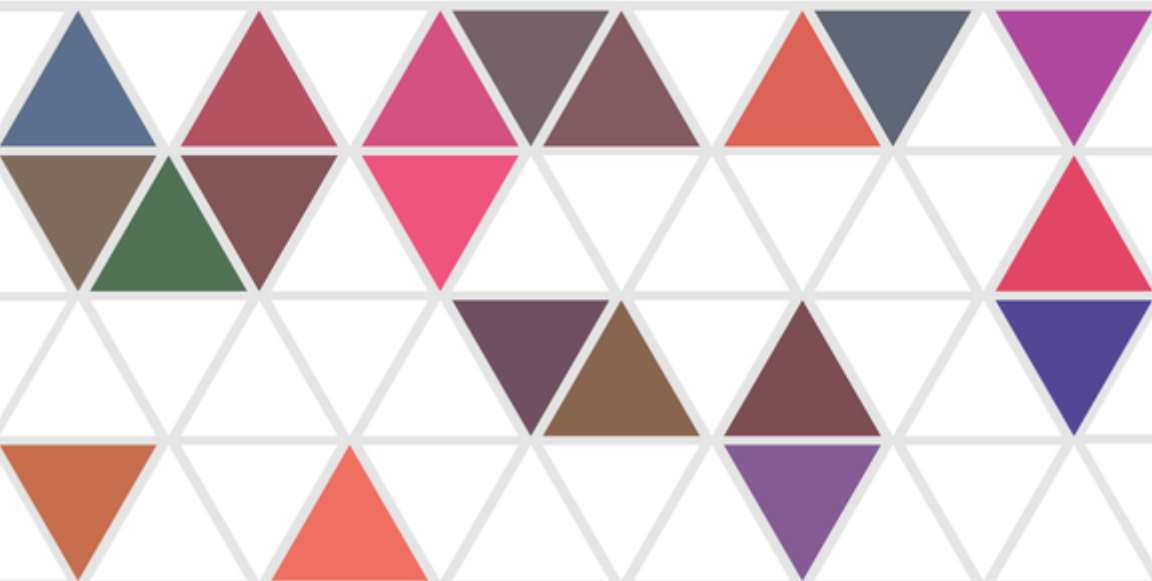
Maybe Haskell

by Pat Brisbin

thoughtbot



Maybe Haskell

Pat Brisbin

February 25, 2015

Contents

Introduction

As a programmer, I spend a lot of time dealing with the fallout from one specific problem: partial functions. A partial function is one that can't provide a valid result for all possible inputs. If you write a function (or method) to return the first element in an array that satisfies some condition, what do you do if no such element exists? You've been given an input for which you can't return a valid result. Aside from raising an exception, what can you do?

The most popular way to to deal with this is to return a special value that indicates failure. Ruby has `nil`, Java has `null`, and many C functions return `-1` in failure cases. This is a huge hassle. You now have a system in which any value at any time can either be the value you expect or `nil`, always.

If you try to find a `User`, and you get back a value, and you try to treat it like a `User` when it's actually `nil`, you get a `NoMethodError`. What's worse, that error may not happen anywhere near the source of the problem. The line of code that created that `nil` may not even appear in the eventual backtrace. The result is various "`nil` checks" peppered throughout the code. Is this the best we can do?

The problem of partial functions is not going away. User input may be invalid, files may not exist, networks may fail. We will always need a way to deal with partial functions. What we don't need is `null`.

An Alternate Solution

In languages with sufficiently-expressive type systems, we have another option: we can state in the types that certain values may not be present. Functions nor-

mally written in a partial way can instead return a type that captures any potential non-presence. Not only does it make it explicit and "type checked" that when a value may not be present, you have code to handle that case, but it also means that if a value is *not* of this special "nullable" type, you can feel safe in your assumption that the value's really there – No `nil`-checks required.

The focus of this book will be Haskell's implementation of this idea via the `Maybe` data type. This type and all of the functions that deal with it are not built-in, language-level constructs. All of it is implemented as libraries, written in a very straightforward way. In fact, we'll write most of that code ourselves over the course of this short book.

Haskell is not the only language to have such a construct. For example, Scala has a similar `Option` type and Swift has `Optional` with various built-in syntax elements to make its usage more convenient. Many of the ideas implemented in these languages were lifted directly from Haskell. If you happen to use one of them, it can be good to learn where the ideas originated.

Required Experience

I'll assume no prior Haskell experience. I expect that those reading this book will have programmed in other, more traditional languages, but I'll also ask that you *actively combat* your prior programming experience.

