The Book of Types

Type-Level Programming in Haskell

Sandy Maguire

Contents

NOTICE

This document is a very early pre-release of what is planned to be a much longer book on type-level programming in Haskell. It's being shared to gauge interest in such a project, as well as to receive early feedback on its style and tone.

Please feel free to share it!

Any questions that you have while reading this book are of particularly high interest to me. Every line of prose is numbered. If you'd be willing to share these questions and the line number they struck you, I'd be forever grateful to you.

My email is `sandy@sandymaguire.me`, and with your help, we can make this book the best it can possibly be.

Thanks for your time,

Sandy Maguire

Chapter 1

Introduction

1.1 The Value of Types

Haskellers are an odd sort of folk. Most of us, I'd suspect, have spent at least one evening of our lives trying to extol the virtues of a strong typesystem to a dynamically typed colleague. They'll say things like "I like Ruby because the types don't get in my way." Though as proponents of strong typing systems, our first instinct might be to forcibly connect our head to the table, I think this is a criticism worth keeping in mind.

We Haskellers certainly have strong opinions about the value our types. They *are* useful, and *do* carry their weight in gold when coding and when refactoring. While we can dismiss our colleague's complaints with a wave of the hand and the justification that they've never seen a "type system" more powerful than Java's, types *often do* get in the way. We've just learned to blind ourselves to these shortcomings rather than bite the bullet and consider that maybe types aren't necessarily the perfect solution to every problem.

Simon Peyton-Jones, one of the primary authors of Haskell, is quick to acknowledge the fact that there are plenty of error-free programs that are ruled out by a type system. Consider, for example, a program which has a type-error, but never actually evaluates it:

```
fst ("no problems", True <> 17)
```

Evaluation of such an expression will happily produce `"no problems"` at runtime, despite the fact that we consider it to be "ill-typed." The usefulness of this example is admittedly low, but the point stands; types often do get in the way of perfectly reasonable programs.

Sometimes this obstruction comes under the guise of "it's not clear what type this thing should have." One particularly poignant case of this is thee C function `printf`.

```
int printf (const char *format, ...)
```

If you've never had the pleasure of using `printf`, it takes a "format string" as its first parameter, parses it, and depending on its contents, will conditionally read additional parameters. For example, the format string `"hello %s"` takes an additional string, and will interpolate it in place of the `%s`. Likewise, the specifier `%d` describes interpolation of a signed decimal integer.

The following calls to `printf` are all valid:

- `printf("hello %s", "world") // hello world`
- `printf("%d + %d = %s", 1, 2, "three") // 1 + 2 = three`
- `printf("no specifiers") // no specifiers`

On first inspection, it's not at all clear what a strongly-typed `printf` should have. Despite this, it's inarguably a *useful* function.

The documentation for `printf` is quick to mention that the format string should not be provided by the user; doing so opens vulnerabilities where attackers can craft a specialized format string that exploits these specifiers and gain access to the system. Indeed, this is often the first homework assignment in any university-level course on software security.

To be clear, the vulnerabilities in `printf` occur when the format string's specifiers do not align with the additional arguments given to the function call. The following calls to `printf` both cause undefined behavior (ie. crashing or security holes):

- `printf("%d") // corrupts the stack`
- `printf("%s", 1) // reads an arbitrary amount of memory`

The problem here is that C's typesystem is insufficiently expressive to describe `printf`, and thus allows type errors to make it all the way to runtime, in the form of undefined behavior. The type of `int printf(const char *format, ...)` is unsatisfying for a number of reasons: that it's "format string" is not really a string, that it doesn't constrain the number of arguments it takes, and that it doesn't enforce anything about the types of those arguments.

In my mind, preventing security holes is a much more important aspect of the value of types than "`null` is the billion dollar mistake" or whichever other arguments are in vogue today. We will return to the problem of `printf` in Chapter 3.

With few exceptions, the prevalent attitude of Haskellers has been to dismiss the usefulness of ill-typed programs rather than to admit the uncomfortable truth that our favorite language can't do something other languages can.

But there is a way to have our cake and eat it too. Indeed, Haskell *is* capable of expressing things as oddly-typed as `printf`, for those of us willing to put in the effort to learn how. This book aims to be *the* comprehensive manual for getting you from here to there, from a competent Haskell programmer to one who convinces the compiler to do their work for them.

Chapter 2

Terms, Types and Kinds

2.1 The Kind System

It is an unfortunate fact of the world that we must walk before we can run. While most of this book will introduce new concepts as *solutions to problems*, where their uses will be motivated, we do not yet have enough vocabulary to even talk about most of the kinds of problems we'd like to be able to solve. And thus it is necessary to discuss a little about the fundamentals of type-level programming. Rest assured that we'll keep this section as brief as possible before diving into applications of the concepts we are about to learn.

In everyday Haskell programming, the fundamental building blocks are those of *terms* and *types*. Terms are the values you can manipulate, the things that exist at runtime, and the types are little more than sanity-checks: proofs to the compiler (and ourselves) that the programs we're trying to write make some amount of sense.

Completely analogously, the fundamental building blocks for type-level programming are *types* and *kinds*. Now it is the types we are manipulating, and the kinds become the proofs we use to keep ourselves honest.

The kind system, if you're unfamiliar with it, can be reasonably described as "the type system for types." By that line of reasoning, then, kinds are loosely "the types of types."

Consider the numbers `4` and `5`, both of type `Int`. As far as the compiler is concerned, we could replace every instance of `4` with `5` in our program, and the whole thing would continue to compile. The program itself might do something different, but by virtue of both being of type `Int`, `4` and `5` are interchangeable to the typechecker.

If kinds are the "types of types," a reasonable question might be "what is the kind of `Int`?" As it happens, the kind of `Int` is Type (historically written as $\star$). Sometimes we call types of kind Type *value types*.

Type is the kind of any type which has values that exist at runtime[1].

[1] The pedantic reader might notice that `Void` has kind Type, but no inhabitants. Ah, but

Other types whose kind is TYPE include `Bool`, `Maybe a` and `ExceptT String IO ()`.

