Maybe Haskell

by Pat Brisbin

thoughtbot

Maybe Haskell

Pat Brisbin

February 16, 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 inconvenience. 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 encode in a value's type the fact that it may not be present. Any partial function can be made total by always returning a valid value, but returning one that also indicates if it is actually present or not. 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 e-book.

Haskell is not the only language to have such a construct. 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. We then assign this function the name `countEvens`.

This is a relatively contrived example, but it's indicative of the sort of thing 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 be spending 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 need exactly 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, one that is often swept under the rug by languages supporting `null`, and put it right in front of us by naming it and requiring we deal with it at every step. Haskell is not a language of short-cuts, and that's a good thing.

From there, we'll walk through 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 three *interfaces* are very important in Haskell. They're used by a number of concrete types and 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`, I'll discuss other types which share these qualities. This is to reinforce the fact that these abstractions are only that: 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, to name a few.

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 the more interesting topics that 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 be going into how to set up a Haskell programming environment, showing you how to write and run complete Haskell programs, or diving 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 for learning Haskell:

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
```

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. 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. For example, if we omitted the type signatures in our first example, the compiler would assign the type `five :: Num a => a` which means that the type of `five` is `a`ny type that's `Num`eric in nature – i.e. it can be added, negated and so forth.

Inferred types, as these are called, are usually fine but can open you up to unclear error messages. If you were to take the result of our generalized `five` and use it in two places, treating it as an `Int` in one and a `Float` in another, the error message may not immediately identify the problem depending on which expression the compiler sees first. On the other hand, if you explicitly state the type as whichever one you want it to be, then using it incorrectly will produce an error message leading you directly to the line and character of your error.

Operators

One potential source of confusion when seeing Haskell for the first time is the use of *operators*. This section intends to explain these particular kinds of functions so their use is not so surprising. Hopefully, you'll see that they are not special built-in syntax, they're functions like anything else. All there is to operators is that they can be used in a few special (and useful) ways.

This section is an attempt to avoid later confusion. We're going to make use of operators throughout the rest of the book and stopping to explain them at their first or every site of use would detract from the actual topic being covered. If you find this section to be dense or unimportant at this time, feel free to skim it now and refer back to it as needed when you come across operators you find confusing later.

The Haskell Report states that any functions made up entirely of punctuation (where "punctuation" is defined very exactly in the linked report) can be referred to as an operator. Operators can be defined and used in all the same ways as any other function, but have the following three additional behaviors:

Operators are placed between their arguments

This is called *infix notation*. It's very intuitive and looks like this:

```
2 + 2
-- => 4
```

Functions are usually called using *prefix notation*. It's not common, but if you do want to call an operator using prefix notation, you have to surround it in parenthesis:

```
(+) 2 2
-- => 4
```

You also have to surround operators with parenthesis if passing them as an argument to another function, as in this `sum` example:

```
sum = foldl (+) 0
```

Without the parenthesis, Haskell would think you were trying to add `foldl` and `0` which is most certainly a type error.

Operators can be assigned a *fixity*

An operator's *fixity* is its associativity and precedence relative to other operators. By default, function application binds most tightly and associates to the left. This means the following expression:

```
2 + 2 * 6
```

would normally be parsed as

```
(((2 +) 2) * 6)
-- => 24
```

and that would not give the correct result by the normal rules of mathematics which state that multiplication should occur before addition. We could disambiguate this with explicit parenthesis:

```
2 + (2 * 6)
-- => 14
```

This is tedious, noisy, and it's better to say once that `(*)` binds more tightly than `(+)`. We'll see examples of declaring exactly that a little later on.

Operators can be used in a *section*

If we surround an operator and one of its arguments in parenthesis, we get a function that will accept the missing argument. We can also choose freely which argument to leave out. This can be seen in the following two expressions:

```
map (/ 10) [100, 200, 300]
-- => [10, 20, 30]

map (10 /) [10, 5, 1]
-- => [1, 2, 10]
```

There is one gotcha to be aware of: the `(-)` operator. This operator can't do a right section since Haskell syntax interprets `(-2)` as the number *negative two*. The function `subtract` is available to work around this limitation. If you ever need to use `(- n)`, you can use `(subtract n)` instead.

As a final note, all of the things I've described here can also be used for any normally-named Haskell function by surrounding it in backticks:

```
-- Normal usage of an elem function
elem needle haystack

-- Reads a little better infix
needle `elem` haystack

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

Now that we understand what operators are, let's explore two such operators whose importance in writing Haskell can't be overstated: `($)` and `(.)`.

Function Application

`($)` is an operator which represents applying a function to an argument. Its complete and deceptively simple definition is the following:

```
infixr 0 $
```

```
($) :: (a -> b) -> a -> b
```

```
f $ x = f x
```

First, the `infixr` statement says that this operator associates to the right and has the lowest precedence possible.

Next, the type signature is given. Since the operator is appearing alone, we surround it with parenthesis. It takes a function from `a` to `b` and value of type `a`. It returns a value of type `b`, presumably applying the first argument to the second to produce it. It's worth noting that this is a very reasonable presumption because that's the only definition possible. This is the first of many benefits of Haskell's strict type system that you'll see: there is only one valid definition for a function of this type.

Finally, we get the expected definition. Operators can appear between their arguments even during definition which can often improve readability, as it does here. We can see that `f $ x` *is* (remember `=` means equivalence, wherever you see the expression on the left it *is* the expression on the right) the application `f x`. It can be useful to pronounce expressions using this operator with the word *of*: f of x is the application `f x`.

Technically speaking, the application `f x` should itself be pronounced "f of x", so this definition could be read as "f of x is f of x". In fact, that's exactly right as `($)` is an identity function. If you're not convinced, try exploring the following Haskell code:

```
id :: a -> a
id x = x
```

```
-- Adding explicit parenthesis shows how ($)'s type is the same but specialized
-- to a specific "shape" of a
($) :: (a -> b) -> (a -> b)

-- This means we could use id as ($)
id (+2) 2
-- => 4
```

At first, it can be hard to see why such a circular definition has any use at all. Having this function available comes with (at least) two very real benefits:

We can speak precisely about function application itself

Function application is one of the most important things you can do in Haskell. For this reason, the authors of the language ensured the doing so was free of any unnecessary syntax: no parenthesis, no commas, put the function name next to its argument, `f x`, that's it. This is great, but sometimes it's useful to name this phenomenon.