For example, you're going to see code like this:

```
countEvens = length . filter even
```

This is a function definition written in an entirely different style than you may be used to. Even so, I'll bet you can guess what it does, and even get close to how it does it: `filter even` probably takes a list and filters it for only even elements. `length` probably takes a list and returns the number of elements it has.

Given those fairly obvious facts, you might guess that putting two things together with `(.)` must mean you do one and then give the result to the other. That makes this expression a function which must take a list and return the number of even elements it has. Without mentioning any actual argument, we can directly assign

this function the name `countEvens`. There's no need in Haskell to say that count-evens *of some x* is to take the length after filtering for the even values *of that x*. We can state directly that count-evens is taking the length after filtering for evens.

This is a relatively contrived example, but it's indicative of the confusion that can happen at any level: if your first reaction is "So much syntax! What is this crazy dot thing!?", you're going to have a bad time. Instead, try to internalize the parts that make sense while getting comfortable with *not* knowing the parts that don't. As you learn more, the various bits will tie together in ways you might not expect.

Structure

We'll spend this entire book focused on a single *type constructor* called `Maybe`. We'll start by quickly covering the basics of Haskell, but only so far that we see the opportunity for such a type and can't help but invent it ourselves. With that defined, we'll quickly see that it's cumbersome to use. This is because Haskell has taken an inherently cumbersome concept and put it right in front of us by naming it and requiring we deal with it at every step.

From there, we'll explore three *type classes* whose presence will make our lives far less cumbersome. We'll see that `Maybe` has all of the properties required to call it a *functor*, an *applicative functor*, and even a *monad*. These terms may sound scary, but we'll go through them slowly, each concept building on the last. These three *interfaces* are crucial to how I/O is handled in a purely functional language such as Haskell. Understanding them will open your eyes to a whole new world of abstractions and demystify notoriously opaque topics.

Finally, with a firm grasp on how these concepts operate in the context of `Maybe`, we'll discuss other types which share these qualities. This is to reinforce the fact that these ideas are *abstractions*. They can be applied to any type that meets certain criteria. Ideas like *functor* and *monad* are not specifically tied to the concept of partial functions or nullable values. They apply broadly to things like lists, trees, exceptions, and program evaluation.

What This Book is Not

I don't intend to teach you Haskell. Rather, I want to show you *barely enough* Haskell so that I can wade into more interesting topics. I want to show how this `Maybe` data type can add safety to your code base while remaining convenient, expressive, and powerful. My hope is to show that Haskell and its "academic" ideas are not limited to PhD thesis papers. These ideas result directly in cleaner, more maintainable code that solves practical problems.

I won't describe how to set up a Haskell programming environment, show you how to write and run complete Haskell programs, or dive deeply into every language construct we'll see. If you are interested in going further and actually learning Haskell (and I hope you are!), then I recommend following Chris Allen's great learning path.

Lastly, a word of general advice before you get started:

The type system is not your enemy, it's your friend. It doesn't slow you down, it keeps you honest. Keep an open mind. Haskell is simpler than you think. Monads are not some mystical burrito, they're a simple abstraction which, when applied to a variety of problems, can lead to elegant solutions. Don't get bogged down in what you don't understand, dig deeper into what you do. And above all, take your time.

Haskell Basics

When we declare a function in Haskell, we first write a type signature:

```
five :: Int
```

We can read this as `five` *of type* `Int`.

Next, we write a definition:

```
five = 5
```

We can read this as `five` *is* `5`.

In Haskell, `=` is not variable assignment, it's defining equivalence. We're saying here that the word `five` *is equivalent to* the literal `5`. Anywhere you see one, you can replace it with the other and the program will always give the same answer. This property is called *referential transparency* and it holds true for any Haskell definition, no matter how complicated.

It's also possible to specify types with an *annotation* rather than a signature. We can annotate any expression with `:: <type>` to explicitly tell the compiler the type we want (or expect) that expression to have.

```
six = (5 :: Int) + 1
```

We can read this as `six` *is* `5`, *of type* `Int`, plus `1`

Type annotations and signatures are usually optional, as Haskell can almost always tell the type of an expression by inspecting the types of its constituent parts or seeing how it is eventually used. This process is called *type inference*. For example, Haskell knows that `six` is an `Int` because it saw that `5` is an `Int`. Since you can only use `(+)` with arguments of the same type, it *enforced* that `1` is also an `Int`. Knowing that `(+)` returns the same type as its arguments, the final result of the addition must itself be an `Int`.

Good Haskellers will include a type signature on all top-level definitions anyway. It provides executable documentation and may, in some cases, prevent errors which occur when the compiler assigns a more generic type than you might otherwise want.

