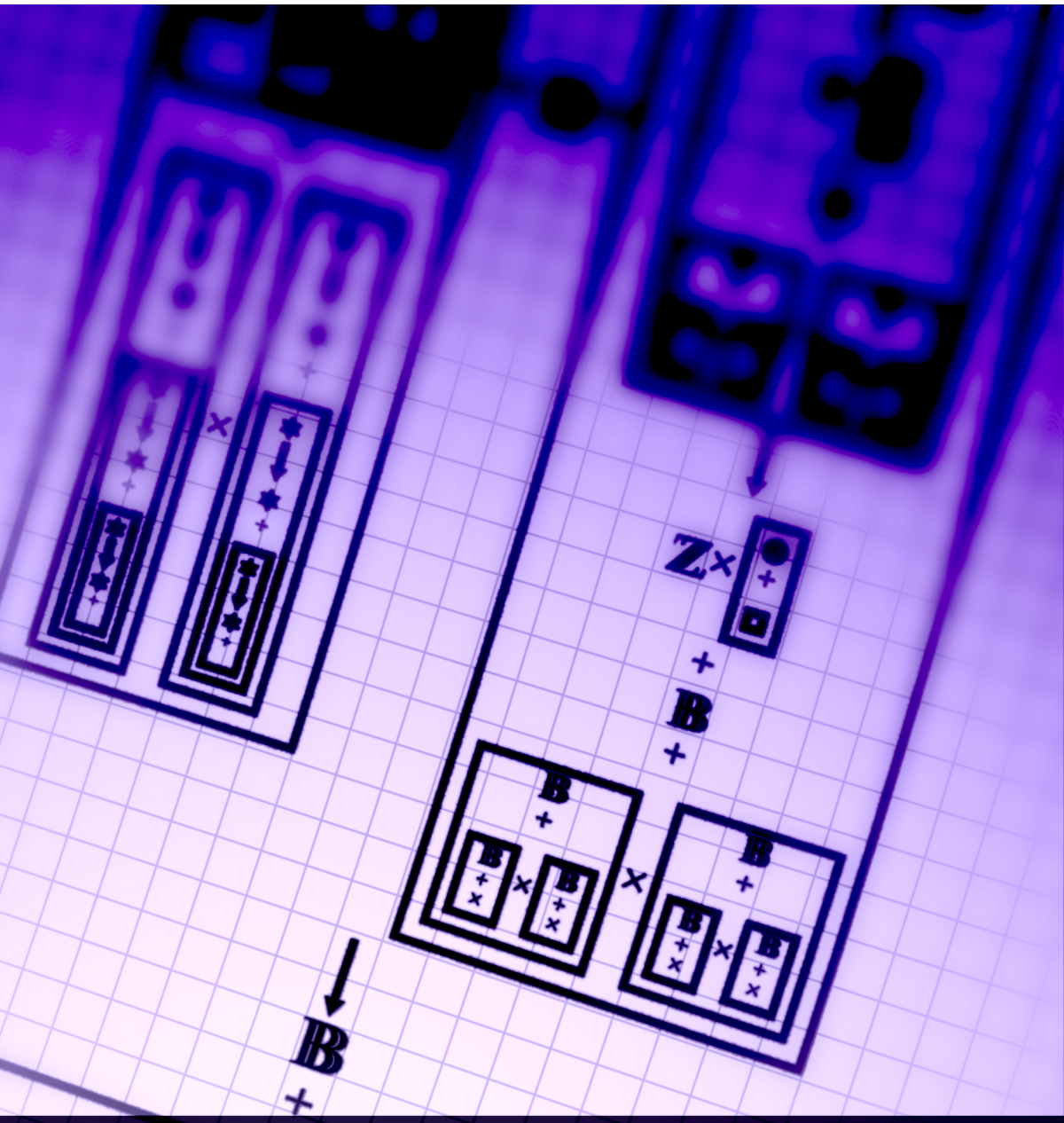


Thinking with Types

Type-Level Programming in Haskell

Sandy Maguire

Thinking with Types

Sandy Maguire

First Edition

When people say
“but most business logic bugs
aren’t type errors,”
I just want to show them
how to make bugs
into type errors.

MATT PARSONS

Contents

Introduction

Type-level programming is an uncommon calling. While most programmers are concerned with getting more of their code to compile, we type-level programmers are trying our best to *prevent* code from compiling.

Strictly speaking, the job of types is twinfold—they prevent (wrong) things from compiling, and in doing so, they help guide us towards more elegant solutions. For example, if there are ten solutions to a problem, and nine of them be poorly-typed, then we need not look very hard for the right answer.

But make no mistake—this book is primarily about reducing the circumstances under which a program compiles. If you're a beginner Haskell programmer who feels like GHC argues with you too often, who often finds type errors inscrutable, then this book is probably not for you. Not yet.

So whom is this book for? The target audience I've been trying to write for are intermediate-to-proficient with the language. They're capable of solving real problems in Haskell, and doing it without too much hassle. They need not have strong opinions on `ExceptT` vs throwing exceptions in `IO`, nor do they need to know how to inspect generated Core to find performance bottlenecks.

But the target reader should have a healthy sense of unease about the programs they write. They should look at their comments saying "don't call this function with

$n = 5$ because it will crash," and wonder if there's some way to teach the compiler about that. The reader should nervously eyeball their calls to `error` that they're convinced can't possibly happen, but are required to make the type-checker happy.