Take the following Haskell code:

```
-- A list of functions (using sections)
fs = [(+1), (+2), (+3)]

-- A list of values
xs = [1, 2, 3]

-- Zip the lists together by calling ($) with an element of each list as we go
zipWith ($) fs xs
-- => [2, 4, 6]
```

Pretty neat right? Here's another example using `($)` in a section:

```
-- A list of predicate functions (also using sections)
ps = [even, (< 10), (> 0)]
```

```
-- Check if a number matches all of those conditions
check x = all ($ x) ps

check 2  -- => True
check 5  -- => False
check 12 -- => False
```

We can get rid of parenthesis

Take the following expression:

```
take 2 drop 5 filter even [1..]
```

This expression is a type error, because Haskell's default precedence misaligns with our intention:

```
(((take 2) drop 5) filter even) [1..]
```

Our intention was for the applications to "go the other way". We can state that explicitly through parenthesis and get this to compile:

```
take 2 (drop 5 (filter even [1..]))
-- => [12, 14]
```

That looks awfully Lispy. Remember the fixity declaration given to `($)`? It associates to the right, and has the lowest possible precedence. That's exactly what we need in this case (and in fact many cases):

```
take 2 $ drop 5 $ filter even [1..]
-- => [12, 14]
```

take 2 of drop 5 of filter even 1..

Much better.

Function Composition

`(.)` is a function which denotes *composing* two functions together. This means to create a new function representing applying one function *after* another. As a quick example, if `appendX` takes a string and appends an "X" on the end, and `appendY` does the same, but with a "Y", then `appendY . appendX` can be read as *append y after append x* and represents a function that takes a string and appends "XY" on the end.

Its complete, and again deceptively simple, definition is as follows:

```
infixr 9 .

(.) :: (b -> c) -> (a -> b) -> a -> c

(.) f g = \x -> f (g x)
```

This is the definition you'll find if you look it up in the Haskell Prelude. It can be a little hard to parse if you're not used to anonymous functions and it doesn't take advantage of the fact that operators can be placed between their arguments when being defined too. For these reasons, I'd like to use an alternate but equivalent definition:

```
(f . g) x = f (g x)
```

Going back to the fixity declaration, we can see that `(.)` also associates to the right, but that it's given the highest precedence possible. The reason should make sense once we see how it works.

`(.)`'s type reads best if we imagine it taking two arguments and returning a function:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

Because of how `(->)` itself associates, the two signatures are equivalent. This is related to an idea called *currying* which I'll talk more about later.

We can read this type as follows: given two functions, one from `b` to `c` and the other from `a` to `b`, we get back a new function this time from `a` to `c`. This is why such a high precedence is desirable. When you glue two functions together with `(.)`, the result should be treated (syntactically speaking) as a single function would.

Looking at the definition, we can see how we get from `a` to `c`: `(f . g)`, when given an `x`, means to call `g` on `x`, then call `f` on the result of that. If `g` is `(a -> b)` and we give its result to `f`, which is `(b -> c)`, ultimately we have a function going all the way from `a` to `c`.

Using functions to build other functions is another frequent and important thing to do in Haskell, so it should also be devoid of unnecessary syntax. Take the following function for trimming whitespace from a string:

```
import Data.Char (isSpace)

trim s = reverse $ dropWhile isSpace $ reverse $ dropWhile isSpace s

trim "  foo bar "
-- => "foo bar"
```

Reading right to left, we first strip the trailing whitespace, then we strip the leading whitespace by reversing the string, stripping trailing whitespace again, then reversing it back. We immediately see that the form `reverse $ dropWhile isSpace x` is repeated twice. We can reduce this repetition with a local definition:

```
trim s = f $ f s

  where
    f x = reverse $ dropWhile isSpace x
```

We've used the tiny name `f` specifically because its meaning should be clear from context, calling it anything else is unnecessary noise. Ruby developers are probably twitching in their chairs at this point, but trust me – you get used to it. Even so, there is still quite a bit of noise in this definition.

What we really mean is that `f` is `reverse` *after* `dropWhile isSpace`. There's no need to name an argument here; we're defining one function (`f`) as the composition of two others (`reverse` and `dropWhile isSpace`). Beginner Haskellers may get confused about which of `($)` or `(.)` we should use in this case. Because we would read the function as reverse-after-drop and not reverse-of-drop, we know that `(.)` is what we want:

```
trim s = f $ f s

  where
    f = reverse . dropWhile isSpace
```

Better, but still noisy. We can pull the same trick with `trim` itself:

```
trim = f . f

  where
    f = reverse . dropWhile isSpace
```

I would read this as "trim is f after f where f is reverse after drop-spaces". Even though it's "full of punctuation" and uses terse variable names, I think this Haskell code comes extremely close to expressing my intent.

Functions like `trim` and `f` are known as *point-free*. That can be a source of confusion because there's visually more "points". The reason is that the `(.)`s are not the points we're talking about. `s` and `x` were "fixed points" of the functions `trim` and `f` respectively (in a mathematical sense), so removing them makes the functions point-free.

I go through all of this for two reasons:

1. Operators can be a huge source of confusion and fear in new Haskell developers. Hopefully this section has pre-emptively reduced that a bit.
2. In later sections, we'll be learning some new operators. I wanted to outline the rules ahead of time so I don't need to stop and explain it at points when you've already got a ton of new stuff on your plate.

Well that's it for functions, next stop: data!

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
```

Here we've stated that the people in our system have a name (a `String`) and an age (an `Int`). 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 something called pattern matching.

```haskell
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 in this case 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.

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

pat :: Person
pat = PersonWithAge "Pat" 29

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

Haskell allows for multiple definitions of the same function, so long as they match different patterns. They 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*. 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.

Kinds and Parameters

Imagine we wanted to generalize this `Person` type. What if people were able to hold arbitrary things? What if what that thing is (its type) doesn't really matter, the only meaningful thing we can say about it is if it's there or not. What we had before was a *person with an age* or a *person without an age*, what we want here is a *person with a thing* or a *person without a thing*.

One way to do this is to *parameterize* the type:

```
data Person a = PersonWith String a | PersonWithout String
--          |                     |
--          |                     ` we can use it as an argument here
--          |
--          ` By adding a "type variable" here
```

Any lowercase value will do, but it's common to use `a` because it's short, and a value of type `a` can be thought of as a value of `a`ny type. Rather than hard-coding that a person has an age (or not), we can say a person is holding some thing of type `a` (or not).