Arguments

Defining functions that take arguments looks like this:

```
add :: Int -> Int -> Int
add x y = x + y
```

The type signature can be confusing because the argument types are not separated from the return type. There is a good reason for this, but I won't go into it yet. For now, feel free to mentally treat the thing after the last arrow as the return type.

After the type signature, we give the function's name (`add`) and names for any arguments it takes (`x` and `y`). On the other side of the `=`, we define an expression using those names.

Higher-order functions

Functions can take and return other functions. These are known as higher-order functions. In type signatures, any function arguments or return values must be surrounded by parentheses:

```
twice :: (Int -> Int) -> Int -> Int
twice op x = op (op x)
```

```
twice (add 2) 3
-- => 7
```

`twice` takes as its first argument a function, `(Int -> Int)`. As its second argument, it takes an `Int`. The body of the function applies the first argument (`op`) to the second (`x`) twice, returning another `Int`.

You also saw an example of *partial application*. The expression `add 2` returns a new function that itself takes the argument we left off. Let's break down that last expression to see how it works:

```
-- Add takes two Ints and returns an Int
add :: Int -> Int -> Int
add x y = x + y

-- Supplying only the first argument gives us a new function that will add 2 to
-- its argument. Its type is Int -> Int
add 2 :: Int -> Int

-- Which is exactly the type of twice's first argument
twice :: (Int -> Int) -> Int -> Int
twice op x = op (op x)

twice (add 2) 3
-- => add 2 (add 2 3)
-- => add 2 5
-- => 7
```

It's OK if this doesn't make complete sense now, I'll talk more about partial application as we go.

Operators

In the definition of `add`, I used something called an *operator*: `(+)`. Operators like this are not special or built-in in any way; we can define and use them like any other function. That said, there are three additional (and convenient) behaviors operators have:

1. They are used *infix* by default, meaning they appear between their arguments (as in `2 + 2`). To use an operator *prefix*, it must be surrounded in parentheses (as in `(+) 2 2`).
2. When defining an operator, we can assign a custom associativity and precedence relative to other operators. This tells Haskell how to group expressions like `2 + 3 * 5 / 10`.
3. We can surround an operator and *either* of its arguments in parentheses to get a new function that accepts whichever argument we left off. Expressions like `(+ 2)` and `(10 /)` are examples. The former adds `2` to something and the latter divides `10` by something. Expressions like these are called *sections*.

In Haskell, any function with a name made up entirely of punctuation (where The Haskell Report states very exactly what "punctuation" means) behaves like an operator. We can also take any normally-named function and treat it like an operator by surrounding it in backticks:

```
-- Normal usage of an elem function
elem 3 [1, 2, 3, 4, 5]
-- => True

-- Reads a little better infix
3 `elem` [1, 2, 3, 4, 5]
-- => True

-- Or as a section, leaving out the first argument
intersects xs ys = any (`elem` xs) ys
```

Lambdas

The last thing we need to know about functions is that they can be *anonymous*. Anonymous functions are called *lambdas* and are most frequently used as arguments to higher-order functions. Often these functional arguments only exist for a single use and giving them a name is not otherwise valuable.

The syntax is a back-slash, the arguments to the function, an arrow, then the body of the function. A back-slash is used because it looks like the Greek letter λ.

Here's an example:

```
twice (\x -> x * x + 10) 5
-- => 35
```

If you come across a code example using a lambda, you can always rewrite it to use named functions. Here's the process for this example:

```
-- Grab the lambda
\x -> x * x + 10

-- Name it
f = \x -> x * x + 10

-- Replace "\... ->" with normal arguments
f x = x * x + 10

-- Use the name instead
twice f 5
```

Our Own Data Types

We're not limited to basic types like `Int` or `String`. As you might expect, Haskell allows you to define custom data types:

```
data Person = MakePerson String Int
--                       |      |
--                       |      ` The persons's age
--                       |
--                       ` The person's name
```

To the left of the `=` is the *type* constructor and to the right can be one or more *data* constructors. The type constructor is the name of the type and is used in type signatures. The data constructors are functions which produce values of the given type. For example, `MakePerson` is a function that takes a `String` and an `Int`, and returns a `Person`. Note that I will often use the general term "constructor" to refer to a *data* constructor if the meaning is clear from context.