In short, the reader should be looking for opportunities to make *less* code compile. This is not out of a sense of masochism, anarchy, or any such thing. Rather, this desire comes from a place of benevolence—a little frustration with the type-checker now is preferable to a hard-to-find bug making its way into production.

Type-level programming, like anything, is best in moderation. It comes with its own costs in terms of complexity, and as such should be wielded with care. While it's pretty crucial that your financial application handling billions of dollars a day runs smoothly, it's a little less critical if your hobbyist video game draws a single frame of gameplay incorrectly. In the first case, it's probably worthwhile to use whatever tools you have in order to prevent things from going wrong. In the second, these techniques are likely too heavy-handed.

Style is a notoriously difficult thing to teach—in a very real sense, style seems to be what's left after we've extracted from a subject all of the things we know how to teach. Unfortunately, when to use type-level programming is largely a matter of style. It's easy to take the ball and run with it, but discretion is divine.

When in doubt, err on the side of *not* doing it at the type-level. Save these techniques for the cases where it'd be catastrophic to get things wrong, for the cases where a little type-level stuff goes a long way, and for the cases where it will drastically improve the API. If your use-case isn't obviously one of these, it's a good bet that there is a cleaner and easier means of doing it with values.

But let's talk more about types themselves.

As a group, I think it's fair to say that Haskellers are contrarians. Most of us, I'd suspect, have spent at least

one evening trying to extol the virtues of a strong type system to a dynamically typed colleague. They'll say things along the lines of "I like Ruby because the types don't get in my way." Though our first instinct, as proponents of strongly typed systems, might be to forcibly connect our head to the table, I think this is a criticism worth keeping in mind.

As Haskellers, we certainly have strong opinions about the value of types. They *are* useful, and they *do* carry their weight in gold when coding, debugging and refactoring. While we can dismiss our colleague's complaints with a wave of the hand and the justification that they've never seen a "real" type system before, we are doing them and ourselves a disservice both. Such a flippant response is to ignore the spirit of their unhappiness—types *often do* get in the way. We've just learned to blind ourselves to these shortcomings, rather than to bite the bullet and entertain that maybe types aren't always the 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 ruled out by a type system. Consider, for example, the following program which has a type-error, but never actually evaluates it:

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

Because the type error gets ignored lazily by `fst`, evaluation of such an expression will happily produce `"no problems"` at runtime. Despite the fact that we consider it to be ill-typed, it is in fact, well-behaved. The usefulness of such an example is admittedly low, but the point stands; types often do get in the way of perfectly reasonable programs.

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

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

If you've never before had the pleasure of using `printf`, it works like this: it parses the `format` parameter, and uses its structure to pop additional arguments off of the call-stack. You see, it's the shape of `format` that decides what parameters should fill in the `...` above.

For example, the format string `"hello %s"` takes an additional string and interpolates 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")`, producing "hello world",
- `printf("%d + %d = %s", 1, 2, "three")`, producing "1 + 2 = three",
- `printf("no specifiers")`, producing "no specifiers".

Notice that, as written, it seems impossible to assign a Haskell-esque type signature to `printf`. The additional parameters denoted by its ellipsis are given types by the value of its first parameter—a string. Such a pattern is common in dynamically typed languages, and in the case of `printf`, it's inarguably useful.

The documentation for `printf` is quick to mention that the format string must not be provided by the user—doing so opens up vulnerabilities in which an attacker can corrupt memory and gain access to the system. Indeed, this is hugely widespread problem—and crafting such a string is often the first homework in any university lecture 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. The following, innocuous-looking calls to `printf` are both malicious.

- `printf("%d")`, which will probably corrupt the stack,
- `printf("%s", 1)`, which will read an arbitrary amount of memory.

C's type system is insufficiently expressive to describe `printf`. But because `printf` is such a useful function, this is not a persuasive-enough reason to exclude it from the language. Thus, type-checking is effectively turned off for calls to `printf` so as to have ones cake and eat it too. However, this opens a hole through which type errors can make it all the way to runtime—in the form of undefined behavior and security issues.

My opinion is that preventing security holes is a much more important aspect of the types, over "`null` is the billion dollar mistake" or whichever other arguments are in vogue today. We will return to the problem of `printf` in chapter 9.

With very few exceptions, the prevalent attitude of Haskellers has been to dismiss the usefulness of ill-typed programs. The alternative is an uncomfortable truth: that our favorite language can't do something useful that other languages can.

But all is not lost. 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.

Part I

Fundamentals

Chapter 1

The Algebra Behind Types

1.1 Isomorphisms and Cardinalities

One of functional programming's killer features is pattern matching, as made possible by *algebraic data types*. But this term isn't just a catchy title for things that we can pattern match on. As their name suggests, there is in fact an *algebra* behind algebraic data types.

Being comfortable understanding and manipulating this algebra is a mighty superpower—it allows us to analyze types, find more convenient forms for them, and determine which operations (eg. typeclasses) are possible to implement.

To start, we can associate each finite type with its *cardinality*—the number of inhabitants it has, ignoring bottoms. Consider the following simple type definitions:

```
data Void
```

```
data () = ()
```

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

`Void` has zero inhabitants, and so it is assigned cardinality 0. The unit type `()` has one inhabitant—thus its cardinality is 1. Not to belabor the point, but `Bool` has cardinality 2, corresponding to its constructors `True` and `False`.