However, it's often more informative to look at things which do not fall into the same pattern. Consider `Maybe`—not `Maybe a` mind—but just the type constructor itself: `Maybe`. `Maybe` isn't like `Int`; there are no terms whose type is `Maybe`, nor can we replace every instance of `Bool` in our program with `Maybe` and expect the thing to still compile. Instead what we get is a *kind error.*

The reason for this is not hard to see; `Maybe` is a type constructor – it has a "type-shaped hole" that needs to be filled in before it is meaningful. As such, we say the kind of `Maybe` is TYPE $\to$ TYPE. Whenever you give `Maybe` a type of kind TYPE, it will give you back a type of kind TYPE. We call `Maybe` a *higher-kinded type.*

It's a bit of a tricky thing to put into words, but if you've written some amount of Haskell, this should fit in with your intuitions behind how such things work. While it's fine to talk about `Maybe String`, neither `Maybe` nor `Maybe Maybe` are meaningful. That's because we're used to thinking about TYPE in our everyday Haskell experiences.

On its own, `Maybe` has the wrong kind for us to care about, and `Maybe Maybe` is meaningless: it's a *kind error* because `Maybe` wants a TYPE but we're giving it a TYPE $\to$ TYPE. This is just as nonsensical as trying to pass in the function `(+1) :: Int -> Int` as an argument to itself.

Several higher-kinded types you're already familiar with will be of kind TYPE $\to$ TYPE, including `IO` and `[ ]` (lists).

But what about `Either`? `Either` takes two type parameters, both of which must be value-types. We thus say that `Either` has kind TYPE $\to$ TYPE $\to$ TYPE: it requires two parameters of kind TYPE before it is fully saturated ("out of type variables").

The function arrow `(->)` also has kind TYPE $\to$ TYPE $\to$ TYPE, because it requires two types to be filled in before it's a saturated value type.

To take things to the next level, what is the kind of the monad transformer `MaybeT`? Well, like `Either`, it takes two type parameters, but unlike `Either`, one of those parameters must be a `Monad` (ie. not a value type.) The eliminator for `MaybeT` is `runMaybeT :: MaybeT m a -> m (Maybe a)`, and from here we can work backwards.

We know that `Maybe` has kind TYPE $\to$ TYPE, so `a` must have kind TYPE. Since we know the kind of `Maybe a` is TYPE, `m` must then have kind TYPE $\to$ TYPE. Putting it all together, the kind of `MaybeT` must then be (TYPE $\to$ TYPE) $\to$ TYPE $\to$ TYPE.

I promise, after some practice this stuff will come just as naturally to you as the typechecking rules do.

However, kinds apply to everything at the type-level, not just the things we traditionally think of as "types." For example, the type of `show` is `Show a => a -> String`. This `Show` thing exists as part of the type signature, even though it's not really a "type" in the traditional sense of the word. Does `Show a` have

---

in fact `undefined` is an inhabitant of `Void`!

a kind?

It does indeed, and that kind is Constraint. More generally, Constraint is the kind of any fully-saturated type class.

With knowledge of this, can you guess what kind `Show` by itself (unsaturated) has? If you said Type $\rightarrow$ Constraint, you're right!

What about the kind of `Functor`?[2] Of `Monad`?[3]? Of `MonadTrans`?[4]

Without further language extensions, this is the extent of the expressiveness of Haskell's kind system. As you can see, it's actually quite limited – we have no notion of polymorphism, of being able to define our own kinds, or of being able to write functions.

Fortunately, those things are the subject matter of the remainder of this book – techniques, tools and understanding for Haskell's more esoteric language extensions.

2.2 Data Kinds

By enabling Haskell's `-XDataKinds` extension, we gain the ability to talk about kinds other than Type and Constraint (and their derivatives.) In particular, `-XDataKinds` lifts data constructors into *type constructors* and types into *kinds.*

As an example, given the familiar `Bool` type definition:

```
data Bool
  = True
  | False
```

we gain the following *kind* definition [5]:

```
kind Bool
  = 'True
  | 'False
```

Which is to say that we have now declared the types `'True` and `'False`, both of kind Bool. We call `'True` and `'False` *promoted data constructors.* The leading ticks on the identifiers (the `'` in `'True`) are used to distinguish promoted data constructors from everyday type constructors, in the common case of a type with a single data constructor:

```
data Unit = Unit
```

In this example, it's very important to differentiate between the *type constructor* `Unit` (of kind Type), and the *promoted data constructor* `'Unit` (of kind Unit.) This is a subtle point, and can often lead to inscrutable compiler

[2] (Type $\rightarrow$ Type) $\rightarrow$ Constraint

[3] Likewise.

[4] (Type $\rightarrow$ Type) $\rightarrow$ Type $\rightarrow$ Constraint

[5] Note that this is not legal Haskell syntax.

errors; while it's fine to ask for values of type `Maybe Unit`, it's a *kind error* to ask for `Maybe 'Unit`—because `'Unit` is the wrong kind!

Promoted data constructors are of the wrong kind to ever exist at runtime, which raises the question "what good are they?" Without any other fancy type-level machinery, we can use them as phantom parameters to handle state transitions.

this is a shit example

Imagine a concurrent application which needs to acquire some locks, but will deadlock if it attempts to acquire the same lock twice. It would be nice to enforce this invariant at the type-level, so that the compiler will refuse to compile any code which attempt to acquire the same lock twice.

```
data LockState
  = Unlocked
  | Locked

data Lock (s :: LockState) = Lock
  { getLockId :: Int
  }

newLock :: IO (Lock 'Unlocked)
newLock = ...

withLock :: Lock 'Unlocked -> (Lock 'Locked -> IO a) -> IO a
withLock = ...
```

In this example, we use `-XDataKinds` to lift `LockState` into LockState, which we then use as a `phantom parameter` to `Lock`. When we construct a new lock via `newLock` we get back a `Lock 'Unlocked`. However, the `withLock` function takes an unlocked lock, and gives us back a locked one, ensuring we can't lock it again! We've successfully used the type system to prevent us from creating a deadlock.

Notice that in the definition of `Lock`, we use a kind signature of LockState. This requires enabling the `-XKindSignatures` extension, which is one you will always want enabled when doing type-level programming.

2.3 Type-Level Functions

Where `-XDataKinds` really begins to shine, however, is through the introduction of *closed type families*. You can think of closed type families as *functions at the type level*. Compare the function `or`, which computes the OR of two `Bool`s:

```
or :: Bool -> Bool -> Bool
or True  _ = True
or False y = y
```

While we're unfortunately unable to automatically promote term-level functions into type-level ones, we can write `or` as a closed type family (if we first remember to enable the `-XTypeFamilies` extension):

```
type family Or (x :: Bool) (y :: Bool) :: Bool where
  Or 'True  y = 'True
  Or 'False y = y
```

Line for line, the similarities between `or` and `Or` are analogous. The type family `Or` requires a capital letter for the beginning of its name, because it exists at the type-level, and besides having a more verbose "kind-signature," the two definitions proceed almost exactly in lockstep.

