Extracted from:

Effective Haskell

Solving Real-World Problems with Strongly Typed Functional Programming

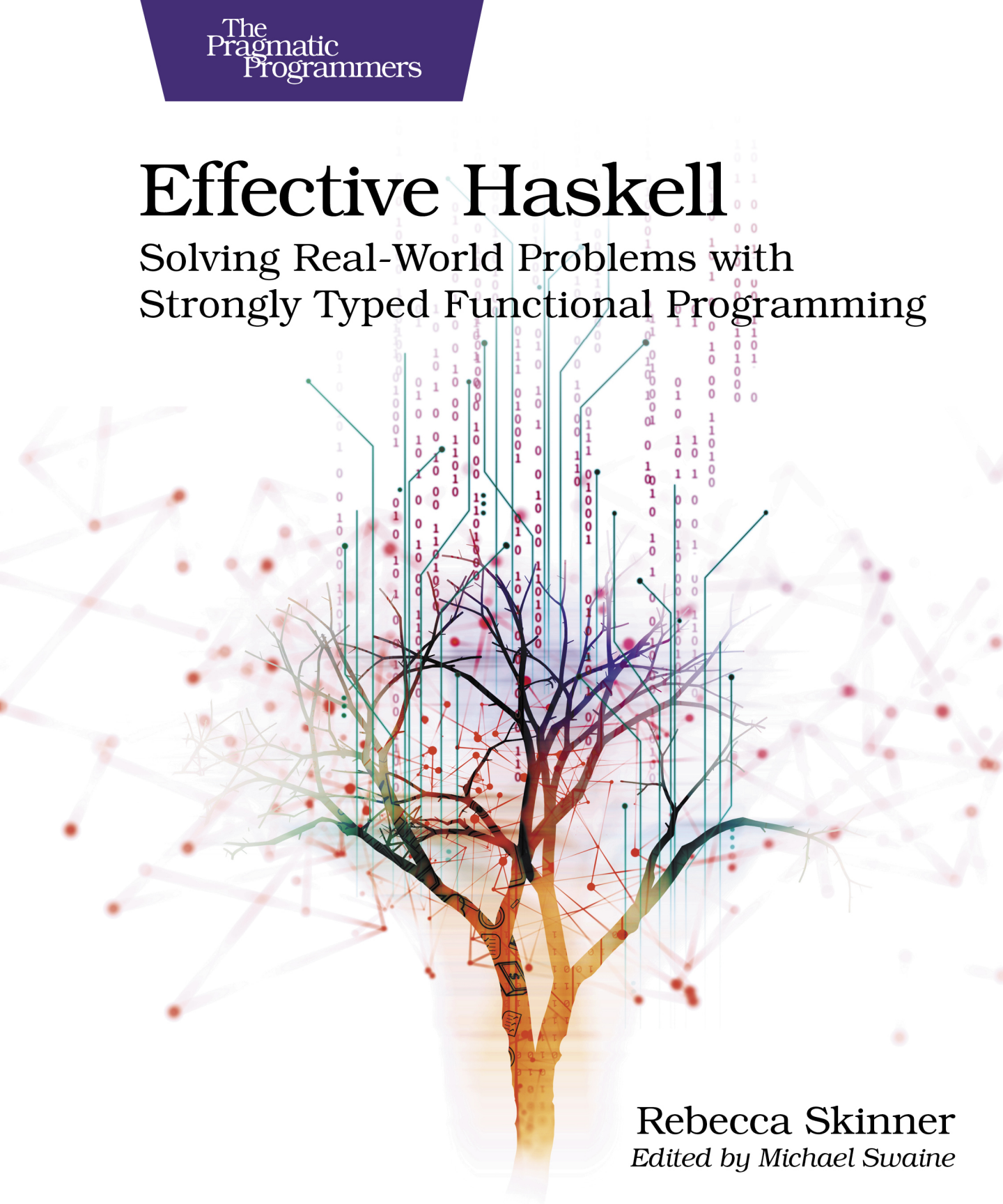
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Effective Haskell

Solving Real-World Problems with Strongly Typed Functional Programming



Rebecca Skinner

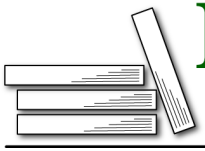
Edited by Michael Swaine

Effective Haskell

Solving Real-World Problems with Strongly Typed Functional Programming

Rebecca Skinner

The Pragmatic Bookshelf
Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Michael Swaine
Copy Editor: Karen Galle
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-934-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2023

Specifying Type Class Instances with Type Applications

A common problem when you're using type class constraints is ambiguity about which specific type class instance should be used. Consider, for example, a program you want to have print out the additive and multiplicative identities of some `Natural` numbers. You might write something like this:

```
showIdentities =
  let mul = multiplicativeIdentity
      add = additiveIdentity
      msg = "The additive identity is: "
            <> show add
            <> " and the multiplicative identity is: "
            <> show mul
  in print msg
```

Unfortunately, this fails to compile! The problem is that multiplicativeIdentity and additiveIdentity both return a type that depends on the type class instance that we're using, but the compiler doesn't have a way to pick any particular instance, and so it has to give up and raise an error. One way we could get around this for our example function is to add a type annotation:

```
showIdentities =
  let mul = multiplicativeIdentity :: Peano
      add = additiveIdentity :: Peano
      msg = "The additive identity is: "
            <> show add
            <> " and the multiplicative identity is: "
            <> show mul
  in print msg
```

This gets us past our error, but it's not an ideal solution. The first problem is that we're assuming that the return type of the function is sufficient to tell the compiler which type class to use. It works out for our small example here, but if the return type of the function had been polymorphic, we'd be back in the same situation. The second problem is that type annotations can be a little syntactically awkward in some places, especially in pointfree code. It would be ideal in cases like this if we could directly tell the compiler which type class to use, just like we did when we passed in a value of our original Natural record type.