We can write these statements about cardinality more formally:

$$\begin{aligned} |\texttt{Void}| &= 0 \\ |\texttt{()}| &= 1 \\ |\texttt{Bool}| &= 2 \end{aligned}$$

Any two finite types that have the same cardinality will always be isomorphic to one another. An *isomorphism* between types `s` and `t` is defined as a pair of functions `to` and `from`:

```
to   :: s -> t
from :: t -> s
```

such that composing either after the other gets you back where you started. In other words, such that:

```
to . from = id
from . to = id
```

We sometimes write an isomorphism between types `s` and `t` as `s` $\cong$ `t`.

If two types have the same cardinality, any one-to-one mapping between their elements is exactly these `to` and `from` functions. But where does such a mapping come from? Anywhere—it doesn't really matter! Just pick an arbitrary ordering on each type—not

necessarily corresponding to an `Ord` instance—and then map the first element under one ordering to the first element under the other. Rinse and repeat.

For example, we can define a new type that also has cardinality 2.

```
data Spin = Up | Down
```

By the argument above, we should expect `Spin` to be isomorphic to `Bool`. Indeed it is:

```
boolToSpin1 :: Bool -> Spin
boolToSpin1 False = Up
boolToSpin1 True  = Down

spinToBool1 :: Spin -> Bool
spinToBool1 Up   = False
spinToBool1 Down = True
```

However, note that there is another isomorphism between `Spin` and `Bool`:

```
boolToSpin2 :: Bool -> Spin
boolToSpin2 False = Down
boolToSpin2 True  = Up

spinToBool2 :: Spin -> Bool
spinToBool2 Up   = True
spinToBool2 Down = False
```

Which of the two isomorphisms should we prefer?

Does it matter?

In general, for any two types with cardinality n, there are $n!$ unique isomorphisms between them. As far as the math goes, any of these is just as good as any other—and for most purposes, knowing that an isomorphism *exists* is enough.

An isomorphism between types `s` and `t` is a proof that *for all intents and purposes,* `s` and `t` *are the same thing.* They might have different instances available, but this is more a statement about Haskell's typeclass machinery than it is about the equivalence of `s` and `t`.

Isomorphisms are a particularly powerful concept in the algebra of types. Throughout this book we shall reason via isomorphism, so it's best to get comfortable with the idea now.

1.2 Sum, Product and Exponential Types

In the language of cardinalities, *sum types* correspond to addition. The canonical example of these is `Either a b`, which is *either* an `a` or a `b`. As a result, the cardinality (remember, the number of inhabitants) of `Either a b` is the cardinality of `a` plus the cardinality of `b`.

$$|\texttt{Either a b}| = |\texttt{a}| + |\texttt{b}|$$

As you might expect, this is why such things are called *sum* types. The intuition behind adding generalizes to any datatype with multiple constructors—the cardinality of a type is always the sum of the cardinalities of its constructors.

```
data Deal a b
  = This a
  | That b
  | TheOther Bool
```

We can analyze `Deal`'s cardinality;

$$\begin{aligned}|\texttt{Deal a b}| &= |\texttt{a}| + |\texttt{b}| + |\texttt{Bool}| \\ &= |\texttt{a}| + |\texttt{b}| + 2\end{aligned}$$

We can also look at the cardinality of `Maybe a`. Because nullary data constructors are uninteresting to construct—there is only one `Nothing`—the cardinality of `Maybe a` can be expressed as follows;

$$|\texttt{Maybe a}| = 1 + |\texttt{a}|$$

Dual to sum types are the so-called *product types*. Again, we will look at the canonical example first—the pair type `(a, b)`. Analogously, the cardinality of a product type is the *product* of their cardinalities.

$$|\texttt{(a, b)}| = |\texttt{a}| \times |\texttt{b}|$$

To give an illustration, consider mixed fractions of the form $5\frac{1}{2}$. We can represent these in Haskell via a product type;

```
data MixedFraction a = Fraction
  { mixedBit    :: Word8
  , numerator   :: a
  , denominator :: a
  }
```

And perform its cardinality analysis as follows:

$$|\texttt{MixedFraction a}| = |\texttt{Word8}| \times |\texttt{a}| \times |\texttt{a}| = 256 \times |\texttt{a}| \times |\texttt{a}|$$

An interesting consequence of all of this cardinality stuff is that we find ourselves able to express *mathematical truths in terms of types*. For example, we can

prove that $a \times 1 = a$ by showing an isomorphism between `(a, ())` and `a`.

```
prodUnitTo :: a -> (a, ())
prodUnitTo a = (a, ())


prodUnitFrom :: (a, ()) -> a
prodUnitFrom (a, ()) = a
```

Here, we can think of the unit type as being a monoidal identity for product types—in the sense that "sticking it in doesn't change anything." Because $a \times 1 = a$, we can pair with as many unit types as we want.

Likewise, `Void` acts as a monoidal unit for sum types. To convince ourselves of this, the trivial statement $a + 0 = a$ can be witnessed as an isomorphism between `Either a Void` and `a`.

```
sumUnitTo :: Either a Void -> a
sumUnitTo (Left a)  = a
sumUnitTo (Right v) = absurd v    ⋯⋯⋯⋯⋯⋯⋯⋯⋯    ❶


sumUnitFrom :: a -> Either a Void
sumUnitFrom = Left
```

The function `absurd` at ❶ has the type `Void -> a`. It's a sort of bluff saying "if you give me a `Void` I can give you anything you want." This is a promise that can never be fulfilled, but because there are no `Void`s to be had in the first place, we can't disprove such a claim.