While the metaphor between type families and functions is enticing, it isn't entirely *correct.* The analogues break down in several ways, but the most important one is that *type families must be saturated.* Another way of saying this is that all of a type family's parameters must be specified simultaneously; there is no currying available.

For example, this means we're unable to write a useful promoted `map :: (a -> b) -> [a] -> [b]`:

Broken Code

```
type family Map (x :: a -> b) (i :: [a]) :: [b]
      where
    Map f '[]       = '[]
    Map f (x ': xs) = f x ': Map f xs

  type AllTrue = Map (Or 'True)
                   '[ 'True, 'False, 'False ]
```

Attempting to compile this gives us the following error:

```
error:
  • The type family ‘Or’ should have 2 arguments, but has been
    given 1
  • In the type synonym declaration for ‘AllTrue’
```

What the error is trying to tell us is that we used the `Or` closed type-family in a non-saturated way. We only passed it one parameter instead of the two it requires, and so unfortunately GHC refuses to compile this program.

While there is nothing preventing us from writing `Map`, its usefulness is severely limited due to our inability to curry the type family we pass into it.

Before we leave this topic, look again at our definition of `Or`. Pay close attention to its "kind-signature." We write it as `Or (x :: Bool) (y :: Bool) :: Bool`, rather than `Or x y :: Bool -> Bool -> Bool`. The kinds of type families are tricky beasts; the kind you write after the `::` is the kind of the type *returned* by the type family, *not* the kind of the type family itself.

```
type family Foo (x :: Bool) (y :: Bool) :: Bool
type family Bar x y :: Bool -> Bool -> Bool
```

Take a moment to think about the kinds of `Foo` and `Bar`. While `Foo` has kind $\textsc{Bool} \to \textsc{Bool} \to \textsc{Bool}$, `Bar` has kind `Type -> Type -> Bool -> Bool -> Bool`.

2.4 Promoting Built-In Types

With `-XDataKinds` enabled, almost all[6] types automatially promote to kinds, including the built-in ones. Since built-in types (strings, numbers, lists and tuples) are special at the term level—at least in terms of syntax—we should expect they should behave a little strange at the type level too.

When playing with these built-in promoted types, it's necessary to first import the `GHC.TypeLits` module. `GHC.TypeLits` defines the kinds themselves, as well as all of the useful type families for manipulating them. We'll cover this more in detail in a second.

Strings promote in the way you'd expect, except that at the type level they are *not* equivalent to lists of characters. Enabling `-XDataKinds` provides access to the $\textsc{Symbol}$ kind (defined in `GHC.TypeLits`), which corresponds to lifted strings. Symbol type-literals can be written as `"hello" :: Symbol`, although the kind signature is unnecessary.

Numbers are a little more odd, in that only the natural numbers $(0, 1, 2, \ldots)$ can be promoted. These natural numbers, naturally enough, are of kind $\textsc{Nat}$.

Lists promote exactly in the way they should, if they were defined as

```
data [] a
  = []
  | a : [a]
```

which is to say that when promoted, we get the following type constructors: `'[]` of kind $[\textsc{a}]$, and `':` of kind $\textsc{a} \to [\textsc{a}] \to [\textsc{a}]$. When compared against the *data* constructors of lists, `[] :: [a]` and `(:) :: a -> [a] -> [a]`, with a little concentration, it should make some sense. Because lists' data constructors have symbolic names, they require the `-XTypeOperators` extension to be enabled if you want to work with them. Don't worry though, GHC will helpfully remind you if you forget.

Note that the promoted data constructors for lists are polymorphic. We'll talk more about kind-level polymorphism later.

There is another subtle point to be noted when dealing with list-kinds. While `[Bool]` is of kind $\textsc{Type}$ and describes a term-level list of booleans, the type `'[Bool]` is of kind $[\textsc{Type}]$ and describes a type-level list with one element (namely, the type `Bool`.)

Further care should be taken when constructing a promoted list; due to the way GHC's lexer parses character literals (`'a'`), make sure you add a space after

[6] Not all types. GADTs and other "tricky" data constructors fail to promote

starting a promoted list. While `'[ 'True ]` is fine, `'['True]` is unfortunately a parse error.

Finally, tuples are promoted with a leading tick. For example, `'( 5, "hello" )` has kind (Nat, Symbol). The aforementioned parsing gotcha applies here as well, so be careful.

Chapter 3

Associated Type Families

3.1 Type Proxies

Let's return to our earlier discussion about `printf`.

One of Haskell's most profound lessons is a deep appreciation for types, and with it, the understanding that `String`s are suitable only for *unstructured* text. Our format "strings" most certainly *are* structured, and thus should not be `String`s.

If `printf`'s format string isn't really a string, what is it?

Well, if we look only at the specifiers in the format string, they're a kind of type signature, describing not only the number of parameters, but also their types. For example, the format string `"%s%d%d"` could be interpreted in Haskell as a function that takes a string, two integers, and returns a string—the concatenation of pushing all of those parameters together. In other words, `"%s%d%d"` corresponds to the type `String -> Int -> Int -> String`.

But, a format string is not only specifiers; it can also contain arbitrary text that should be strung together between the arguments. In our earlier example, this corresponds to format strings like `"hello %s"`. The type corresponding to this function is still only `String -> String`, but its actual implementation should be `\s -> "hello " <> s`.

We can build the desired structure for the semantics behind these format strings by realizing that what we're trying to describe is nothing more than a sequence of Type and Symbol (promoted `String`s), in any order. The Types correspond to parameters to our formatting function, and the Symbols are text to be interspersed between.

With this in mind, we can write the following data definition, whose only purpose is to let us keep track of our kinds:

```
data (a :: k1) :<< (b :: k2)
infixr 5 :<<
```

The `(:<<)` symbol was chosen due to the similarly it has with C++'s `<<` output stream operator, but has no other special meaning to Haskell or to us.

Notice here that `(:<<)` doesn't have any data constructors, so we are unable to construct one of them at the term-level. But because we have a type constructor, we can build types out of them.

