java is to JavaScript, so apologies in advance for the similar words. Blame mathematicians again.

Here's the definition of `Monoid`:

```
typeclass Monoid a
    empty : a
    (<>) : a → a → a
```

This can be read a couple of different ways. Usually the one folks see first treats `<>` as an operator for glomming two instances of the monoid together, with `empty` as the "neutral" element; for example, with integers:

```
instance Monoid Int as Sum
    empty = 0
    (<>)  = (+)

sum : [Int] → Int
sum xs = foldl (<> using Prod)
```

Notice that I have named the instance; this can sometimes be useful, because there may be multiple ways for a given data type to implement a typeclass. Such as:

```
instance Monoid Int as Product
    empty = 1
    (<>)  = (*)
```

Each of these specifies a particular way that integers can be stuck together. With these examples handy, we can write down the *monoid laws*:

Associativity `(x <> y) <> z == x <> (y <> z)`

Identity `x <> empty == empty <> x == x`

The requirement that `<>` be associative means that there aren't monoid instances for division or subtraction. (By the way, division has another problem too—`<>` should always be defined for all values, but division by zero isn't defined!)

The other way to interpret a monoid is as a way to choose between two values with `<>`, with `empty` providing a default choice.

```
instance Monoid (Maybe a) as First
    empty = Nothing

    Just x <> Just y = Just x
    x <> Nothing     = x
    Nothing <> x     = x

instance Monoid (Maybe a) as Last
    empty = Nothing

    Just x <> Just y = Just y
    x <> Nothing     = x
    Nothing <> x     = x
```

Here, the `First` instance always chooses the first non-`Nothing` value it was given; likewise, `Last` always chooses the last.

As a final example, `Bool` also admits two possible monoids:

```
instance Monoid Bool as All
    empty = True
    (<>)  = (&&)

instance Monoid (Maybe a) as Any
    empty = False
    (<>)  = (||)
```

3.4 Applicative

The extravagantly-named *applicative functor* is, of course, simply a functor that is applicative!

That sounds deeply, almost offensively unhelpful, but interestingly it's one of the more meaningful names for important concepts (looking at you, 'Monad'). To illustrate what it means, let's consider a puzzle. A program has asked the user for two integers, `x` and `y`, but since getting these integers involves communing with the outside world of side effects, they are both of type `IO Int`. Your goal is to add them together. How can we do this?

Unlike most data types, `IO` values cannot be "unwrapped", because that would defeat the purpose of keeping side effects contained. `IO` is a functor, so we can do things like

```
x : IO Int
x = askUserForInt

y : IO Int
y = askUserForInt

z : IO Int
z = map (*2) x -- double it!
```

but before you ask, `x + y` doesn't work because `IO Int` is not a number! It's more like a *promise* of a number, and in fact thinking about `IO` like an ES `Promise` or a Java `CompletableFuture` is not a terrible approximation.

Okay fine, it's a trick question, and presumably you have already figured out that the answer has to do with whatever an applicative is. Plain functors simply don't provide enough power to support this sort of operation. Happily, `IO` is an `Applicative`, which gives us access to this gadget:

```
liftA2 : (Applicative f) ⇒ (a → b → c) → (f a → f b → f c)
-- definition will come in a moment!

addTwoIOs : IO Int → IO Int → IO Int
addTwoIOs = liftA2 (+)

addXAndY = addTwoIOs x y -- ta da!
```

The function `liftA2` takes a pure function of two arguments, and turns it into a function over an Applicative. The term *lift* is one that will occur a lot; it's usually given to a function that takes a "plain" function and transforms it into a "special" one—e.g. *lifting* the humble `(+)` into the exciting world of `IO`. "`liftA`" denotes a lift into Applicatives, and "`liftA2`" indicates that it operates on functions of two arguments; once you get over that hurdle, it's easy enough to construct `liftA`n but usually that's excessive. In fact, you've already seen `liftA1`: it's just functor `map`!

```
liftA1 : (Applicative f) ⇒ (a → b) → (f a → f b)
-- where have I seen this type signature before?
```

Hopefully that is enough to start shedding light on the name *applicative functor*. Let's look at how it's actually defined.

```
typeclass (Functor f) ⇒ Applicative f
    pure  : a → f a
    (<*>) : f (a → b) → f a → f b
```

The `pure` function lifts a plain value into an applicative. The name is intended to suggest that we're getting "just" that value: no spooky side effects, no accidental emails to scandalize grandma, it's a pure value. For instance, if we didn't want to bother asking the user for numbers (they would probably screw it up anyway), we could just say

```
myX : IO Int
myX = pure 2

myY : IO Int
myY = pure 3
```