Function types also have an encoding as statements about cardinality—they correspond to

exponentialization. To give an example, there are exactly four (2^2) inhabitants of the type `Bool -> Bool`. These functions are `id`, `not`, `const True` and `const False`. Try as hard as you can, but you won't find any other pure functions between `Bool`s!

More generally, the type `a -> b` has cardinality $|b|^{|a|}$. While this might be surprising at first—it always seems backwards to me—the argument is straightforward. For every value of `a` in the domain, we need to give back a `b`. But we can chose any value of `b` for every value of `a`—resulting in the following equality.

$$|\texttt{a -> b}| = \underbrace{|\texttt{b}| \times |\texttt{b}| \times \cdots \times |\texttt{b}|}_{|\texttt{a}|\text{times}} = |\texttt{b}|^{|\texttt{a}|}$$

Exercise 1.2-i

Determine the cardinality of `Either Bool (Bool, Maybe Bool) -> Bool`.

The inquisitive reader might wonder whether subtraction, division and other mathematical operations have meaning when applied to types. Indeed they do, but such things are hard, if not impossible, to express in Haskell. Subtraction corresponds to types with particular values removed, while division of a type makes some of its values equal (in the sense of being defined equally—rather than having an `Eq` instance which equates them.)

In fact, even the notion of differentiation in calculus has meaning in the domain of types. Though we will not discuss it further, the interested reader is encouraged to refer to Conor McBride's paper "The Derivative of a Regular Type is its Type of One-Hole Contexts."[**?**].

1.3 Example: Tic-Tac-Toe

I said earlier that being able to manipulate the algebra behind types is a mighty superpower. Let's prove it.

Imagine we wanted to write a game of tic-tac-toe. The standard tic-tac-toe board has nine spaces, which we could naively implement like this:

```
data TicTacToe a = TicTacToe
  { topLeft   :: a
  , topCenter :: a
  , topRight  :: a
  , midLeft   :: a
  , midCenter :: a
  , midRight  :: a
  , botLeft   :: a
  , botCenter :: a
  , botRight  :: a
  }
```

While such a thing works, it's rather unwieldy to program against. If we wanted to construct an empty board for example, there's quite a lot to fill in.

```
emptyBoard :: TicTacToe (Maybe Bool)
emptyBoard =
  TicTacToe
    Nothing Nothing Nothing
    Nothing Nothing Nothing
    Nothing Nothing Nothing
```

Writing functions like `checkWinner` turn out to be even more involved.

Rather than going through all of this trouble, we can

use our knowledge of the algebra of types to help. The first step is to perform a cardinality analysis on `TicTacToe`;

$$\begin{aligned} |\texttt{TicTacToe a}| &= \underbrace{|\texttt{a}| \times |\texttt{a}| \times \cdots \times |\texttt{a}|}_{9 \text{ times}} \\ &= |\texttt{a}|^{9} \\ &= |\texttt{a}|^{3 \times 3} \end{aligned}$$

When written like this, we see that `TicTacToe` is isomorphic to a function `(Three, Three) -> a`, or in its curried form: `Three -> Three -> a`. Of course, `Three` is any type with three inhabitants; perhaps it looks like this:

```
data Three = One | Two | Three
  deriving (Eq, Ord, Enum, Bounded)
```

Due to this isomorphism, we can instead represent `TicTacToe` in this form:

```
data TicTacToe2 a = TicTacToe2
  { board :: Three -> Three -> a
  }
```

And thus simplify our implementation of `emptyBoard`:

```
emptyBoard2 :: TicTacToe2 (Maybe Bool)
emptyBoard2 =
  TicTacToe2 $ const $ const Nothing
```

Such a transformation doesn't let us do anything we couldn't have done otherwise, but it does drastically improve the ergonomics. By making this change, we are rewarded with the entire toolbox of combinators for

working with functions; we gain better compositionality and have to pay less of a cognitive burden.

Let us not forget that programming is primarily a human endeavor, and ergonomics are indeed a worthwhile pursuit. Your colleagues and collaborators will thank you later!

1.4 The Curry–Howard Isomorphism

Our previous discussion of the algebraic relationships between types and their cardinalities can be summarized in the following table.

Algebra	Logic	Types
$a + b$	$a \vee b$	`Either a b`
$a \times b$	$a \wedge b$	`(a, b)`
b^a	$a \implies b$	`a -> b`
$a = b$	$a \iff b$	*isomorphism*
0	$\bot$	`Void`
1	$\top$	`()`

This table itself forms a more-general isomorphism between mathematics and types. It's known as the *Curry–Howard isomorphism*—loosely stating that every statement in logic is equivalent to some computer program, and vice versa.

The Curry–Howard isomorphism is a profound insight about our universe. It allows us to analyze mathematical theorems through the lens of functional programming. What's better is that often even "boring" mathematical theorems are interesting when expressed as types.

To illustrate, consider the theorem $a^1 = a$. When viewed through Curry–Howard, it describes an

isomorphism between `() -> a` and `a`. Said another way, this theorem shows that there is no essential distinction between having a value and having a (pure) program that computes that value. This insight is the core principle behind why writing Haskell is such a joy compared with other programming languages.

Exercise 1.4-i

Use Curry–Howard to prove that $(a^b)^c = a^{b \times c}$. That is, provide a function of type `(b -> c -> a) -> (b, c) -> a`, and one of `((b, c) -> a) -> b -> c -> a`. Make sure they satisfy the equalities `to . from = id` and `from . to = id`. Do these functions remind you of anything from `Prelude`?

Exercise 1.4-ii

Give a proof of the exponent law that $a^b \times a^c = a^{b+c}$.