Now we can construct people with or without something:

```haskell
patWithAge :: Person Int
patWithAge = PersonWith "pat" 29

patWithoutAge :: Person Int
patWithoutAge = PersonWithout "pat"
```

Notice how even in the case where I have no age, I can still specify the type of that thing which I do not have.

Functions that operate on people can choose if they care about what the person's holding or not. For example, getting someone's name shouldn't be affected by them holding something or not, so we can leave it unspecified, again using `a` to mean `a`ny type:

```haskell
getName :: Person a -> String
getName (PersonWith name _) = name
getName (PersonWithout name) = name
```

But a function which does care, must both specify the type *and* account for the case of non-presence:

```haskell
doubleAge :: Person Int -> Int
doubleAge (PersonWith _ age) = 2 * age
doubleAge (PersonWithout _) = 1 -- perhaps provide a sane default?
```

Congrats. You now completely understand parameterized types.

Maybe

Haskell's `Maybe` type should make all kinds of sense now:

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

It's a bit like `PersonWith | PersonWithout`, except 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 supports pattern matching not unlike you've seen before:

```haskell
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 have to have this sort of matching expression peppered throughout the code base, how is that better than the analogous "`nil` checks"?

Don't Give Up

The above might leave you feeling underwhelmed. That code doesn't look all that better than the equivalent Ruby:

```ruby
def find_user(uid)
  if user = all_users.detect? { |u| u.matches_id?(uid) }
    user
  else
    # what to do? error?
  end
end
```

First of all, the Haskell version is type safe. I'd put money on most Ruby developers returning `nil` from the `else` branch. The Haskell type system won't allow that and that's a good thing. I understand that without spending time programming in Haskell it's hard to see the benefits of ruthless type safety employed at every turn. I assure you it's a coding experience like no other, but I'm not here to convince you of that – at least not directly.

The bottom line is that an experienced Haskeller would not write this code this way. `case` is a code smell when it comes to `Maybe`. Almost all code using `Maybe` can be improved from a tedious `case` evaluation using one of the three abstractions I'll be exploring in this book.

Let's get started.

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 this context unchanged.

Even though it's good design 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 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 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.

About Type Classes

Haskell uses type classes for functions which may be implemented in different ways for different data types. For example, we can add or negate various kinds of numbers: integers, floating points, rational numbers, etc. To accommodate this, Haskell has a `Num` type class with functions like `(+)` and `negate` as part of it. Each concrete type (`Int`, `Float`, etc) then defines its own version of the required functions.

Being a member of a type class requires you implement any *member functions* with the correct type signatures. For example, to make `Int` a `Num` someone defined a `negate` function with the type `Int -> Int`.

Usually, but not always, there are *laws* associated with these that your implementations must satisfy. For example, if you negate a number twice, you should get back to the same number. This can be stated formally as:

```
negate (negate x) == x -- for any x
```

The first requirement is enforced by the type system. If your code compiles, you got this part right. Unfortunately, the second requirement cannot be enforced by the compiler. You'll need to verify that you got this right using tests. Most laws can be stated as *properties* and thoroughly tested with a tool like QuickCheck.

Functor

To say that a type (like `Maybe`) is a `Functor`, we have to define `fmap`. The `Functor` class definition states that it must have the following type:

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

Where `f` is the type you're instantiating as a `Functor`, which in our case is `Maybe`. We can see that `whenJust` has the correct type:

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

Many types implement `fmap`. Another notable example is `[]` (List), it's implementation is the `map` you're probably familiar with. If we remove some syntactic sugar and write the type for *a list of* `a`s as `List a` rather than `[a]`, you can confirm it too has the correct type:

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

The Functor Laws

Type class laws are a formal way of defining what it means for implementations to be "well-behaved". If someone writes a library function and says it can work with "any `Functor`", that code can rely both on that type having an `fmap` implementation, and that it operates in accordance with these laws.

For example, let's look at the first Functor law:

```
fmap id x == id x
--
-- for any value x, of type f a (e.g. Maybe a)
--
```

Where `id` is the *identity* function, one which returns whatever you give it:

```
id :: a -> a
id x = x
```

Since pure functions always give the same result given the same input, it's equally correct to say that the functions themselves must be equivalent, rather than applying them to "any `x`" and saying the results must be equivalent. For this reason, the laws are usually stated as:

```
fmap id == id
```

This law says that if we call `fmap id`, the function we get back should be equivalent to `id` itself. This is what "well-behaved" means in this context. If you think about `fmap` for `[]`, you would expect that applying `id` to every element in the list (as `fmap id` does) gives you back the same exact list, and that is exactly what you expect to get if you apply `id` directly to the list itself. I encourage you to go through the same thought exercise for `Maybe` so you can see that the law holds true for its implementation as well.

The second law has to do with order of operations, it states:

```
fmap (f . g) == fmap f . fmap g
```

Where `(.)` is a function that takes two functions and *composes* them together:

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

What this law says is that if we compose two functions together, then `fmap` the resulting function, we should get a function which behaves the same as when we `fmap` each function individually, then compose the two results.

This one's a little trickier when you're not familiar with function composition. Let's prove that this law holds for `Maybe` by walking through an example with `actuallyFive` and `notReallyFive`. If you already feel comfortable with what the law is stating and why it holds, feel free to skip this section.

First, lets define two concrete functions, `f` and `g`

```haskell
f :: Int -> Int
f = (+2)

g :: Int -> Int
g = (+3)
```

We can *compose* these two functions to get a new function, and call that `h`:

```haskell
h :: Int -> Int
h = f . g
```

Given the definition of `(.)`, this is equivalent to:

```haskell
h :: Int -> Int
h x = f (g x)
```

This new function takes a number and gives it to `(+3)`, then it takes the result and gives it to `(+2)`. The result is a function that will add 5 to its argument.