When there is only one data constructor, it's quite common to give it the same name as the type constructor. This is because it's syntactically impossible to use one in place of the other, so the compiler makes no restriction. Naming is hard, so if you have a good one, you might as well use it in both contexts.

```
data Person = Person String Int
--   |        |
--   |        ` Data constructor
--   |
--   ` Type constructor
```

With this data type declared, we can now use it to write functions that construct values of this type:

```
pat :: Person
pat = Person "Pat" 29
```

Pattern Matching

To get the individual parts back out again, we use pattern matching:

```
getName :: Person -> String
getName (Person name _) = name

getAge :: Person -> Int
getAge (Person _ age) = age
```

In the above definitions, each function is looking for values constructed with `Person`. If it gets an argument that matches (which is guaranteed since that's the only way to get a `Person` in our system so far), Haskell will use that function body with each part of the constructed value bound to the variables given. The `_` pattern (called a *wildcard*) is used for any parts we don't care about. Again, this is using `=` for equivalence (as always). We're saying that `getName`, when given `(Person name _)`, *is equivalent to* `name`. Similarly for `getAge`.

There are other ways to do this sort of thing, but we won't get into that here.

Sum Types

As alluded to earlier, types can have more than one data constructor, each separated by a `|` symbol. This is called a *sum type* because the total number of values you can build of this type is the sum of the number of values you can build with each constructor.

```
data Person = PersonWithAge String Int | PersonWithoutAge String

pat :: Person
pat = PersonWithAge "Pat" 29

jim :: Person
jim = PersonWithoutAge "Jim"
```

Notice that `pat` and `jim` are both values of type `Person`, but they've been constructed differently. We can use pattern matching to inspect how a value was constructed and choose what to do accordingly. Syntactically, this is accomplished by providing multiple definitions of the same function, each matching a different pattern. Each definition will be tried in the order defined, and the first function to match will be used.

This works well for pulling the name out of a value of our new `Person` type:

```
getName :: Person -> String
getName (PersonWithAge name _) = name
getName (PersonWithoutAge name) = name
```

But we must be careful when trying to pull out a person's age:

```
getAge :: Person -> Int
getAge (PersonWithAge _ age) = age
getAge (PersonWithoutAge _) = -- uh-oh
```

If we decide to be lazy and not define that second function body, Haskell will compile, but warn us about the *non-exhaustive* pattern match. What we've created at

that point is a *partial function*. If such a program ever attempts to match `getAge` with a `Person` that has no age, we'll see one of the few runtime errors possible in Haskell.

A person's name is always there, but their age may or may not be. Defining two constructors makes both cases explicit and forces anyone attempting to access a person's age to deal with its potential non-presence.

Maybe

Haskell's `Maybe` type is very similar to our `Person` example:

```
data Maybe a = Nothing | Just a
```

The difference is we're not dragging along a name this time. This type is only concerned with representing a value (of any type) which is either *present* or *not*.

We can use this to take functions which would otherwise be *partial* and make them *total*:

```
-- | Find the first element from the list for which the predicate function
--   returns True. Return Nothing if there is no such element.
find :: (a -> Bool) -> [a] -> Maybe a
find _ [] = Nothing
find predicate (first:rest) =
    if predicate first
        then Just first
        else find predicate rest
```

This function has two definitions matching two different patterns: if given the empty list, we immediately return `Nothing`. Otherwise, the (non-empty) list is deconstructed into its `first` element and the `rest` of the list by matching on the `(:)` (pronounced *cons*) constructor. Then we test if applying the `predicate` function to `first` returns `True`. If it does, we return `Just` it. Otherwise, we recurse and try to find the element in the `rest` of the list.

Returning a `Maybe` value forces all callers of `find` to deal with the potential `Nothing` case:

```
--
-- Warning: this is a type error, not working code!
--
findUser :: UserId -> User
findUser uid = find (matchesId uid) allUsers
```

This is a type error since the expression actually returns a `Maybe User`. Instead, we have to take that `Maybe User` and inspect it to see if something's there or not. We can do this via `case` which also supports pattern matching:

```
findUser :: UserId -> User
findUser uid =
    case find (matchesId uid) allUsers of
        Just u  -> u
        Nothing -> -- what to do? error?
```

Depending on your domain and the likelihood of Maybe values, you might find this sort of "stair-casing" propagating throughout your system. This can lead to the thought that `Maybe` isn't really all that valuable over some *null* value built into the language. If you need these `case` expressions peppered throughout the code base, how is that better than the analogous "`nil` checks"?

Functor

In the last chapter, we defined a type that allows any value of type `a` to carry with it additional information about if it's actually there or not:

```
data Maybe a = Nothing | Just a

actuallyFive :: Maybe Int
actuallyFive = Just 5

notReallyFive :: Maybe Int
notReallyFive = Nothing
```

As you can see, attempting to get at the value inside is dangerous:

```
getValue :: Maybe a -> a
getValue (Just x) = x
getValue Nothing = error "uh-oh"

getValue actuallyFive
-- => 5

getValue notReallyFive
-- => Runtime error!
```

At first, this seems severely limiting: how can we use something if we can't (safely) get at the value inside?