Exercise 1.4-iii

Prove $(a \times b)^c = a^c \times b^c$.

1.5 Canonical Representations

A direct corollary that any two types with the same cardinality are isomorphic, is that there are multiple ways to represent any given type. Although you

shouldn't necessarily let it change the way you model types, it's good to keep in mind that you have a choice.

Due to the isomorphism, all of these representations of a type are "just as good" as any other. However, as we'll see on page ??, it's often useful to have a conventional form when working with types generically. This *canonical representation* is known as a *sum of products*, and refers to any type `t` of the form,

$$t = \sum_m \prod_n t_{m,n}$$

The big Σ means addition, and the Π means multiplication—so we can read this as "addition on the outside and multiplication on the inside." We also make the stipulation that all additions must be represented via `Either`, and that multiplications via `(,)`. Don't worry, writing out the rules like this makes it seem much more complicated than it really is.

All of this is to say that each of following types is in its canonical representation:

- `()`
- `Either a b`
- `Either (a, b) (c, d)`
- `Either a (Either b (c, d))`
- `a -> b`
- `(a, b)`
- `(a, Int)`—we make an exception to the rule for numeric types, as it would be too much work to express them as sums.

But neither of the following types are in their canonical representation;

- `(a, Bool)`
- `(a, Either b c)`

As an example, the canonical representation of `Maybe a` is `Either a ()`. To reiterate, this doesn't mean you should prefer using `Either a ()` over `Maybe a`. For now it's enough to know that the two types are equivalent. We shall return to canonical forms in chapter 13.

Chapter 3

Variance

Consider the following type declarations. Which of them have viable `Functor` instances?

```
newtype T1 a = T1 (Int -> a)

newtype T2 a = T2 (a -> Int)

newtype T3 a = T3 (a -> a)

newtype T4 a = T4 ((Int -> a) -> Int)

newtype T5 a = T5 ((a -> Int) -> Int)
```

Exercise 3-i

Which of these types are `Functor`s? Give instances for the ones that are.

Despite all of their similarities, only `T1` and `T5` are `Functor`s. The reason behind this is one of *variance*: if we can transform an `a` into a `b`, does that mean we can necessarily transform a `T a` into a `T b`?

As it happens, we can sometimes do this, but it has a great deal to do with what `T` looks like. Depending on the shape of `T` (of kind $\text{TYPE} \to \text{TYPE}$) there are three possibilities for `T`'s variance:[1]

1. *Covariant*: Any function `a -> b` can be lifted into a function `T a -> T b`.

2. *Contravariant*: Any function `a -> b` can be lifted into a function `T b -> T a`.

3. *Invariant*: In general, functions `a -> b` cannot be lifted into a function over `T a`.

Covariance is the sort we're most familiar with—it corresponds directly to `Functor`s. And in fact, the type of `fmap` is exactly witness to this "lifting" motion `(a -> b) -> T a -> T b`. A type `T` is a `Functor` if and only if it is covariant.

Before we get to when is a type covariant, let's first look at contravariance and invariance.

The `contravariant`[?] and `invariant`[?] packages, both by Ed Kmett, give us access to the `Contravariant` and `Invariant` classes. These classes are to their sorts of variance as `Functor` is to covariance.

A contravariant type allows you to map a function *backwards* across its type constructor.

[1] Precisely speaking, variance is a property of a type in relation to one of its type-constructors. Because we have the convention that `map`-like functions transform the last type parameter, we can unambiguously say "`T` is contravariant" as a short-hand for "`T a` is contravariant with respect to `a`."

```haskell
class Contravariant f where
  contramap :: (a -> b) -> f b -> f a
```

On the other hand, an invariant type `T` allows you to map from `a` to `b` if and only if `a` and `b` are isomorphic. In a very real sense, this isn't an interesting property—an isomorphism between `a` and `b` means they're already the same thing to begin with.

```haskell
class Invariant f where
  invmap :: (a -> b) -> (b -> a) -> f a -> f b
```

The variance of a type `T a` with respect to its type variable `a` is fully specified by whether `a` appears solely in *positive position*, solely in *negative position* or in a mix of both.

Type variables which appear exclusively in positive position are covariant. Those exclusively in negative position are contravariant. And type variables which find themselves in both become invariant.

But what *is* a positive or negative position? Recall that all types have a canonical representation expressed as some combination of `(,)`, `Either` and `(->)`. We can therefore define positive and negative positions in terms of these fundamental building blocks, and develop our intuition afterwards.

Type	Position of	
	`a`	`b`
`Either a b`	+	+
`(a, b)`	+	+
`a -> b`	−	+

The conclusion is clear—our only means of introducing type variables in negative position is to put them on the left-side of an arrow. This should correspond to your intuition that the type of a function goes "backwards" when pre-composed with another function.

In the following example, pre-composing with `show :: Bool -> String` transforms a type `String -> [String]` into `Bool -> [String]`.

GHCi

```
> :t words
words :: String -> [String]

> :t show :: Bool -> String
show :: Bool -> String :: Bool -> String

> :t words . (show :: Bool -> String)
words . (show :: Bool -> String) :: Bool
   ↪ -> [String]
```

Mathematically, things are often called "positive" and "negative" if their signs follow the usual laws of multiplication. That is to say, a positive multiplied by a positive remains positive, a negative multiplied with a positive is a negative, and so on.