In GHCi, we can use the `:kind!` command to inspect our handiwork, which will tell us the *kind of a given type*:

```
> :kind! (:<<)
(:<<) :: k1 -> k2 -> Type
= (:<<)
```

`(:<<)`, when promoted, has kind $\textsc{k1} \to \textsc{k2} \to \textsc{Type}$, corresponding roughly to a promoted 2-tuple. The kind of a saturated `(:<<)` is TYPE, but this is unimportant for our purposes; all we want is a way of defining sequences of things of arbitrary kinds.

To convince ourselves that this works, we can again ask GHCi:

```
> :kind! "hello " :<< String :<< "!"
"hello " :<< String :<< "!" :: *
= "hello " :<< (String :<< "!")
```

Notice that due to our `infixr 5 :<<` declaration, repeated applications of `(:<<)` associate as we'd expect.

Armed with a means of storing our format "string", our next step is to use it to construct the proper type signature of our formatting function. Which is to say, given eg. a type `Int :<< ":" :<< Bool :<< "!"`, we'd like to produce the type `Int -> Bool -> String`. This sounds likes a type-level function, and so we should immediately begin to think about type families.

However, instead of using *closed* type families which are useful when promoting functions from the term level to the type level, we instead will use an *associated type family*. Associated type families, as the name suggests, are associated with a type class, and provide a convenient way to bundle term-level code with computed types. We'll talk about the bundled term-level code in a moment, but first, we can define our associated type family:

```
class HasPrintf a where
  type Printf a :: Type
```

Here we're saying we have a typeclass `HasPrintf a`, of which every instance must provide an associated type `Printf a`, whose kind must be TYPE. `Printf a` will correspond to the desired type of our formatting function, and we will construct it in a moment.

Defining a type family as an associated type family allows us to take advantage of Haskell's capabilities for overlapping typeclasses, which will simplify our logic a great deal.

For simplicity, we will say our format types will always be of the form `a :<< ... :<< "symbol"`, which is to say, some they will always end with a SYMBOL.

Such a simplification gives us a convenient base case for the structural recursion we want to build.

If you're unfamiliar with the concept, structural recursion refers to the technique of technique of producing something by tearing a recursive structure apart into smaller and smaller pieces, until you find a case simple enough you know how to handle. It's really just a fancy name for "divide and conquer."

In our `printf` example, we will use three cases:

1. `instance HasPrintf (text :: Symbol)`
2. `instance HasPrintf a => HasPrintf ((text :: Symbol) :<< a)`
3. `instance HasPrintf a => HasPrintf ((param :: Type) :<< a)`

With these three cases, we can tear down any right-associative sequence of `(:<<)`s via case 2 or 3 until we run out of `(:<<)` constructors, where we are finally left with a Symbol we can handle with case 1.

Case 1 corresponds to having no more parameters, which in turn means it should expand to the desired return type of our formatting function.

```
instance HasPrintf (sym :: Symbol) where
  type Printf sym = String
```

Case 2 corresponds to having additional text we want to inject into our final formatted string, but it doesn't correspond any parameters on its own, so its `Printf` type should be the same as the one it received via structural recursion:

```
instance HasPrintf a
    => HasPrintf ((text :: Symbol) :<< a) where
  type Printf (text :<< a) = Printf a
```

We know that this is an acceptable thing to do, because `Printf a` comes associated with `HasPrintf a`, which is a constraint on our instance of `HasPrintf`.

Case 3 is the most interesting; here we want to add our `param` type as a parameter to the generated function. We can do that by defining `Printf` as an arrow type:

```
instance HasPrintf a
    => HasPrintf ((param :: Type) :<< a) where
  type Printf (param :<< a) = param -> Printf a
```

Here we're saying our formatting type requires a `param`, and then gives back our recursively-defined `Printf a` type. Strictly speaking, the Type kind signature here isn't necessary—GHC will infer it based on `param -> Printf a`—but it adds to the readability, so we keep it.

We can walk through our earlier example of `Int :<< ":" :<< Bool :<< "!"` to convince ourselves that `Printf` expands correctly. First, we see that `Int :<< ":" :<< Bool :<< "!"` is of the form `(param :: Type) :<< a` given the equations `(param ~ Int)` and `(a ~ ":" :<< Bool :<< "!")`.

From here, we expand the definition of `Printf (param :<< a)` into `param -> Printf a`, or, substituting for our earlier type equalities: `Int -> Printf (":" :<< Bool :<< "!")`.

We continue matching `Printf (":" :<< Bool :<< "!")` and notice now that it matches case 2, giving us `Int -> Printf (Bool :<< "!")`. Expansion here again follows case 3, and expands to `Int -> Bool -> Printf "!"`.

Finally, we have run out of `(:<<)` constructors, and so `Printf "!"` matches case 1, where `Printf text = String`. Here our recursion ends, and we find ourselves with the generated type `Int -> Bool -> String`, exactly the type we were looking for.

Analysis of this form is painstaking and time-intensive. Instead, in the future, we can just ask GHCi if we got it right, again with the `:kind!` command:

```
> :kind! Printf (Int :<< ":" :<< Bool :<< "!")
Printf (Int :<< ":" :<< Bool :<< "!") :: Type
= Int -> Bool -> String
```

Much easier.

3.2 Generating Associated Terms

Building the type `Printf a` is wonderful and all, but producing a type without a corresponding term isn't going to do us much good. Our next step is to update the definition of `HasPrintf` to also provide a `format` function.

```
class HasPrintf a where
  type Printf a :: *
  format :: String -> Proxy a -> Printf a
```

The type of `format` is a little odd, and could use an explanation. Looking at the second parameter first, we find a term of type `Proxy a`. This `Proxy` exists only to allow Haskell to find the correct instance of `HasPrintf` from the call-site of `format`. You might think Haskell would be able to find an instance based on the `a` in `Printf a`, but this isn't so for reasons we will discuss soon.

The first parameter, the `String` is an implementation detail, and will act as an accumulator where we can keep track of all of the formatting done by earlier steps in the recursion.

We can update our instance definitions of the three cases so they correctly implement `format`. In the first case, we have no work to do, so the only thing necessary is to return the accumulator and append the final text to it.

```
instance KnownSymbol text => HasPrintf (text :: Symbol) where
  type Printf text = String
  format s _ = s <> symbolVal (Proxy @text)
```

`symbolVal` is a function that converts a Symbol into a `String`. For example, it will turn the type `"hello" :: Symbol` into the *term* `"hello" :: String`. `symbolVal`'s is `KnownSymbol sym => Proxy sym -> String`, and all this `KnownSymbol` stuff is simply a proof that GHC knows what Symbol we're talking about; it will automatically generate the `KnownSymbol` instance for us, so it's nothing we need to worry about.