Choices

When confronted with some `Maybe a`, and you want to do something with an `a`, you have three choices:

1. Use the value if you can, otherwise throw an exception
2. Use the value if you can, but still have some way of returning a valid result if it's not there
3. Pass the buck, return a `Maybe` result yourself

The first option is a non-starter. As you saw, it is possible to throw runtime exceptions in Haskell via the `error` function, but you should avoid this at all costs. We're trying to remove runtime exceptions, not add them.

The second option is only possible in certain scenarios. You need to have some way to handle an incoming `Nothing`. That may mean skipping certain aspects of your computation or substituting another appropriate value. Usually, if you're given a completely abstract `Maybe a`, it's not possible to determine a substitute because you can't produce a value of type `a` out of nowhere.

Even if you did know the type (say you were given a `Maybe Int`) it would be unfair to your callers if you defined the safe substitute yourself. In one case `0` might be best because we're going to add something, but in another `1` would be better because we plan to multiply. It's best to let them handle it themselves using a utility function like `fromMaybe`:

```
fromMaybe :: a -> Maybe a -> a
fromMaybe x Nothing = x
fromMaybe _ (Just x) = x

fromMaybe 10 actuallyFive
-- => 5

fromMaybe 10 notReallyFive
-- => 10
```

Option 3 is actually a variation on option 2. By making your own result a `Maybe` you always have the ability to return `Nothing` yourself if the value's not present. If the

value *is* present, you can perform whatever computation you need to and wrap what would be your normal result in `Just`.

The main downside is that now your callers also have to consider how to deal with the `Maybe`. Given the same situation, they should again make the same choice (option 3), but that only pushes the problem up to their callers – any `Maybe` values tend to go *viral*.

Eventually, probably at some UI boundary, someone will need to "deal with" the `Maybe`, either by providing a substitute or skipping some action that might otherwise take place. This should happen only once, at that boundary. Every function between the source and final use should pass along the value's potential non-presence unchanged.

Even though it's safest for every function in our system to pass along a `Maybe` value, it would be extremely annoying to force them all to actually take and return `Maybe` values. Each function separately checking if they should go ahead and perform their computations will become repetitive and tedious. Instead, we can completely abstract this "pass along the `Maybe`" concern using higher-order functions and something called *functors*.

Discovering a Functor

Imagine we had a higher-order function called `whenJust`:

```
whenJust :: (a -> b) -> Maybe a -> Maybe b
whenJust f (Just x) = Just (f x)
whenJust _ Nothing = Nothing
```

It takes a function from `a` to `b` and a `Maybe a`. If the value's there, it applies the function and wraps the result in `Just`. If the value's not there, it returns `Nothing`. Note that it constructs a new value using the `Nothing` constructor. This is important because the value we're given is type `Maybe a` and we must return type `Maybe b`.

This allows the internals of our system to be made of functions (e.g. the `f` given to `whenJust`) that take and return normal, non-`Maybe` values, but still "pass along the `Maybe`" whenever we need to take a value from some source that may fail and manipulate it in some way. If it's there, we go ahead and manipulate it, but return the result as a `Maybe` as well. If it's not, we return `Nothing` directly.

```
whenJust (+5) actuallyFive
-- => Just 10

whenJust (+5) notReallyFive
-- => Nothing
```

This function exists in the Haskell Prelude as `fmap` in the `Functor` type class.

Functor

Haskell defines the type class `Functor` with a single member function, `fmap`:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Type constructors, like `Maybe`, implement `fmap` by defining a function where that `f` is replaced by themselves. We can see that `whenJust` has the correct type:

```
--          (a -> b) -> f     a -> f     b
whenJust :: (a -> b) -> Maybe a -> Maybe b
```

Therefore, we could implement a `Functor` instance for `Maybe` with the following code:

```
class Functor Maybe where
    fmap = whenJust
```

In reality, there is no `whenJust` function; `fmap` is implemented directly:

```
class Functor Maybe where
    fmap _ Nothing = Nothing
    fmap f (Just x) = Just (f x)
```

List

The most familiar example for an implementation of `fmap` is the one for `[]`. Like `Maybe` is the type constructor in `Maybe a`, `[]` is the type constructor in `[a]`. You can pronounce `[]` as *list* and `[a]` as *list of a*.