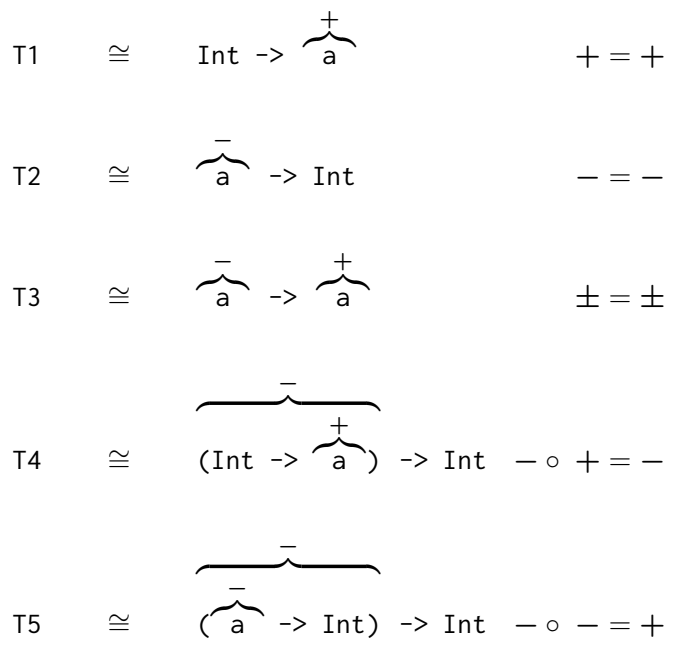
Variances are no different. To illustrate, consider the type `(a, Bool) -> Int`. The `a` in the subtype `(a, Bool)` is in positive position, but `(a, Bool)` is in negative position relative to `(a, Bool) -> Int`. As we remember from early arithmetic in school, a positive times a negative is

negative, and so `(a, Bool) -> Int` is contravariant with respect to `a`.

This relationship can be expressed with a simple table—but again, note that the mnemonic suggested by the name of positive and negative positions should be enough to commit this table to memory.

a	b	a ∘ b
+	+	+
+	−	−
−	+	−
−	−	+

We can use this knowledge to convince ourselves why `Functor` instances exist only for the `T1` and `T5` types defined above.



This analysis also shows us that `T2` and `T4` have `Contravariant` instances, and `T3` has an `Invariant` one.

A type's variance also has a more concrete interpretation: variables in positive position are *produced* or *owned*, while those in negative position are *consumed*. Products, sums and the right-side of an arrow are all pieces of data that already exist or are produced, but the type on the left-side of an arrow is indeed consumed.

There are some special names for types with multiple type variables. A type that is covariant in two arguments (like `Either` and `(,)`) is called a *bifunctor*. A type that is contravariant in its first argument, but covariant in its second (like `(->)`) is known as a *profunctor*. As you might imagine, Ed Kmett has packages which provide both of these typeclasses—although `Bifunctor` now exists in `base`.

Positional analysis like this is a powerful tool—it's quick to eyeball, and lets you know at a glance which class instances you need to provide. Better yet, it's impressive as hell to anyone who doesn't know the trick.

Part II

Lifting Restrictions

Chapter 4

Working with Types

4.1 Type Scoping

Haskell uses (a generalization of) the Hindley–Milner type system. One of Hindley–Milner's greatest contributions is its ability to infer the types of programs—without needing any explicit annotations. The result is that term-level Haskell programmers rarely need to pay much attention to types. It's often enough to just annotate the top-level declarations. And even then, this is done more for our benefit than the compiler's.

This state of affairs is ubiquitous and the message it sends is loud and clear: "types are something we need not think much about". Unfortunately, such an attitude on the language's part is not particularly helpful for the type-level programmer. It often goes wrong—consider the following function, which doesn't compile *because* of its type annotation:

```
broken :: (a -> b) -> a -> b
broken f a = apply
  where
    apply :: b
```

```
    apply = f a
```

The problem with `broken` is that, despite all appearances, the type `b` in `apply` is not the same `b` in `broken`. Haskell thinks it knows better than us here, and introduces a new type variable for `apply`. The result of this is effectively as though we had instead written the following:

```
broken :: (a -> b) -> a -> b
broken f a = apply
  where
    apply :: c
    apply = f a
```

Hindley–Milner seems to take the view that types should be "neither seen nor heard," and an egregious consequence of this is that type variables have no notion of scope. This is why the example fails to compile—in essence we've tried to reference an undefined variable, and Haskell has "helpfully" created a new one for us. The Haskell Report provides us with no means of referencing type variables outside of the contexts in which they're declared.

There are several language extensions which can assuage this pain, the most important one being `-XScopedTypeVariables`. When enabled, it allows us to bind type variables and refer to them later. However, this behavior is only turned on for types that begin with an explicit `forall` quantifier. For example, with `-XScopedTypeVariables`, `broken` is still broken, but the following works:

```
working :: forall a b. (a -> b) -> a -> b
```

```
working f a = apply
  where
    apply :: b
    apply = f a
```

The `forall a b.` quantifier introduces a type scope, and exposes the type variables `a` and `b` to the remainder of the function's definition. This allows us to reuse `b` when adding the type signature to `apply`, rather than introducing a *new* type variable as it did before.

`-XScopedTypeVariables` lets us talk about types, but we are still left without a good way of *instantiating* types. If we wanted to specialize `fmap` to `Maybe`, for example, the only solution sanctioned by the Haskell Report is to add an inline type signature.

If we wanted to implement a function that provides a `String` corresponding to a type's name, it's unclear how we could do such a thing. By default, we have no way to explicitly pass type information, and so even *calling* such a function would be difficult.