Case 2 is very similar; here we want to update our accumulator with the `symbolVal` of `text`, but also structurally recursively call `format`. This requires conjuring up a `Proxy a`, which we can do via `-XTypeApplications`:

```
instance (HasPrintf a, KnownSymbol text)
    => HasPrintf ((text :: Symbol) :<< a) where
  type Printf (text :<< a) = Printf a
  format s _ = format (s <> symbolVal (Proxy @text))
                      (Proxy @a)
```

All that's left is case 3, which if you've been paying attention to the other cases, should look familiar:

```
instance (HasPrintf a, Show p)
    => HasPrintf ((param :: Type) :<< a) where
  type Printf (param :<< a) = param -> Printf a
  format s _ p = format (s <> show p) (Proxy @a)
```

Notice the `p` parameter to our `format` function here—this corresponds to the `param` parameter in case 3's `Printf` instance. For any specifier, we use its `Show` instance to convert the parameter into a string, and append it to our accumulator.

With all three of our cases covered, we appear to be finished. We can define a helper function to hide the accumulator from the user, since it's purely an implementation detail:

```
printf :: HasPrintf a => Proxy a -> Printf a
printf = format ""
```

Firing up GHCi allows us to try it:

```
> printf (Proxy @"test")
"test"
> printf (Proxy @(Int :<< " + " :<< Int :<< " = 3")) 1 2
"1 + 2 = 3"
> printf (Proxy @(String :<< " world!")) "hello"
"\"hello\" world!"
```

It works pretty well for our first attempt, all things considered. One noticeable flaw is that `String`s gain an extra set of quotes due to being `show`n. We can fix this infelicity by providing a special instance of `HasPrintf` just for `String`s:

```
instance {-# OVERLAPPING #-} HasPrintf a
    => HasPrintf (String :<< a) where
  type Printf (String :<< a) = String -> Printf a
  format s _ param = format (s <> param) (Proxy @a)
```

Writing this instance will require the `-XFlexibleInstances` extension, since the instance head is no longer just a single type constructor and type variables. We mark the instance with the `{-# OVERLAPPING #-}` pragma because we'd like to select this instance instead of case 3 when the parameter is a `String`.

```
> printf (Proxy @(String :<< " world!")) "hello"
"hello world!"
```

Marvelous.

What we've accomplished here is a type-safe version of `printf`, but recognizing that C++'s "format string" is better thought of as a "structured type signature." Using type-level programming, we were able to convert such a thing into a function with the correct type, that implements nontrivial logic.

3.3 Ambiguous Types and Non-Injectivity

We return now to the question of what's going with that pesky `Proxy` thing in the earlier examples. The purpose of `Proxy a` in the type signature of `format` is to gently guide GHC towards finding the instance of `HasPrintf` that we want.

But `format :: String -> Proxy a -> Printf a` already has an `a` visible in its type signature, in the form of `Printf a`. Why isn't this enough to drive instance resolution?

To see why this must be the case, let's look at a simpler example of a closed type family:

```
type family AlwaysUnit a where
  AlwaysUnit a = ()
```

The `AlwaysUnit` family maps every type to `()`. The usefulness of such a thing is limited, but it serves to illustrate our problem.

Broken Code

```
class TypeName a where
  typeName :: AlwaysUnit a -> String

 instance TypeName String where
  typeName _ = "String"

 instance TypeName Bool where
  typeName _ = "Bool"

 name :: String
 name = typeName ()
```

The problem, of course is that both `AlwaysUnit String` $\sim$ `()` and `AlwaysUnit Bool` $\sim$ `()`, which means that given `AlwaysUnit a` $\sim$ `()`, we can't learn anything about `a`. More specifically, the problem is that `AlwaysUnit` doesn't have an inverse; there's no `Inverse` type family such that `Inverse (AlwaysUnit a)` $\sim$ `a`. In mathematics, this property is known as *non-injectivity.*

Consider an analogous example from cryptography; just because you know the hash of someone's password is `1234567890abcdef` doesn't mean you know what the password is; any good hashing function, like `AlwaysUnit`, is *one way*. Just because we can go one way doesn't mean we can go backwards.

In the case of `name` above, when GHC sees `typeName ()`, it doesn't have enough information to go backwards and figure out what `a` is supposed to be, because all it know is `AlwaysUnit a` $\sim$ `()`. But again, this is true for all types `a`, so it is not a particularly helpful fact to know.

Going back to `Printf a`, we can see that both `Printf "hello"` $\sim$ `String` and `Printf "goodbye"` $\sim$ `String`, so `Printf` can't possibly be injective.

The solution to non-injectivity is to give GHC some other way of determining the otherwise ambiguous type. This can be done like in our examples by adding a `Proxy a` parameter whose only purpose is to drive inference, or it can be accomplished by enabling `-XAllowAmbiguousTypes` at the definition site, and using `-XTypeApplications` at the call-site to fill in the ambiguous parameter manually. We will discuss these two extensions more in a later chapter.

Chapter 4

Generics

4.1 Overview

Haskell offers two kinds of polymorphism: parametric polymorphism, which has one definition for every possible type (think `head :: [a] -> a`); and ad-hoc polymorphism, where every type can do its own thing (`mempty :: a`). But for our purposes, there's also a third category to keep in mind: *boilerplate polymorphism.*

Boilerplate polymorphism doesn't have any formal definition, but it's the kind of thing you recognize when you see it. It's ad-hoc polymorphism that doesn't require any sort of thought in order to write. `Eq`, `Show` and `Functor` instances are good examples of boilerplate polymorphism—there's nothing interesting about writing these instances. The tedium of writing boilerplate polymorphism is somewhat assuaged by the compiler's willingness to write some of them for us.

Consider the `Eq` typeclass; while every type needs its own implementation of `(==)`, these implementations are always of the form:

```
instance Eq Foo where
  F0       == F0       = True
  F1 a1    == F1 a2    = a1 == a2
  F2 b1 c1 == F1 b2 c2 = b1 == b2 && c1 == c2
  _        == _        = False
```

Instances of `Eq` are always the same; the same data constructors are equal if and only if all of their components are equal.