The basic `map` function, which exists in many languages, including Haskell, has the following type:

```
map :: (a -> b) -> [a] -> [b]
```

It takes a function from `a` to `b` and a *list of a*. It returns a *list of b* by applying the given function to each element in the list. Knowing that `[]` is a type constructor, and `[a]` represents applying that type constructor to some `a`, we can write this signature in a different way; one that shows it is the same as `fmap` when we replace the parameter `f` with `[]`:

```
--     (a -> b) -> f  a -> f  b
map :: (a -> b) -> [] a -> [] b
```

This is the same process as replacing `f` with `Maybe` used to show that `whenJust` and `fmap` were also equivalent.

`map` does exist in the Haskell Prelude (unlike `whenJust`), so the `Functor` instance for `[]` is in fact defined in terms of it:

```
instance Functor [] where
    fmap = map
```

What's in a Map

Now that we have two points of reference (`Maybe` and `[]`) for this idea of *mapping*, we can talk about some of the more interesting aspects of what it really means.

For example, you may have wondered why Haskell type signatures don't separate arguments from return values. The answer to that question should help clarify why the abstract function `fmap`, which works with far more than only lists, is aptly named.

Curried Form

In the implementation of purely functional programming languages, there is value in all functions taking exactly one argument and returning exactly one result. Therefore, users of Haskell have two choices for defining "multi-argument" functions.

We could rely solely on tuples:

```
add :: (Int, Int) -> Int
add (x, y) = x + y
```

This results in type signatures you might expect, where the argument types are shown separate from the return types. The problem with this form is that partial application can be cumbersome. How do you add 5 to every element in a list?

```
f :: [Int]
f = map add5 [1,2,3]

  where
    add5 :: Int -> Int
    add5 x = add (x, 5)
```

Alternatively, we could write all functions in "curried" form:

```
--
--     / One argument type, an Int
--     |
--     |       / One return type, a function from Int to Int
--     |       |
add :: Int -> (Int -> Int)
add x = \y -> x + y
--  |   |
--  |   ` One body expression, a lambda from Int to Int
--  |
--  ` One argument variable, an Int
--
```

This makes partial application simpler. Since `add 5` is a valid expression and is of the correct type to pass to `map`, we can use it directly:

```
f :: [Int]
f = map (add 5) [1,2,3]
```

While both forms are valid Haskell (in fact, the `curry` and `uncurry` functions in the Prelude convert functions between the two forms), the latter was chosen as the default and so Haskell's syntax allows some things that make it more convenient.

For example, we can name function arguments in whatever way we like; we don't have to always assign a single lambda expression as the function body. In fact, these are all equivalent:

```
add = \x -> \y -> x + y
add x = \y -> x + y
add x y = x + y
```

In type signatures, `(->)` is right-associative. This means that instead of writing:

```
addThree :: Int -> (Int -> (Int -> Int))
addThree x y z = x + y + z
```

We can write the less-noisy:

```
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

And it has the same meaning. This is why Haskell type signatures don't appear to separate argument types from return types. Technically, the first type is the only argument, everything else is a functional return type.

Similarly, function application is left-associative. This means that instead of writing:

```
six :: Int
six = ((addThree 1) 2) 3
```

We can write the less-noisy:

```
six :: Int
six = addThree 1 2 3
```

And it has the same meaning as well.

Partial Application

I mentioned partial application in an earlier chapter, but it's worth discussing again in the context of the `map` example above. We can partially apply functions by supplying only some of their arguments and getting back another function which accepts any arguments we left out. Technically, this is not "partial" at all, since all functions really only take a single argument. In fact, this mechanism happens even in the cases you wouldn't conceptually refer to as "partial application":

When we wrote the following expression:

```
maybeName = fmap userUpperName (findUser someId)
```

What really happened is `fmap` was first applied to the function `userUpperName` to return a new function of type `Maybe User -> Maybe String`.

```
fmap :: (a -> b) -> Maybe a -> Maybe b

userUpperName :: (User -> String)

fmap userUpperName :: Maybe User -> Maybe String
```

This function is then immediately applied to `(findUser someId)` to ultimately get that `Maybe String`.

Applicative

Follow The Types

We can start by trying to do what we want with the only tool we have so far: `fmap`.

What happens when we apply `fmap` to `User`? It's not immediately clear because `User` has the type `String -> String -> User` which doesn't line up with `(a -> b)`. Fortunately, it only *appears* not to line up. Remember, every function in Haskell takes one argument and returns one result: `User`'s actual type is `String -> (String -> User)`. In other words, it takes a `String` and returns a function, `(String -> User)`. In this light, it indeed lines up with the type `(a -> b)` by taking `a` as `String` and `b` as `(String -> User)`.

By substituting our types for `f`, `a`, and `b`, we can see what the type of `fmap User` is:

```
fmap :: (a -> b) -> f a -> f b

--      a      ->  b
User :: String -> (String -> User)