Some older libraries often use a `Proxy` parameter in order to help with these problems. Its definition is this:

```
data Proxy a = Proxy
```

In terms of value-level information content, `Proxy` is exactly equivalent to the unit type `()`. But it also has a phantom type parameter `a`, whose only purpose is to allow users to keep track of a type, and pass it around like a value.

For example, the module `Data.Typeable` provides a mechanism for getting information about types at runtime. This is the function `typeRep`, whose type is `Typeable a => Proxy a -> TypeRep`. Again, the `Proxy`'s only

purpose is to let `typeRep` know which type representation we're looking for. As such, `typeRep` has to be called as `typeRep (Proxy :: Proxy Bool)`.

4.2 Type Applications

Clearly, Haskell's inability to directly specify types has ugly user-facing ramifications. The extension `-XTypeApplications` patches this glaring issue in the language.

`-XTypeApplications`, as its name suggests, allows us to directly apply types to expressions. By prefixing a type with an `@`, we can explicitly fill in type variables. This can be demonstrated in GHCi:

GHCi

```
> :set -XTypeApplications

> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f
   ↪ b

> :t fmap @Maybe
fmap @Maybe :: (a -> b) -> Maybe a ->
   ↪ Maybe b
```

While `fmap` lifts a function over any functor `f`, `fmap @Maybe` lifts a function over `Maybe`. We've applied the type `Maybe` to the polymorphic function `fmap` in the same way we can apply value arguments to functions.

There are two rules to keep in mind when thinking about type applications. The first is that types are applied in the same order they appear in a type signature—including its context and `forall` quantifiers. This means that applying a type `Int` to `a -> b -> a` results in `Int -> b -> Int`. But type applying it to `forall b a. a -> b -> a` is in fact `a -> Int -> a`.

Recall that typeclass methods have their context at the beginning of their type signature. `fmap`, for example, has type `Functor f => (a -> b) -> f a -> f b`. This is why we were able to fill in the functor parameter of `fmap`—because it comes first!

The second rule of type applications is that you can avoid applying a type with an underscore: `@_`. This means we can also specialize type variables which are not the first in line. Looking again at GHCi, we can type apply `fmap`'s `a` and `b` parameters while leaving `f` polymorphic:

GHCi

```
> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f
    ↪ b

> :t fmap @_ @Int @Bool
fmap @_ @Int @Bool :: Functor w => (Int
    ↪ -> Bool) -> w Int -> w Bool
```

Because types are applied in the order they're defined, in the presence of `-XTypeApplications` types become part of a public signature. Changing the order of type variables can break downstream code, so be careful when performing refactors of this nature.

Pay attention to type order whenever you write a function that might be type applied. As a guiding principle, the hardest types to infer must come first. This will often require using `-XScopedTypeVariables` and an explicitly scoped `forall`.

`-XTypeApplications` and `-XScopedTypeVariables` are the two most fundamental extensions in a type-programmer's toolbox. They go together hand in hand.

4.3 Ambiguous Types

Returning again to the example of `Data.Typeable`'s `typeRep` function, we can use it to implement a function that will give us the name of a type. And we can do so without requiring the `Proxy` parameter.

```
typeName :: forall a. Typeable a => String  ...  ❶
typeName = show . typeRep $ Proxy @a  ......  ❷
```

There are two interesting things to note in `typeName`. At ❷, `Proxy @a` is written as shorthand for `Proxy :: Proxy a`—this is because the `Proxy` data constructor has type `Proxy t`. The type variable `t` here is the first one in its type signature, so we're capable of type applying it. Type applications aren't reserved for functions, they can be used anywhere types are present.

At ❶ we see that the type `a` doesn't actually appear to the right of the fat context arrow (`=>`). Because Hindley–Milner's type inference only works to the right of the context arrow, it means the type parameter `a` in `typeName` can never be correctly inferred. Haskell refers to such a type as being *ambiguous*.

By default, Haskell will refuse to compile any

programs with ambiguous types. We can bypass this behavior by enabling the aptly-named `-XAllowAmbiguousTypes` extension anywhere we'd like to define one. Actually *using* code that has ambiguous types, will require `-XTypeApplications`.

The two extensions are thus either side of the same coin. `-XAllowAmbiguousTypes` allows us to define ambiguously typed functions, and `-XTypeApplications` enables us to call them.

We can see this for ourselves. By enabling `-XAllowAmbiguousTypes`, we can compile `typeName` and play with it.

GHCi

```
> :set -XTypeApplications

> typeName @Bool
"Bool"

> typeName @String
"[Char]"

> typeName @(Maybe [Int])
"Maybe [Int]"
```

Though this is a silly example, ambiguous types are very useful when doing type-level programming. Often we'll want to get our hands on a term-level representation of types—think about drawing a picture of a type, or about a program that will dump a schema of a type. Such a function is almost always going to be ambiguously typed, as we'll see soon.

However, ambiguous types aren't always this obvious to spot. To compare, let's look at a surprising example. Consider the following type family:

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

Given this definition, are all of the following type signatures non-ambiguous? Take a second to think through each example.

1. `AlwaysUnit a -> a`
2. `b -> AlwaysUnit a -> b`
3. `Show a => AlwaysUnit a -> String`

The third example here is, in fact, ambiguous. But why? The problem is that it's not clear which `Show a` instance we're asking for! Even though there is an `a` in `Show a => AlwaysUnit a -> String`, we're unable to access it—`AlwaysUnit a` is equal to `()` for all `a`s!