Boilerplate polymorphism is mindless work to write, but needs to be done. In the case of some of the standard Haskell typeclasses, GHC is capable of writing these instances for you via `deriving`. Unfortunately, for custom typeclasses we're on our own, without any direct support from the compiler.

As terrible as this situation appears, all hope is not lost. Using `GHC.Generics`, we're capable of writing our own machinery for helping GHC derive our typeclasses, all in regular Haskell code.

4.2 Generic Representations

Before diving into the generic machinery, it will be instructive to spend some time thinking about the *functoriality* of datatypes. As it happens, all datatypes have a "canonical" representation as a *sum-of-products*. In Haskell, the canonical sum type is `Either`, and the canonical product is the 2-tuple `(,)`. As it happens, every `data` and `newtype` definition is equivalent to some composition of sums and products.

Take for example, `Maybe a`, which you'll recall has this definition:

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

`Maybe a`, in its canonical sum-of-products form is `Either a ()`, which we can prove is the same thing via an isomorphism:

```
toCanonical :: Maybe a -> Either a ()
toCanonical (Just a) = Left a
toCanonical Nothing  = Right ()

unCanonical :: Either a () -> Maybe a
unCanonical Left a     = Just a
unCanonical (Right ()) = Nothing
```

Since we can convert to and from the canonical representation without losing any information in either direction, we know that these types must be equivalent. But the question remains: who cares?

Well, the point is that if we have a small number of primitive building blocks, we can write our generic code to operate over those primitives. If we then also had the ability to convert to and from our canonical representations at will, we'd be able to take regular Haskell data, convert it to its canonical representation, manipulate it, and then convert it back.

How can such a thing be possible? The secret is in the `-XDeriveGeneric` extension, which will automatically derive an instance of `Generic` for you:

```
class Generic a where
  type Rep a :: Type
  from :: a -> Rep a x
  to   :: Rep a x -> a
```

`Rep a` is the canonical sum-of-products representation of our type `a`, while `from` and `to` correspond to our above isomorphism between the two types.

Let's look at `Rep Bool` for inspiration about what this thing might look like.

```
> :kind! Rep Bool
Rep Bool :: * -> *
```

```
= D1 ('MetaData "Bool" "GHC.Types" "ghc-prim" 'False)
         ( C1 ('MetaCons "False" 'PrefixI 'False) U1
       :+: C1 ('MetaCons "True"  'PrefixI 'False) U1
         )
```

Quite a mouthful, but at it's heart the interesting parts of this are the `(:+:)` and `U1` types. These correspond to the *canonical sum* and *canonical unit*, respectively. When viewed beside the definition of `Bool`, some similarities appear:

```
data Bool
  = False
  | True
```

While `(:+:)` and `U1` give us the *shape* of `Bool`, the `D1` and `C1` give us metadata about `Bool`'s definition itself. `D1` describes the definition of `Bool`, including its name, module, package and whether or not the type is a `newtype`. `C1` is used to describe data constructors, with its name, fixity definition, and whether or not it contains record selectors. All of this metadata is provided via a data kind, so it exists in the type-level and can be manipulated accordingly.

4.3 Deriving Any Class

Armed with the knowledge of `Rep`, we can write an illustrative example of generically deriving `Eq`. The approach is threefold:

1. Define a typeclass to act as a "carrier."

2. Provide inductive instances of the class for the generic constructors.

3. Finally, write a helper function to map between the `Rep`.

We begin by defining our typeclass. A good convention is add a `G` prefix to a generic typeclass. If you want to derive `Eq` generically, call your carrier typeclass `GEq`.

```
class GEq a where
  geq :: a x -> a x -> Bool
```

Our `GEq` class has a single method, `geq`, whose signature closely matches `(==) ::  a -> a -> Bool`.

Notice that the type parameter `a` to `GEq` has kind Type $\to$ Type. This is a quirk of `GHC.Generics`, and allows the same `Rep` machinery when dealing with higher-kinded classes. When writing classes for types of kind Type, we will always saturate `a` with a dummy type `x`.

A good approach when writing generic instances is to work "inside-out." Start with the innermost constructors (`K1`, `U1` and `V1`), as these are the base cases of the structural induction.

In our case, U1 is the simplest, so we will start there. Recall that U1 represents a data constructor with no parameters, in which case it's just () with a different name. Since () is always equal to itself, so too should U1 be.

```
instance GEq U1 where
  geq U1 U1 = True
```

Similarly for V1 which corresponds to types that can't be constructed. It might seem silly to provide an Eq instance for such types, but it costs us nothing. Consider instances over V1 as being vacuous; if you *could* give me a value of V1, I claim that I could give you back a function comparing it for equality. Since you *can't* actually construct a V1, then my claim can never be tested, and so we might as well consider it true.

Strictly speaking, V1 instances often aren't necessary, but we might as well provide one if we can.

```
instance GEq V1 where
  geq _ _ = True
```

The one other case we need to consider is what should happen for real parameters to data constructors? Such things are denoted via K1, and in this case, we want to fall back on an Eq (*not* GEq!) instance to compare the two. The analogous non-generic behavior for this is how the Eq instance for Maybe a is Eq a => Eq (Maybe a); most datatypes simply want to lift equality over their constituent fields.

```
instance Eq a => GEq (K1 _i a) where
  geq (K1 a) (K1 b) = a == b
```

But why should we use an Eq constraint rather than GEq? Well we're using GEq to help derive Eq, which implies Eq is the actual type we care about. If we were to use a GEq constraint, we'd remove the ability for anyone to write a non-generic instance of Eq!