--           f     a      -> f      b
fmap User :: Maybe String -> Maybe (String -> User)
```

So now we have a function that takes a `Maybe String` and returns a `Maybe (String -> User)`. We also have a value of type `Maybe String` that we can give to this function, `getParam "name" params`:

```
getParam "name" params             :: Maybe String

fmap User                          :: Maybe String -> Maybe (String -> User)

fmap User (getParam "name" params) ::                 Maybe (String -> User)
```

The `Control.Applicative` module exports an operator synonym for `fmap` called `(<$>)` (I pronounce this as *fmap* because that's what it's a synonym for). The reason this synonym exists is to get us closer to our original goal of making expressions look as if there are no `Maybe`s involved. Since operators are placed between their arguments, we can use `(<$>)` to rewrite our above expression to an equivalent one with less noise:

```
User <$> getParam "name" params :: Maybe (String -> User)
```

This expression represents a "`Maybe` function". We're accustom to *values* in a context: a `Maybe Int`, `Maybe String`, etc; and we saw how these were *functors*. In this case, we have a *function* in a context: a `Maybe (String -> User)`. Since functions are things that *can be applied*, these are called *applicative functors*.

By using `fmap`, we reduced our problem space and isolated the functionality we're lacking; functionality we'll ultimately get from `Applicative`:

We have this:

```
fmapUser :: Maybe (String -> User)
fmapUser = User <$> getParam "name" params
```

And we have this:

```
aMaybeEmail :: Maybe String
aMaybeEmail = getParam "email" params
```

And we're trying to ultimately get to this:

```
userFromParams :: Params -> Maybe User
userFromParams params = fmapUser <*> aMaybeEmail
```

We only have to figure out what that `(<*>)` should do. At this point, we have enough things defined that we know exactly what its type needs to be. In the next section, we'll see how its type pushes us to the correct implementation.

Apply

The `(<*>)` operator is pronounced *apply*. Specialized to `Maybe`, its job is to apply a `Maybe` function to a `Maybe` value to produce a `Maybe` result.

In our example, we have `fmapUser` of type `Maybe (String -> User)` and `aMaybeEmail` of type `Maybe String`. We're trying to use `(<*>)` to put those together and get a `Maybe User`. We can write that down as a type signature:

```
(<*>) :: Maybe (String -> User) -> Maybe String -> Maybe User
```

With such a specific type, this function won't be very useful, so let's generalize it away from `String`s and `User`s:

```
(<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
```

This function is part of the `Applicative` type class, meaning it will be defined for many types. Therefore, its actual type signature is:

```
(<*>) :: f (a -> b) -> f a -> f b
```

Where `f` is any type that has an `Applicative` instance (e.g. `Maybe`).

It's important to mention this because it is the type signature you're going to see in any documentation about `Applicative`. Now that I've done so, I'm going to go back to type signatures using `Maybe` since that's the specific instance we're discussing here.

The semantics of our `(<*>)` function are as follows:

- If both the `Maybe` function and the `Maybe` value are present, apply the function to the value and return the result wrapped in `Just`

- Otherwise, return `Nothing`

We can translate that directly into code via pattern matching:

```
(<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
Just f <*> Just x = Just (f x)
_      <*> _      = Nothing
```

With this definition, and a few line breaks for readability, we arrive at our desired goal:

```
userFromParams :: Params -> Maybe User
userFromParams params = User
    <$> getParam "name" params
    <*> getParam "email" params
```

The result is an elegant expression with minimal noise. Compare *that* to the staircase we started with!

Not only is this expression elegant, it's also safe. Because of the semantics of `fmap` and `(<*>)`, if any of the `getParam` calls return `Nothing`, our whole `userFromParams` expression results in `Nothing`. Only if they all return `Just` values, do we get `Just` our user.

As always, Haskell being referentially transparent means we can prove this by substituting the definitions of `fmap` and `(<*>)` and tracing how the expression expands given some example `Maybe` values.

If the first value is present, but the second is not:

```
User <$> Just "Pat" <*> Nothing
-- => fmap User (Just "Pat") <*> Nothing        (<$> == fmap)
-- => Just (User "Pat")      <*> Nothing        (fmap definition, first pattern)
-- => Nothing                                   (<*> definition, second pattern)
```

If the second value's present but the first is not:

```
User <$> Nothing <*> Just "pat@thoughtbot.com"
-- => fmap User Nothing <*> Just "pat@thoughtbot.com"
-- => Nothing           <*> Just "pat@thoughtbot.com"   (fmap, second pattern)
-- => Nothing
```

Finally, if both values are present:

```
User <$> Just "Pat" <*> Just "pat@thoughtbot.com"
-- => fmap User (Just "Pat") <*> Just "pat@thoughtbot.com"
-- => Just (User "Pat")      <*> Just "pat@thoughtbot.com"
-- => Just (User "Pat" "pat@thoughtbot.com")            (<*>, first pattern)
```

Chaining

One of the nice things about this pattern is that it scales up to functions with (conceptually) any number of arguments. Imagine that our `User` type had a third field representing their age:

```
data User = User String String Int
```

Since our `getParam` function can only look up parameters of `String` values, we'll also need a `getIntParam` function to pull the user's age out of our `Params`:

```
getIntParam :: String -> Params -> Maybe Int
getIntParam = undefined
```

With these defined, let's trace through the types of our applicative expression again. This time, we have to remember that our new `User` function is of type `String -> (String -> (Int -> User))`:

```
User :: String -> (String -> (Int -> User))
```

```
User <$> getParam "name" params :: Maybe (String -> (Int -> User))
```

```
User <$> getParam "name" params <*> getParam "email" params :: Maybe (Int -> User)
```

This time, we arrive at a `Maybe (Int -> User)`. Knowing that `getIntParam "age" params` is of type `Maybe Int`, we're in the exact same position as last time when we first discovered a need for `(<*>)`. Being in the same position, we can do the same thing again:

```
userFromParams :: Params -> Maybe User
userFromParams params = User
    <$> getParam "name" params
    <*> getParam "email" params
    <*> getIntParam "age" params
```

As our pure function (`User`) gains more arguments, we can continue to apply it to values in context by repeatedly using `(<*>)`. The process by which this happens may be complicated, but the result is well worth it: an expression that is concise, readable, and above all safe.

Other Types

IO

Other instances

Unlike previous chapters, here I jumped right into `Monad`. This was because there's a natural flow from imperative code to monadic programming with do-notation, to the underlying expressions combined with `(>>=)`. As I mentioned, this is the only way to combine `IO` values. While `IO` does have instances for `Functor` and `Applicative`, the functions in these classes (`fmap` and `(<*>)`) are defined in terms of `return` and `(>>=)` from its `Monad` instance. For this reason, I won't be showing their definitions. That said, these instances are still useful. If your `IO` code doesn't require the full power of monads, it's better to use a weaker constraint. More general programs are better and weaker constraints on what kind of data your functions can work with makes them more generally useful.

Functor

`fmap`, when specialized to `IO`, has the following type:

```
fmap :: (a -> b) -> IO a -> IO b
```

It takes a function and an `IO` action and returns another `IO` action which represents applying that function to the *eventual* result returned by the first.

It's common to see Haskell code like this:

```
readInUpper :: FilePath -> IO String
readInUpper fp = do
    contents <- readFile fp

    return (map toUpper contents)
```

All this code does is form a new action that applies a function to the eventual result of another. We can say this more concisely using `fmap`:

```
readInUpper :: FilePath -> IO String
readInUpper fp = fmap (map toUpper) (readFile fp)
```

As another example, we can use `fmap` with the Prelude function `lookup` to write a safer version of `getEnv` from the `System.Environment` module. `getEnv` has the nasty quality of raising an exception if the environment variable you're looking for isn't present. Hopefully this book has convinced you it's better to return a `Maybe` in this case. The `lookupEnv` function was eventually added to the module, but if you intend to support old versions, you'll need to define it yourself:

```
import System.Environment (getEnvironment)

-- lookup :: Eq a => a -> [(a, b)] -> Maybe b
--
-- getEnvironment :: IO [(String, String)]

lookupEnv :: String -> IO (Maybe String)
lookupEnv v = fmap (lookup v) getEnvironment
```

Applicative

Imagine a library function for finding differences between two strings:

```
data Diff = Diff [Difference]
data Difference = Added | Removed | Changed

diff :: String -> String -> Diff
diff = undefined
```

How would we run this code on files from the file system? One way, using `Monad`, would look like this:

```
diffFiles :: FilePath -> FilePath -> IO Diff
diffFiles fp1 fp2 = do
    s1 <- readFile fp1
    s2 <- readFile fp2

    return (diff s1 s2)
```

Notice that the second `readFile` does not depend on the result of the first. Both `readFile` actions produce values that are combined *at-once* using the pure function `diff`. We can make this lack of dependency explicit and bring the expression closer to what it would look like without `IO` values by using `Applicative`:

```
diffFiles :: FilePath -> FilePath -> IO Diff
diffFiles fp1 fp2 = diff <$> readFile fp1 <*> readFile fp2
```

As an exercise, try breaking down the types of the intermediate expressions here, like we did for `Maybe` in the Follow the Types sub-section of the Applicative chapter.