TypeApplications

The TypeApplications extension has been available since GHC 8.0.1. This extension is enabled by default in GHC2021 but you'll need to enable it manually if you are using Haskell2010. This is a safe extension, and shouldn't introduce any problems with existing code.

The TypeApplications language extension allows us to do exactly that. Type applications gives you the ability to pass type names as arguments to polymorphic functions, to select the type class instance that's used. To see it in action, let's start up a ghci session. TypeApplications is enabled with GHC2021, but

if you're using a version of GHC older than 9.0, you'll need to enable the extension manually:

```
λ :set -XTypeApplications
```

With the language extension enabled we can use `@TypeName` to pass a type name into a polymorphic function. A good way to see this quickly is by using `read`. The `read` function has type `read :: Read a => String -> a`, and so by controlling the `Read` instance it uses to parse the string, we can control the return type. Let's run through a few examples:

```
λ read @Integer "1"
1
λ read @Float "1"
1.0
```

You can see in these examples how the output of the function call depends only on the type parameter. You can partially apply type applications as well, just like regular arguments:

```
λ readInt = read @Int
λ readFloat = read @Float
λ :type readInt
readInt :: String -> Int
λ :type readFloat
readFloat :: String -> Float
λ
```

You can use multiple type applications in functions that have more than one variable with a type class constraint. For example, let's write a function that takes a string and returns an `Either` value that depends on the length of the input:

```
showLeftRight :: (Read a, Read b) => String -> Either a b
showLeftRight s
  | length s > 5 = Left (read s)
  | otherwise = Right (read s)
```

Just like before, we'll need to use type applications to tell the compiler which instance of the `Read` type class to use, but now we have two type variables to work with, `a` and `b`. We'll use two type applications; the first will select the type to use for `a`, and the second will select the type to use for `b`:

```
λ showLeftRight @Float @Int "3.1415"
Left 3.1415
λ showLeftRight @Float @Int "321"
Right 321
```

You'll notice that since we're using an `Either` here, only one of the two type applications will ever be relevant. If we're returning a `Left` value, we don't care

about the second type variable's instance, since we'll never use it. In that case, you can just provide one type application:

```
λ showLeftRight @Float "3.1415"
Left 3.1415
```

If you only want to provide the second type, you can use @_ as a placeholder. This allows us to skip type applications when they aren't relevant, so for example, if you know that you'll only be using the Right constructor you can say:

```
λ showLeftRight @_ @Int "123"
Right 123
```

Type applications themselves can also be polymorphic. Using polymorphic type applications allows you to create some types of abstractions that would otherwise be difficult to express. Understanding how these work will be easier when working in a source file, since we'll be wanting to write type annotations for functions, so create a new file. In addition to TypeApplications, we'll need to enable another extension, ScopedTypeVariables. We'll look at the new features that this extension enables as we're working through the examples.

ScopedTypeVariables

The ScopedTypeVariables extension has been available since GHC 6.8.1. It's enabled by default in GHC2021 but you'll need to enable it manually if you are using Haskell2010. This extension changes the way type checking works, and may cause some existing programs to stop compiling. It may be beneficial to consider trying to enable this extension project wide in Haskell2010 codebases to identify any problems before upgrading to GHC2021. This extension implies ExplicitForAll. If you are using ScopedTypeVariables you don't need to manually enable ExplicitForAll.

```
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TypeApplications    #-}
```

Next, let's write a function that will guarantee that the Read and Show instances behave as we expect. One of the generally implied contracts about the behavior of these two type classes is that when we show a value, and then read it, we should get the original value back. We can start testing this by writing a function like adheresToReadShowContract:

```
adheresToReadShowContract val =
  let a  = show . read . show $ val
      b = show val
  in a == b
```

Unfortunately, a construct like `show . read . show` is too much for GHC to be able to handle with type inference, and we'll get a couple of errors where the compiler tells us that it can't figure out what type it should use to instantiate the type class instances. If you haven't already, try to compile your code so that you can see the error yourself.

We could solve this problem by using explicit type application to provide some type like `Int` or `Bool` or whatever to `read`, but that is overly restrictive. One of the benefits of our function is that right now it should allow us to test any type that has a `Read` and a `Show` instance. We don't want to give that up!

If our program compiled, we would expect the type signature for it to be something like:

```
adheresToReadShowContract :: (Read a, Show a) => a -> Bool
```

We'd like to be able to tell GHC that, whatever type it uses to instantiate `a`, that should also be the instance that it uses for the calls to `read` and `show`. To do that we'll need to use some syntax that is available thanks to the `ScopedTypeVariables` extension that we've added. Let's take a look at the code first and then break down what's happening:

```
adheresToReadShowContract :: forall a. (Read a, Show a) => a -> Bool
adheresToReadShowContract val =
  let a = show . read @a . show $ val
      b = show val
  in a == b
```

The first thing you'll notice is that we've added a new element to our type signature, `forall a.`. The use of `forall` here is introducing explicit *universal quantification*. This isn't a term you'll often need to use, except perhaps when reading some specific GHC documentation. More generally, it's simply referred to as *explicit forall*. In the code that you've written so far, the `forall` has been implied when you've used type variables. Writing it explicitly will not generally change the way your program works, but with the `ScopedTypeVariables` language extension, using an explicit `forall` brings the type variables into scope in the body of the function. That means that we can refer to the type variable when we're using explicit type applications.

Since our use of `ScopedTypeVariables` has allowed us to bring our type variable `a` into scope, we can apply it to `read`, which gives GHC enough information to successfully compile the program.