With our base cases complete, we're ready to lift them over products and sums. We can lift equality over products component-wise, and over sums on whether or not they are the same data constructor.

```
instance (GEq a, GEq b) => GEq (a :*: b) where
  geq (a1 :*: b1) (a2 :*: b2) = geq a1 a2 && geq b1 b2

instance (GEq a, GEq b) => GEq (a :+: b) where
  geq (L1 a1) (L1 a2) = geq a1 a2
  geq (R1 b1) (R1 b2) = geq b1 b2
  geq _ _             = False
```

Finally, we want to lift all of our GEq instances through the Rep's metadata constructors, since the names of things aren't relevant for defining Eq. Fortunately, all of the various types of metadata provided by GHC.Generics are all type synonyms of M1:

```
instance GEq a => GEq (M1 _x _y a) where
  geq (M1 a1) (M1 a2) = geq a1 a2
```

Using `-XDefaultSignatures`, we're also capable of getting `-XDeriveAnyClass` to work for us!

```
class MyEq a where
  eq :: a -> a -> Bool
  default eq :: (Generic a, GEq (Rep a)) => a -> a -> Bool
  eq a b = geq (from a) (from b)

data Foo
  = F0
  | F1 String
  deriving (Generic, MyEq)  . . . . . . . . . . . . . . . . . . . . .  ❶
```

Notice how at ❶, we have *not* derived an instance of `Eq`. We can fire up the REPL to see how we did:

```
> eq F0 F0
True

> eq (F1 "foo") (F1 "foo")
True

> eq F0 (F1 "hello")
False

> eq (F1 "foo") (F1 "bar")
False
```

Just as we'd expect! Awesome!

4.4 Generic Metadata

For reasons beyond the author's comprehension, rather than imbue JavaScript with a reasonable typesystem, its proponents have instead blessed us with JSON Schema. For those of you lucky enough to be unfamiliar with it, JSON Schema is, in its own words "a vocabulary that allows you to annotate and validate JSON documents." It's sort of like a typesystem, but ad-hoc and not very powerful.

For example, the following Haskell type:

```
data Person = Person
  { name  :: String
  , age   :: Int
  , phone :: Maybe String
  }
```

would be described in JSON Schema as:

```
{ "title": "Person"
, "type": "object"
, "properties":
    { "name":  { "type": "string"  }
    , "age":   { "type": "integer" }
    , "phone": { "type": "string"  }
    }
, "required": ["name" , "age"]
}
```

I'm sorry to have had to introduce you to this abomination, but it is what it is. We can't do anything about JSON Schema, but at least it provides a motivating example of generating code generically, lest anyone need to write such a monstrosity themselves.

We begin as always with the definition of our typeclass. Obviously such a thing needs to produce a `Value` (aeson's representation of a JSON value). Less clear is that in order to produce the `required` property, we'll also need to propagate information about required-ness upwards. As such, we decide our `GSchema` typeclass will look like this:

```
class GSchema (a :: Type -> Type) where
  gschema :: (Value, [Text])
```

Notice that `gschema` doesn't have any way of binding the `a` type parameter. While we *could* use a `Proxy` to drive the instance lookups the way we did for `HasPrintf`, a cleaner interface is to enable `-XAllowAmbiguousTypes` and later use `-XTypeApplications` to fill in the desired type variable. We will see this usage in a moment.

For our purposes, we will assume we only want to generate JSON Schema for Haskell records – anonymous products are out, and it's not clear how to represent sums anyway.

To begin with, we'll write a helper function to generate the property objects for us—things of the form `{"foo": {"type": "integer"}}`. We'll write this function to take its name (the `"foo"`) from a `KnownSymbol`, because that's the form we'll receive it in from `GHC.Generics`.

use a type fam instead of Typeable

```
makePropertyObj
    :: forall name. KnownSymbol name
    => String  --^ The JSON type.
    -> Value
makePropertyObj ty = object
  [ pack (symbolVal $ Proxy @name) .= object
      [ "type" .= String (pack ty)
      ]
  ]
```

In order to get access to the record name, it's insufficient to simply define an instance of `GSchema` for `K1`, since by the time we get to `K1` we've lost access to the metadata. Instead, we can do (type-level) pattern matching on `M1 S meta (K1 _ a)`; the `S` type is used as a parameter to `M1` to describe *record selector metadata*.

open type family to generate js names

If the type we're eventually wrapping is a `Maybe Int`, we'd like to say that its type in JSON Schema is an `integer`, and that it isn't required. Since `Maybe a` is a more specific type than `a`, it seems like a good place to start. We'll mark it as an `OVERLAPPING` instance:

```
instance {-# OVERLAPPING #-} (Typeable a, KnownSymbol nm)
    => GSchema (M1 S ('MetaSel ('Just nm) _1 b_c _d)
                     (K1 _e (Maybe a))) where
  gschema =
    ( makePropertyObj @nm (show . typeRep $ Proxy @a)
    , mempty
    )
```

This instance will give us back our property object, some JSON of the form `{"foo": {"type": "integer"}}`, and an empty list corresponding to having not yet discovered any required fields. We leave the responsibility of combining these JSON objects to other instances of `GSchema`.

The case for `K1 a` is very similar, except that we also emit the name as a required field:

```
instance (Typeable a, KnownSymbol nm)
    => GSchema (M1 S ('MetaSel ('Just nm) _1 b_c _d)
                     (K1 _e a)) where
  gschema =
    ( makePropertyObj @nm (show . typeRep $ Proxy @a)
    , pure . symbolVal $ Proxy @nm
    )
```

We can define products `(:*:)` for our types as simply merging our property objects together via their underlying semigroup:

```
addObjects
    :: Semigroup a
    => (Value, a)
    -> (Value, a)
    -> (Value, a)
addObjects (Object a, x) (Object b, y)
  = (Object $ a <> b, x <> y)

instance (GSchema f, GSchema g) => GSchema (f :*: g) where
  gschema = addObjects (gschema @f) (gschema @g)
```

It's unclear how to represent sum-types in JSON schema, so we won't provide an instance of `(:+:)`. Furthermore, it doesn't seem valuable to marshall unit and void types to JavaScript, so we won't provide instances for `U1` or `V1` either.

All that remains is the remaining `M1` instances; `M1 D` is used for type information, which doesn't matter to us, and `M1 C` which gives us data constructor information. It is from `M1 C` that we will get our object's `"title"`, and set up the top-level information about our object:

```
instance GSchema a => GSchema (M1 D _1 a) where
  gschema = gschema @a

instance (GSchema a, KnownSymbol nm)
    => GSchema (M1 C ('MetaCons nm _1 _2) a) where
  gschema =
    let (sch, req) = gschema @a
     in ( object
              [ "title" .= (String . pack . symbolVal $ Proxy @nm)
              , "type" .= String "object"
              , "properties" .= sch
              ]
          , req
          )
```

Here we use `_1` and `_2` as "wildcard types." These are not special in any way, but Haskell doesn't allow us to write `_` as a type variable. Most of the time, most the data in the `M1` constructor won't be relevant to you, and so you'll probably end up using a lot ot these wildcard types.