```haskell
h 5
-- => 10
```

We can give this function to `fmap` to get one that works with `Maybe` values:

```haskell
fmap h actuallyFive
-- => Just 10

fmap h notReallyFive
-- => Nothing
```

Similarly, we can give `f` and `g` to `fmap` to produce functions which can add 2 or 3 to a `Maybe Int` to produce another `Maybe Int`. The resulting functions can also be composed with `(.)` to produce a new function, `fh`:

```haskell
fh :: Maybe Int -> Maybe Int
fh = fmap f . fmap g
```

Again, given the definition of `(.)`, this is equivalent to:

```
fh :: Maybe Int -> Maybe Int
fh x = fmap f (fmap g x)
```

This function will call `fmap g` on its argument which will add 3 if the number's there or return `Nothing` if it's not, then give that result to `fmap f` which will add 2 if the number's there, or return `Nothing` if it's not. This results in a function which will add 5 to a number if it's there, or return `Nothing` if it's not:

```
fh actuallyFive
-- => Just 10

fh notReallyFive
-- => Nothing
```

You should convince yourself that `fh` and `fmap h` behave in exactly the same way. The second functor law states that this must be the case if your type is a valid `Functor`.

Because Haskell is referentially transparent, we can replace functions with their implementations freely – it may require some explicit parenthesis here and there, but the code will always give the same answer. Doing so brings us back directly to the statement of the second law:

```
(fmap f . fmap g) actuallyFive
-- => Just 10

fmap (f . g) actuallyFive
-- => Just 10

(fmap f . fmap g) notReallyFive
-- => Nothing

fmap (f . g) notReallyFive
-- => Nothing

-- Therefore:
fmap (f . g) == fmap f . fmap g
```

Not only can we take normal functions (those which operate on fully present values) and give them to `fmap` to get ones that can operate on `Maybe`, but this law states we can do so in any order. We can compose our system of functions together *then* give that to `fmap` or we can `fmap` individual functions and compose *those* together – either way, we're guaranteed the same result. We can rely on this fact whenever we use `fmap` for any type that's in the `Functor` type class.

Why Is This Useful?

OK, enough theory. Now that we know how it works, let's see how it's used. Say we have a lookup function to get from a `UserId` to the `User` for that id. Since the user may not exist, it returns a `Maybe User`:

```
findUser :: UserId -> Maybe User
findUser = undefined
```

Now imagine we have a view somewhere that is displaying the user's name in all capitals:

```
userUpperName :: User -> String
userUpperName u = map toUpper (userName u)
```

The logic of getting from a `User` to that capitalized `String` is not terribly complex, but it could be – imagine something like getting from a `User` to their yearly spending on products valued over $1,000. In our case the transformation is only one function, but realistically it could be a whole suite of functions wired together. Ideally, none of these functions should have to think about potential non-presence or contain any "nil-checks" as that's not their purpose; they should all be written to work on values that are fully present.

Given `userUpperName`, which works only on present values, we can use `fmap` to apply it to a value which may not be present to get back the result we expect with the same level of *present-ness*:

```
maybeName :: Maybe String
maybeName = fmap userUpperName (findUser someId)
```

We can do this repeatedly with every function in our system that's required to get from `findUser` to the eventual display of this name. Because of the second functor law, we know that if we compose all of these functions together then `fmap` the result, or if we `fmap` any individual functions and compose the results, we'll always get the same answer. We're free to architect our system as we see fit, but still pass along the `Maybe`s everywhere we need to.

The view template, being the only place that has to "deal with" the Maybe value, can do so in a number of ways. Here, I'm using the `fromMaybe` function to specify a default value of the empty string:

```
widget = "<span class=\"username\">" ++ fromMaybe "" maybeName ++ "</span>"
```

What's in a Map

The definition of `fmap` for `Maybe` is simple and its usage is intuitive, but 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 will 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
```

Haskell also defines `->` to be right-associative and function application to be left associative. That means we don't need to add any parenthesis unless we want to explicitly group in some other way. This means rather than writing:

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

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

We can write:

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

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

And that's why Haskell type signatures have the form they do.

Partial Application

As in 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`.

Map, Not Just for Lists

Decoupling the concept of `map` such that it's useful for anything besides lists requires looking at it though this lens of single-argument, single-result functions. Viewed this way, the generic behavior of `map` is right there in the name: map a key to a value.

```
map :: (a -> b) -> ([a] -> [b])
--     |           |
--     |           ` The value, a function that operates on lists of as and bs
--     |
--     ` The key, a function that operates on as and bs
```

We can look at `fmap` as the same, only not specialized for lists:

```
fmap :: (a -> b) -> (f a -> f b)
--      |           |
--      |           ` The value, a function that operates on f as and f bs
--      |
--      ` The key, a function that operates on as and bs
```

We can instantiate `f` as one of any number of types and get a map which takes the functional key and maps it to a different functional value, one that operates in that type's space.

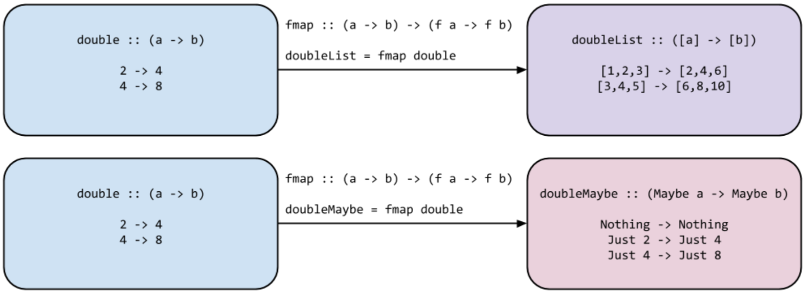

Figure 2.1: fmap

Functors come from Category theory, where they represent mapping *morphisms* from one category to another. In Haskell, *morphisms* are functions and `fmap` maps them from the "category" of `a`s and `b`s to the "category" of `f a`s and `f b`s. Practically speaking, this means that if you have a lot of Maybe values in your domain, you can use `fmap` to turn all your normal functions into more useful forms.

For a more in-depth discussion on functors, I recommend Daniel Mlot's What Does fmap Preserve?.

Applicative

In the last section we saw how to use `fmap` to take a system full of functions which operate on fully present values, free of any `nil`-checks, and employ them to safely manipulate values which may in fact not be present. This immediately makes many uses of `Maybe` more convenient, while still being explicit and safe in the face of failure and partial functions.

There's another notable case where `Maybe` can cause inconvenience, one that can't be solved by `fmap` alone. Imagine we're writing some code using a web framework. It provides a function `getParam` which takes the name of a query parameter (passed as part of the URL in a GET HTTP request), and returns the value for that parameter as parsed out of the current URL. Since the parameter you name could be missing or invalid, this function returns `Maybe`:

```
-- Don't worry about how these are represented
data Params = Params

-- Or how this function works internally. All we care about is its type.
getParam :: String -> Params -> Maybe String
getParam = undefined
```

Let's say we have a `User` data type in our system. `User`s have a name and email address, both `String`s.