The other thing, `(<*>)`, is pronounced "apply", and it takes a lifted single-argument function and applies it to a lifted value. These two things together allow us to define `liftA2`:

```
liftA2 : (Applicative f) ⇒ (a → b → c) → (f a → f b → f c)
liftA2 f x y = pure f <*> x <*> y
```

Which is to say, we lift `f` up into the applicative, (partially!) apply it to `x`, and then finally apply that to `y`. In fact, we could have started with `liftA2` instead:

```
(<*>) : (Applicative f) → f (a → b) → f a → f b
f <*> x = liftA2 id f x
```

3.5 Traversable

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

3.6 Monad

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Chapter 4

Monads: A Field Guide

4.1 Optional

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

4.2 List

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

4.3 Either

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

4.4 IO

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

4.5 Reader

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

4.6 Writer

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue,

a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

4.7 State

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Chapter 5

Recursion Schemes

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Chapter 6

Advanced Typeclasses

6.1 Alternative

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

6.2 MonadPlus

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

6.3 Arrow

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Appendix A

Appendix

A.1 Language-Specific Caveats

Here we will discuss some "gotchas" or signficant syntactic deviations from some popular programming languages that might catch the inexperienced reader off-guard if they try to apply the concepts above to that language. Once again, the goal is not to teach a specific language; this is just a collection of warnings and disclaimers that would otherwise have cluttered the main narrative.

A.1.1 Scala / `cats`

- typeclasses via implicits/traits

A.1.2 Haskell

- type name translations (Maybe vs Optional, etc)
- data vs newtype
- newtype for re-implementing typeclasses
- dangers with laziness (e.g. foldr vs foldl)
- reader, writer, state defined in terms of their transformers

A.1.3 TypeScript / `fp-ts`

- syntax difficulties
- type name translations (Option vs Optional, etc)

A.1.4 Idris

- labelled interfaces

A.2 Right Folds From The Left

Here we are going to look into how to implement a right fold generically, given only a left fold and no other information about the data structure. The idea is that we fold the structure up, from the left, *into a function*, where the resulting function is designed to evaluate the right-most values first. Here's what that looks like:

```
foldr : Foldable t ⇒ (a → b → b) → b → t a → b
foldr f z t = (foldl foo bar t) z where
    foo = _
    bar = _
```

This is pretty much a direct translation of the idea above: we left-fold (somehow) the structure `t` into a function that evaluates from the right, then kick it off with the given starting value. Now we just need to figure out what to use for `foo` and `bar`! Let's start by looking at their types, to see if they give us any clues. We know that `foldl foo bar t` must be a function, and it should have type `b → b`. If we compare that to the type of `foldl`,

```
foldl : Foldable t ⇒ (c → a → c) → c → t a → c
```

in order to have the correct result type, we must have

```
foo : (b → b) → a → (b → b)
bar : (b → b)
```

The second one is easy: whenever you need a value that fits that type signature, it almost certainly should be the identity function `id`. What about `foo`? Well let's treat it as a function of two arguments:

```
foo : (b → b) → a → (b → b)
foo g x = _
```

At this point, let's consider what gadgets we have available to us. We haven't yet used `f : a → b → b`, and now we also have `g : b → b` and `x : a`. In general, unless you have a good reason, you want to try to use all of the variables you have handy. Interestingly, `f x` will have type `b → b`, so what if we just compose that with `g`?

```
foldr : Foldable t ⇒ (a → b → b) → b → t a → b
foldr f z t = (foldl foo id t) z where
    foo g x = g . f x
```

If you try this out, you'll find that this definition works exactly as we wanted it to! This is actually somewhat amazing, which is a pretty common occurrence with "type-driven development" as this method is usually called. We could have arrived at the same result if we sat down and worked out exactly what it means to "left-fold a structure into a function that executes a right-fold", but that would have required a lot more noodling.

To be honest, though, we cheated a little bit. Doing the composition in `foo` the other way around would have typechecked, but produces the wrong results:

```
notFoldr : Foldable t ⇒ (a → b → b) → b → t a → b
notFoldr f z t = (foldl foo id t) z where
    foo g x = f x . g
```

If you work out an example, this turns out to look like a right fold... but for a reversed input! This is the price of type-driven development: sometimes there is more than one choice to fill in a value for a given type, and the only way to determine which choice is correct is by testing it out yourself. Static type checking is not a substitute for *all* tests!