All that's left is to put a nice veneer in front of `gschema`, and to build our `"required"` field.

```
schema :: forall a. (GSchema (Rep a), Generic a) => Value
schema =
  let (v, reqs) = gschema @(Rep a)
   in fst $ addObjects (v, ())
        ( object
            [ "required" .=
                Array (fromList $ String . pack <$> reqs)
            ]
        , ()
        )
```

And we're done. We've successfully used the metadata in `GHC.Generics` to automatically marshall a description of our Haskell datatypes into JSON Schema. We didn't need to resort to using code generation—which would have complicated our compilation pipeline—and we've written nothing but everday Haskell in order to accomplish it.

4.5 Performance

Before diving much further into `GHC.Generics` and all of the wonderful things we can do with them, it's worthwhile to take a minute to slow down. Sure, describing all of this behavior in Haskell is great, but do we pay an overhead at runtime to get it? If so, it might not be a very good investment; writing things by hand is annoying and tedious, but at least we have some understanding of what's going on under the hood. With `GHC.Generics`, it's certainly less clear.

There is good and bad news here. The good news is that usually adding `INLINE` pragmas to each of your class' methods is enough to optimize away all usage of `GHC.Generics`. The bad news is that this is *usually* enough to optimize them away. Since there is no separate compilation step when working with `GHC.Generics`, it's quite a lot of work to actually determine whether or not your generic code is being optimized away.

Enter the `inspection-testing`[1] library. `inspection-testing` provides a plugin to GHC which allows us to make assertions about our generated code. We can use it to ensure GHC optimizes away all of our usages of `GHC.Generics`, and instead spits out the same code that we would have written by hand.

We can use `inspection-testing` like so:

1. Enable the `{-# OPTIONS_GHC -O -fplugin Test.Inspection.Plugin #-}` pragma.
2. Enable `-XTemplateHaskell`.
3. Import `Test.Inspection`.
4. Write some code that exercises the generic code path. Call it `foo`, for example.
5. Add `inspect $ hasNoGenerics 'foo` to your top level module.

real example here

And that's it. Now when you compile your module, GHC will yell at you and refuse to compile if your generic code has any runtime overhead. Unfortunately for us, `inspection-testing` isn't magic and can't guarantee our implementation is as good as a hand-written example, but at least it can prove the generic representations don't exist at runtime.

In order to prove two implementations (eg. one written generically and one written by hand) *are* equal, you can use `inspection-testing`'s `(===)` combinator, which causes a compile-time error if the actual generate code isn't identical. This is often impractical to do for complicate usages of `GHC.Generics`, but it's comforting to know that it's possible in principle.

However, there is a particularly egregious case that GHC is unable to optimize. It's described colloquially as "functions that are too polymorphic." But what does it mean to be *too polymorphic*?

This class of problems sets in when GHC requires knowing about the functor/applicative/monad laws in order to perform the inlining, but the type itself is polymorphic. That is to say, a generic function that produces a `forall m. m a` will perform poorly, but `Maybe a` is fine. A good rule of thumb is that if you have a polymorphic higher-kinded type, your performance is going to go into the toolies.

4.6 Yoneda and Codensity

day currying?

The good news is that reclaiming our performance from the clutches of too-polymorphic generics isn't very hard. The secret is to rewrite our types from the form `forall f. Functor f => f a` into `forall f. Yoneda f a`, and from the form `forall f. Monad m => m a` (and `Applicatives`) into `forall m. Codensity m a`. Both come from the `kan-extensions`[2] We'll talk more about this transformation in a moment.

In essence, the trick here is to write our "too polymorphic" code in a form that amenable to GHC's inlining abilities, and then transform it back into the desired form at the very end. `Yoneda` and `Codensity` are tools that can help with this transformation.

Consider the definition of `Yoneda`:

```
newtype Yoneda f a = Yoneda
  { runYoneda :: forall b. (a -> b) -> f b
  }
```

When we ask GHCi about the type of `runYoneda`, we see something interesting:

```
> :t runYoneda
runYoneda :: Yoneda f a -> (a -> b) -> f b
```

`runYoneda` looks a lot like `flip fmap :: f a -> (a -> b) -> f b`, doesn't it? This is not an accident. The `Functor` instance for `Yoneda` is particularly enlightening:

```
instance Functor (Yoneda f) where
  fmap f (Yoneda y) = Yoneda $ f . y
```

Note the lack of a `Functor f` constraint on this instance! `Yoneda f` is a `Functor` *even when* `f` *isn't.* In essence, `Yoneda f` gives us a free instance of `Functor` for any type of kind Type → Type. We call `Yoneda` the *free* `Functor`. There's lots of interesting category theory behind all of this, but it's not important to us.

But how does `Yoneda` work? Keeping in mind the functor law that `fmap f . fmap g = fmap (f . g)`, the implementation of `Yoneda`'s `Functor` instance looks very similar. All `Yoneda` is doing is accumulating all of the functions we'd like to `fmap`, so that it can perform them all at once.

As interesting as all of this is, the question remains: how does `Yoneda` help GHC optimize our programs? Recall that GHC's failure to inline "too polymorphic" functions is due to it being unable to perform the functor/etc laws while inlining polymorphic code. But since `Yoneda f` is a functor even when `f` isn't, `Yoneda`'s `Functor` instance can't possibly depend on `f`. That means `Yoneda f` is never "too polymorphic," and as a result, acts as a fantastic carrier for our optimization tricks.

Finally, the functions `liftYoneda :: Functor f => f a -> Yoneda f a` and `lowerYoneda :: Yoneda f a -> f a` witness that `Yoneda f a` is isomorphic to `f a`. Whenever your generic code needs to do something in `f`, it should use `liftYoneda`, and the final interface to your generic code should make a call to `lowerYoneda` to hide it as an implementation detail.

`Codensity` is to `Monad` as `Yoneda` is to `Functor`. This means `Codensity` is the *free* `Monad`, and can likewise to be used to optimize "too polymophic" monadic code.

Both types have other interesting uses—notably, `Codensity` is invaluable in flattening JavaScript-esque "callback pyramids of doom"—though we will not dwell on them any further, as such things do not fall within the scope of this book. The motivated reader is encouraged to familiarize themself with the `kan-extensions` package.

Bibliography

[1] Joachim Breitner. https://github.com/nomeata/inspection-testing

[2] Ed Kmett. https://github.com/ekmett/kan-extensions