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

How do we build a `User` from query params representing their name and email?

The simplest way is the following:

```
userFromParams :: Params -> Maybe User
userFromParams params =
    case getParam "name" params of
        Just name -> case getParam "email" params of
            Just email -> Just (User name email)
            Nothing -> Nothing
        Nothing -> Nothing
```

`Maybe` is not making our lives easier here. Yes, type safety is a huge implicit win, but this still looks a lot like tedious, defensive coding you'd find in any language:

```
def user_from_params(params)
  if name = get_param "name" params
    if email = get_param "email" params
      return User.new(name, email)
    end
  end
end
```

Follow the Types

`fmap` alone is not powerful enough to address this directly, but it's a start. 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)`. To reason about what happens, we have to remember that every function in Haskell really only takes one argument: `User` takes a `String` and returns a function of type `String -> User`.

```
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`. We happen to have one of those. What happens when we apply `fmap User` to `getParam "name" params`?

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

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

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

Interesting. This is a common thing to do when starting out with Haskell or even if you're experienced with Haskell but are learning a new abstraction or library: follow the types, see what fits together. Through this process, we've reduced things down to a smaller problem.

We have this:

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

And we have this:

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

And we want this:

```
userFromParams :: Params -> Maybe User
userFromParams params = x ?+? y
```

We only have to figure out what that `?+?` should be. What it looks like we need is some way to apply a `Maybe` function to a `Maybe` value to get a `Maybe` result.

Apply

We can define exactly such a function in terms of `Maybe` directly:

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

If both the function and the value are present, pull them out, apply the function, and wrap the result in `Just`. If either are missing, return `Nothing` directly.

Here's how things look when we plug it in:

```
userFromParams :: Params -> Maybe User
userFromParams params = apply
    (fmap User (getParam "name" params))
    (getParam "email" params)
```

This type checks and indeed works as expected.

We can make this expression read a bit better if we treat `apply` as an operator by surrounding it with backticks:

```
userFromParams :: Params -> Maybe User
userFromParams params =
    fmap User (getParam "name" params) `apply` getParam "email" params
```

With a bit more effort, we could use the same trick with `fmap` and end up with the following chaining:

```
userFromParams :: Params -> Maybe User
userFromParams params =
    User `fmap` getParam "name" params `apply` getParam "email" params
```

Making this form compile requires assigning the right *fixity* to `fmap` and `apply` so that Haskell knows which values are to be taken as arguments to which functions. We won't go any further down that route, as it'd be wasted effort. Instead, we'll now shift from inventing things ourselves to looking at what's already been invented – you'll see it ends up even better than the above.

Applicative Operators

Since the `apply` we made up reads best when written infix, it was defined as an operator named `(<*>)` in the `Applicative` type class. Like `fmap` it is a type class function, meaning it can be implemented by many types. There are actually two types in that type class, the first is:

```
pure :: a -> f a
```

This is relatively simple and represents taking some value and placing in the *minimal* or *default* context of `f`, whatever that means for the particular type. For `Maybe`, it is implemented as wrapping the value in `Just`. We won't be discussing this function any further since the most common way it is used can be accomplished with `fmap`, something you already know.

The second function is our friend, `apply`:

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

Both of these come with laws, but I won't be going into them. You can read all about them in the module where these functions are defined. They're more complicated than the ones for `Functor` and understanding them can come later. Generally speaking, you don't need to understand a type class's laws to effectively use types in the class, only to define instances for your own types (since then you'd have to abide by them).

The same module that defines this type class also exports an operator synonym for `fmap` named `(<$>)`. The reason for this should become clear when you see the affect the two operators have on our example:

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

Since the functions are operators, we don't need any backticks. Since they also have the correct fixity, parenthesis are not required. The result is an elegant expression with minimal noise. Compare *that* to the stair-`case` we started with!

Hiding Details

The point of all this is to write code that looks as if there is no `Maybe` involved (because that's convenient) but correctly accounts for `Maybe` at every step along the way (because that's safe). If there were no `Maybe`s involved, and we were constructing a normal `User` value, the code may look like this:

```
userFromValues :: User
userFromValues = User aName anEmail
```

Compared with a `Maybe` version:

```
userFromMaybeValues :: Maybe User
userFromMaybeValues = User <$> aMaybeName <*> aMaybeEmail
```

The resemblance is uncanny.

Applicative In the Wild

This pattern is used in a number of places in the Haskell ecosystem.

As one example, the aeson package defines a number of functions for parsing things out of JSON values, these functions return their results wrapped in a `Parser` type which is very much like `Maybe` except that it holds a bit more information about *why* the computation failed, not only *that* the computation failed. The `Applicative` instance for this type can be used to combine these sub-parsers into something domain-specific.

Again, imagine we had a rich `User` data type:

```
data User
    String       -- Name
    String       -- Email
    Int          -- Age
    UTCTime      -- Date of birth
```

We can tell aeson how to create one of these values from JSON, by implementing the `parseJSON` function which takes a JSON object (represented by the `Value` type) and returns a `Parser User`:

```
parseJSON :: Value -> Parser User
parseJSON v = User
    <$> v .: "name"
    <*> v .: "email"
    <*> v .: "age"
    <*> v .: "birth_date"
```

Each individual `v .: "..."` call is a function that attempts to pull the value for the given key out of the JSON object. Potential failure (missing key, invalid type, etc) is captured by returning a value wrapped in the `Parser` type. We can combine the individual `Parser` values together into one `Parser User` using `(<$>)` and `(<*>)`.

If any key is missing, the whole thing fails. If they're all there, we get the `User` we wanted. This concern is completely isolated within the implementation of `(<$>)` and `(<*>)`.

Monad

So far, we've seen that using this new `Maybe` type to represent failure can be very inconvenient. We addressed a number of scenarios by using `fmap` to "upgrade" a system full of normal functions (free of any `nil`-checks) into one that can take and pass along `Maybe` values. When confronted with a new scenario that could not be handled by `fmap` alone, we discovered a new function `(<*>)` which helped ease our pain again. This chapter is about addressing a third scenario, one that `fmap` and even `(<*>)` cannot solve: dependent computations.

Let's throw a monkey wrench into our `getParam` example from earlier. This time, let's say we're accepting logins by either username or password. The user can say which method they're using by passing a `type` param specifying "username" or "password".

Note: this whole thing is wildly insecure, but bear with me.

Again, all of this is fraught with `Maybe`-ness and again, writing it with straight-line `case` matches can get very tedious:

```
loginUser :: Params -> Maybe User
loginUser params = case getParam "type" of
    Just t -> case t of
      "username" -> case getParam "username" of
          Just u -> findUserByUserName u
          Nothing -> Nothing
      "email" -> case getParam "email" of
          Just e -> findUserByEmail e
          Nothing -> Nothing
```

```
        _ -> Nothing
    Nothing -> Nothing
```

Yikes.

More Power

We can't clean this up with `(<*>)` because each individual part of an `Applicative` expression doesn't have access to the results from any other part's evaluation. What does that mean? If we look at the `Applicative` expression from before:

```
User <$> getParam "name" params <*> getParam "email" params
```

Here, the two results from `getParam "name"` and `getParam "email"` (either of which could be present or not) are passed together to `User`. If they're both present we get a `Just User`, otherwise `Nothing`. Within the `getParam "email"` expression, you can't reference the (potential) result of `getParam "name"`.

We need that ability to solve our current conundrum because we need to check the value of the "type" param to know what to do next. We need… *monads*.

And Then?

Let's start with a minor refactor; let's pull out a `loginByType` function:

```
loginUser :: Params -> Maybe User
loginUser params = case getParam "type" params of
    Just t -> loginByType params t
    Nothing -> Nothing

loginByType :: Params -> String -> Maybe User
loginByType params "username" = case getParam "username" params of
    Just u -> findUserByUserName u
    Nothing -> Nothing
```

```haskell
loginByType params "email" = case getParam "email" params of
    Just e -> findUserByEmail e
    Nothing -> Nothing

loginByType _ _ = Nothing
```

Things seem to be following a pattern now: we have some value that might not be present and some function that needs the (fully present) value, does some other computation with it, but may itself fail.

Let's abstract this concern into a new function called `andThen`:

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

Again, we'll use the function infix via backticks for readability:

```haskell
loginUser :: Params -> Maybe User
loginUser params =
    getParam "type" params `andThen` loginByType params

loginByType :: Params -> String -> Maybe User
loginByType params "username" =
    getParam "username" params `andThen` findUserByUserName

loginByType params "email" =
    getParam "email" params `andThen` findUserByEmail

-- Still needed in case we get an invalid type
loginByType _ _ = Nothing
```

This cleans things up nicely. The "passing along the `Maybe`" concern is completely abstracted away behind `andThen` and we're free to describe the nature of *our* computation. If only Haskell had such a function...

Bind

If you haven't guessed it, Haskell does have exactly this function. It's name is *bind* and it's defined as part of the `Monad` type class. Here is its type signature:

```
(>>=) :: m a -> (a -> m b) -> m b
--
-- where m is the type you're saying is a Monad (e.g. Maybe)
--
```

Again, you can see that `andThen` has the correct signature:

```
--         m     a    (a -> m     b) -> m     b
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
```

`Monad` comes with another function, `return`:

```
return :: a -> m a
```

This is the same as `pure` was in `Applicative`. We take a value and place it in the *minimal* or *default* context appropriate for your type. In the case of `Maybe`, that means wrapping the value in the `Just` constructor. In fact, the Haskell language recently made it such that to be a `Monad` you must already be an `Applicative` (something that was almost always true in practice) so `return` even has a default definition of `pure`.

Chaining

`(>>=)` is defined as an operator because it's meant to be used infix. It's also given an appropriate fixity so that it can be chained together intuitively. The word `andThen` comes to mind again as having multiple dependent computations lends itself to an `x andThen y andThen z` nature. To see this in action, let's walk through another example.

Suppose we are working on a system with the following functions for dealing with users' addresses and their zip codes:

```haskell
-- Returns Maybe because the user may not exist
findUser :: UserId -> Maybe User
findUser = undefined

-- Returns Maybe because Users aren't required to have Addresses
userAddress :: User -> Maybe Address
userAddress = undefined

addressZip :: Address -> ZipCode
addressZip = undefined
```

Let's also say we have a function to calculate shipping costs by zip code. It employs `Maybe` to handle invalid zip codes:

```haskell
shippingCost :: ZipCode -> Maybe Cost
shippingCost = undefined
```

We could naively calculate the shipping cost for some user given their Id:

```haskell
findUserShippingCost :: UserId -> Maybe Cost
findUserShippingCost uid =
    case findUser uid of
        Just u -> case userAddress u of
            Just a -> case shippingCost a of
                Just c  -> Just c

                -- User has an invalid zip code
                Nothing -> Nothing

            -- Use has no address
            Nothing -> Nothing

        -- User not found
        Nothing -> Nothing
```

This code is offensively ugly, but it's the sort of code I write every day in Ruby. We might hide it behind three-line methods each holding one level of conditional, but it's there.

How does this code look with `(>>=)`?

```
findUserShippingCost :: UserId -> Maybe Cost
findUserShippingCost uid = findUser uid >>= userAddress >>= shippingCost
```

You have to admit, that's quite nice. Hopefully even more so when you look back at the definition for `andThen` to see that that's all it took to clean up this boilerplate.

Do Notation

There's one more topic I'd like to mention related to monads: *do-notation*.

This bit of syntactic sugar is provided by Haskell for any of its `Monad`s. The reason is to allow monadic code to read like imperative code when composing monadic expressions. This is valuable because monadic expressions, especially those of type `IO a`, are often best understood as a series of imperative steps:

```
f = do
    x <- something
    y <- anotherThing
    z <- combineThings x y

    return (finalizeThing z)
```

That said, this sugar is available for any `Monad` and so we can use it for `Maybe` as well. We can use `Maybe` as an example for seeing how *do-notation* works. Then, if and when you come across some `IO` expressions using *do-notation*, you won't be as surprised or confused.

De-sugaring *do-notation* is a straight-forward process followed out during Haskell compilation and can be understood best by doing it manually. Let's start with our end result from the last example. We'll translate this code step-by-step into the equivalent *do-notation* form, then follow the same process backward, as the compiler would if we had written it that way in the first place.

```
findUserShippingCost :: UserId -> Maybe Cost
findUserShippingCost uid = findUser uid >>= userAddress >>= shippingCost
```

First, to make things clearer, let's add some arbitrary line breaks:

```
findUserShippingCost :: UserId -> Maybe Cost
findUserShippingCost uid =
    findUser uid >>=
    userAddress >>=
    shippingCost
```

Next, let's name the arguments to each expression via anonymous functions, rather than relying on partial application and their curried nature. We'll also add another arbitrary line break to highlight the final expression in the chain.

```
findUserShippingCost :: UserId -> Maybe Cost
findUserShippingCost uid =
    findUser uid >>= \u ->
    userAddress u >>= \a ->

    shippingCost a
```

Next, we'll take each lambda and translate it into a *binding*, which looks a bit like variable assignment and uses `(<-)`. You can read `x <- y` as "x from y":

```
findUserShippingCost :: UserId -> Maybe Cost
findUserShippingCost uid =
    u <- findUser uid
    a <- userAddress u

    shippingCost a
```

Finally, we prefix the series of "statements" with `do`:

```
findUserShippingCost :: UserId -> Maybe Cost
findUserShippingCost uid = do
    u <- findUser uid
    a <- userAddress u

    shippingCost a
```

Et Viola, you have the equivalent *do-notation* version of our function. When the compiler sees code written like this, it follows (mostly) the same process we did, but in reverse:

Remove the `do` keyword:

```
findUserShippingCost :: UserId -> Maybe Cost
findUserShippingCost uid =
    u <- findUser uid
    a <- userAddress u

    shippingCost a
```

Translate each binding into a version using `(>>=)` and lambdas:

```
findUserShippingCost :: UserId -> Maybe Cost
findUserShippingCost uid =
    findUser uid >>= \u ->
    userAddress u >>= \a ->

    shippingCost a
```

The compiler can stop here as all remaining steps are stylistic changes only (removing whitespace and *eta-reducing*[^eta-reduce] the lambdas).

Will it Pipe?

Both notations have their place and which to use is often up to the individual developer, but I do have a personal guideline I can offer.

As mentioned, *do-notation* is typically useful in the `IO` monad where the computation is probably representing a series of dependent actions to take place in the real world. If your process is a straight pipe-line, chaining expressions together with `(>>=)` will usually read better:

```haskell
-- Read stdin, pass it to the given function, and print the result on stdout
interact :: (String -> String) -> IO ()
interact f = getContents >>= f >>= putStr

-- vs
interact f = do
    c <- getContents

    putStr (f c)
```

If instead you find yourself manipulating one result many times, *do-notation* is probably the way to go:

```haskell
-- Build a user instance, then execute some actions with it before returning
createUser :: Params -> IO User
createUser params = do
    user <- buildUser params

    storeInDatabase user
    sendConfirmationEmail user

    return user

-- vs (something like)
createUser params = buildUser params >>= \user ->
    storeInDatabase user >> sendConfirmationEmail user >> return user
```

Don't worry if you don't follow all of the new information here (i.e. `IO ()` or the `(>>)` function). These examples were only to show the differences between *do-notation* and relying only on `(>>=)` for composing monadic expressions.

Wrapping Up

And thus ends our discussion of monads. This also ends our discussion of `Maybe`. You've now seen the type itself and three of Haskell's most important abstractions which make its use convenient while still remaining explicit and safe. To highlight the point that these abstractions (`Functor`, `Applicative` and `Monad`) are *interfaces* shared by many types, the next and final section will briefly show a few other useful types which also have these three interfaces.

Other Types

The three abstractions you've seen all require a certain kind of value. Specifically, a value with some other bit of information, often referred to as its *context*. In the case of a type like `Maybe a`, the `a` represents the value itself and `Maybe` represents the fact that it may or may not be present. This potential non-presence is that other bit of information, its context.

Haskell's type system is unique in that it lets us speak specifically about this other bit of information without involving the value itself. In fact, when defining instances for `Functor`, `Applicative` and `Monad`, we were defining an instance for `Maybe`, not for `Maybe a`. When we define these instances we're not defining how `Maybe a`, a value in some context, behaves under certain computations, we're actually defining how `Maybe`, the context itself, behaves under certain operations.

This kind of separation of concerns is difficult to understand when you're only accustomed to languages that don't allow for it. I believe it's why topics like monads seem so opaque to those unfamiliar with a type system like this. To strengthen the point that what we're really talking about are behaviors and contexts, not any one specific *thing*, this chapter will explore types which represent other kinds of contexts and show how they behave under all the same computations we saw for `Maybe`.

Either

Haskell has another type to help with computations that may fail:

```
data Either a b = Left a | Right b
```

Traditionally, the `Right` constructor is used for a successful result (what a function would've returned normally) and `Left` is used in the failure case. The value of type `a` given to the `Left` constructor is meant to hold information about the failure – i.e. why did it fail? This is only convention, but it's a strong one that we'll use throughout this chapter. To see one formalization of this convention, take a look at Control.Monad.Except. It can appear intimidating because it is so generalized, but Example 1 should look a lot like what I'm about to walk through here.

With `Maybe`, the `a` was the value and `Maybe` was the context. Therefore, we made instances of `Functor`, `Applicative`, and `Monad` for `Maybe` (not `Maybe a`). With `Either` as we've written it above, `b` is the value and `Either a` is the context, therefore Haskell has instances of `Functor`, `Applicative`, and `Monad` for `Either a` (not `Either a b`).

This use of `Left a` to represent failure with error information of type `a` can get confusing when we start looking at functions like `fmap` since the generalized type of `fmap` talks about `f a` and I said our instance would be for `Either a` making that `Either a a`, but they aren't the same `a`!

For this reason, we can imagine an alternate definition of `Either` that uses different variables. This is perfectly reasonable since the variables are chosen arbitrarily anyway:

```
data Either e a = Left e | Right a
```

When we get to `fmap` (and others), things are clearer:

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

ParserError

As an example, consider some kind of parser. If the parser fails, it would be nice to include the line and character that triggered the failure. To accomplish this, we first define a type to represent information about the failure. For our purposes, we'll say it's the line and character where something unexpected appeared, but it could be much richer than that including what was expected and what was seen instead:

```haskell
data ParserError = ParserError Int Int
```

From this, we can make a domain-specific type alias built on top of `Either`. We can say that a value which we parse may fail, and if it does, there will be error information in a `Left`-constructed result. If it succeeds, we'll get the `a` we originally wanted in a `Right`-constructed result.

```haskell
--             Either e           a    = Left e           | Right a
type Parsed a = Either ParserError a -- = Left ParserError | Right a
```

Finally, we can give functions that may produce such results an informative type:

```haskell
parseJSON :: String -> Parsed JSON
parseJSON = undefined
```

This informs callers of `parseJSON` that it may fail and, if it does, the invalid character and line can be found:

```haskell
jsonString = "..."

case parseJSON jsonString of
   Right json ->               -- do something with json
   Left (ParserError ln ch) -> -- do something with the error information
```

Functor

You may have noticed that we've reached the same conundrum as `Maybe`: often, the best thing to do if we encounter a `Left` result is to pass it along to our own callers. Wouldn't it be nice if we could take some JSON-manipulating function and apply it directly to something we parse? Wouldn't it be nice if the "pass along the errors" concern were handled separately?

```haskell
-- Replace the value at the given key with the new value
replace :: Key -> Value -> JSON -> JSON
replace = undefined
```

```
--
-- This is a type error!
--
--  replace "admin" False is (JSON -> JSON), but parseJSON returns (Parsed JSON)
--
replace "admin" False (parseJSON jsonString)
```

`Parsed a` is a value in some context, like `Maybe a`. This time, rather than only present-or-non-present, the context is richer. It represents present-or-non-present-with-error. Can you think of how this context should be accounted for under an operation like `fmap`?

```
--      (a -> b) -> f        a -> f        b
--      (a -> b) -> Either e a -> Either e b
fmap :: (a -> b) -> Parsed a   -> Parsed   b
fmap f (Right v) = Right (f v)
fmap _ (Left e)  = Left e
```

If the value is there, we apply the given function to it. If its not, we pass along the error. Now we can do something like this:

```
fmap (replace "admin" False) (parseJSON jsonString)
```

If the incoming string is valid, we get a successful `Parsed JSON` result with the `"admin"` key replaced by `False`. Otherwise, we get an unsuccessful `Parsed JSON` result with the original error message still available.

Knowing that `Control.Applicative` provides `(<$>)` is an infix synonym for `fmap`, we could also use that to make this read a bit better:

```
replace "admin" False <$> parseJSON jsonString
```

Speaking of `Applicative`...

Applicative

It would also be nice if we could take two potentially failed results and pass them as arguments to some function that takes normal values. If any result fails, the overall result is also a failure. If all are successful, we get a successful overall result. This sounds a lot like what we did with `Maybe`, the only difference is we're doing it for a different kind of context.

```
-- Given two json objects, merge them. Duplicate keys result in those in the
-- second object being kept.
merge :: JSON -> JSON
merge = undefined

jsonString1 = "..."

jsonString2 = "..."

merge <$> parseJSON jsonString1 <*> parseJSON jsonString2
```

Defining `(<*>)` starts out all right: if both values are present we'll get the result of applying the function wrapped up again in `Right`. If the second value's not there, that error is preserved as a new `Left` value:

```
--       f        (a -> b) -> f        a -> f        b
--       Either e (a -> b) -> Either e a -> Either e b
(<*>) :: Parsed   (a -> b) -> Parsed   a -> Parsed   b
Right _ <*> Left e = Left e
```

Astute readers may notice that we could reduce this to one pattern by using `fmap` – this is left as an exercise.

What about the case where the first argument is `Left`? At first this seems trivial: there's no use inspecting the second value, we know something has already failed so let's pass that along, right? Well, what if the second value was also an error? Which error should we keep? Either way we discard one of them, and any potential loss of information should be met with pause.

It turns out, it doesn't matter – at least not as far as the Applicative Laws are concerned. If choosing one over the other had violated any of the laws, we would've had our answer. Beyond those, we don't know how this instance will eventually be used by end-users and we can't say which is the "right" choice standing here now.

Given that the choice is arbitrary, I present the actual definition from `Control.Applicative`:

```
Left e <*> _ = Left e
```

Monad

When thinking through the `Monad` instance for our `Parsed` type, we don't have the same issue of deciding which error to propagate. Remember that the extra power offered by monads is that computations can depend on the results of prior computations. When the context involved represents failure (which may not always be the case!), any single failing computation must trigger the omission of all subsequent computations (since they could be depending on some result that's not there). This means we only need to propagate that first failure.

Let's say that our domain has some JSON responses whose values contain HTML. Parsing these values into an `HTML` data type may also fail. Since our `Parsed` type is polymorphic in its result (i.e. it's `Parsed a` not `Parsed JSON`), we can reuse it here:

```
parseHTML :: Value -> Parsed HTML
parseHTML = undefined
```

We can directly parse a `String` of JSON into the HTML present at one of its keys by binding the two parses together with `(>>=)`:

```
-- Grab the value at the given key
at :: Key -> JSON -> Value
at = undefined

parseBody :: String -> Parsed HTML
parseBody jsonString = parseJSON jsonString >>= parseHTML . at "body"
```

First, `parseJSON jsonString` gives us a `Parsed JSON`. This is the `m a` in `(>>=)`'s type signature. Then we use `(.)` to compose a function for getting the value at the `"body"` key and passing it to `parseHTML`. The type of this function is `(JSON -> Parsed HTML)` which aligns with the `(a -> m b)` of `(>>=)`'s second argument. Knowing that `(>>=)` will return `m b`, we can see that that's the `Parsed HTML` we're after.

If both parses succeed, we get a `Right`-constructed value containing the `HTML` we want. If either parse, fails we get a `Left`-constructed value containing the `ParserError` from whichever failed.

Allowing such a readable expression (*parse JSON and then parse HTML at body*), requires the following straight-forward implementation for `(>>=)`:

```haskell
--         m        a -> (a -> m        b) -> m        b
--         Either e a -> (a -> Either e b) -> Either e b
(>>=) :: Parsed   a -> (a -> Parsed   b) -> Parsed   b
Right v >>= f = f v
Left e >>= _ = Left e
```

Armed with instances for `Functor`, `Applicative`, and `Monad` for both `Maybe` and `Either e`, we can use the same set of functions (those with `Functor f`, `Applicative f` or `Monad m` in their class constraints) and apply them to a variety of functions which may fail (with or without useful error information).

This is a great way to reduce a project's maintenance burden: if you start with functions returning `Maybe` values but use generalized functions for (e.g.) any `Monad m`, you can later upgrade to a fully fledged `Error` type based on `Either` without having to change most of the code base.