More specifically, the issue is that `AlwaysUnit` doesn't have an inverse; there's no `Inverse` type family such that `Inverse (AlwaysUnit a)` equals `a`. In mathematics, this lack of an inverse is known as *non-injectivity*.

Because `AlwaysUnit` is non-injective, we're unable to learn what `a` is, given `AlwaysUnit a`.

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 forwards doesn't mean we can also come back again.

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.

Chapter 5

Constraints and GADTs

5.1 Introduction

CONSTRAINTs are odd. They don't behave like TYPEs nor like promoted data kinds. They are a fundamentally different thing altogether, and thus worth studying.

The CONSTRAINT kind is reserved for things that can appear on the left side of the fat context arrow (`=>`). This includes fully-saturated typeclasses (like `Show a`), tuples of other CONSTRAINTs, and type equalities (`Int ~ a`.) We will discuss type equalities in a moment.

Typeclass constraints are certainly the most familiar. We use them all the time, even when we are not writing type-level Haskell. Consider the equality function `(==) :: Eq a => a -> a -> Bool`. Tuples of CONSTRAINTs are similarly well-known: `sequenceA :: (Applicative f, Traversable t) => t (f a) -> f (t a)`.

Type equalities are more interesting, and are enabled via `-XGADTs`. Compare the following two programs:

```
five :: Int
five = 5
```

```
five_ :: (a ~ Int) => a
five_ = 5
```

Both `five` and `five_` are identical as far as Haskell is concerned. While `five` has type `Int`, `five_` has type `a`, along with a constraint saying that `a` equals `Int`. Of course, nobody would actually write `five_`, but it's a neat feature of the type system regardless.

Type equalities form an equivalence relation, meaning that they have the following properties:

- *reflexivity*—a type is always equal to itself: `a ~ a`
- *symmetry*—`a ~ b` holds if and only if `b ~ a`
- *transitivity*—if we know both `a ~ b` and `b ~ c`, we (and GHC) can infer that `a ~ c`.

5.2 GADTs

Generalized algebraic datatypes (GADTs; pronounced "gad-its") are an extension to Haskell's type system that allow explicit type signatures to be written for data constructors. They, like type equality constraints, are also enabled via `-XGADTs`.

The canonical example of a GADT is a type safe syntax tree. For example, we can declare a small language with integers, booleans, addition, logical negation, and if statements.

```
data Expr a where                    ❶
  LitInt  :: Int -> Expr Int         ❷
  LitBool :: Bool -> Expr Bool
  Add     :: Expr Int -> Expr Int -> Expr Int
```

```
  Not      :: Expr Bool -> Expr Bool
  If       :: Expr Bool -> Expr a -> Expr a -> Expr a   ·
  ↪ ❸
```

The `where` at ❶ is what turns on GADT syntax for the rest of the declaration. Each of `LitInt`, `LitBool`, `Add`, etc. corresponds to a data constructor of `Expr`. These constructors all take some number of arguments before resulting in an `Expr`.

For example, `LitInt` at ❷ takes an `Int` before returning a `Expr Int`. On the other hand, the data constructor `If` at ❸ takes three arguments (one `Expr Bool` and two `Expr a`s) and returns an `Expr a`.

It is this ability to specify the return type that is of particular interest.

You might be pleased that `Expr` is *correct by construction.* We are incapable of building a poorly-typed `Expr`. While this might not sound immediately remarkable, it is—we've reflected the *typing rules of* `Expr` in the type system of Haskell. For example, we're unable to build an AST which attempts to add an `Expr Int` to a `Expr Bool`.

To convince ourselves that the type signatures written in GADT syntax are indeed respected by the compiler, we can look in GHCi:

GHCi

```
> :t LitInt
LitInt :: Int -> Expr Int

> :t If
If :: Expr Bool -> Expr a -> Expr a ->
   ↪ Expr a
```

Because GADTs allow us to specify a data constructor's type, we can use them to *constrain* a type variable in certain circumstances. Such a thing is not possible otherwise.[1]

The value of GADTs is that Haskell can use the knowledge of these constrained types. In fact, we can use this to write a typesafe evaluator over `Expr`:

```
evalExpr :: Expr a -> a
evalExpr (LitInt i)  = i   . . . . . .  ❶
evalExpr (LitBool b) = b   . . . . . .  ❷
evalExpr (Add x y)   = evalExpr x + evalExpr y
evalExpr (Not x)     = not $ evalExpr x
evalExpr (If b x y)  =
  if evalExpr b
     then evalExpr x
     else evalExpr y
```

In just this amount of code, we have a fully functioning little language and interpreter. Consider:

GHCi

```
> evalExpr . If (LitBool False) (LitInt
    ↪ 1) . Add (LitInt 5) $ LitInt 13
18

> evalExpr . Not $ LitBool True
```

[1] Or equivalently—as we will see—without type equalities.

```
  False
}
```

Pay careful attention here! At ❶, `evalExpr` returns an `Int`, but at ❷ it returns a `Bool`! This is possible because Haskell can *reason* about GADTs. In the `LitInt` case, the only way such a pattern could have matched is if `a ~ Int`, in which case it's certainly okay to return a `Int`. The reasoning for the other patterns is similar; Haskell can use information from inside a pattern match to drive type inference.

GADT syntax is indeed provided by `-XGADTs`, but it is not the syntax that is fundamentally interesting. The extension is poorly named—a more appropriate name might be "`-XTypeEqualities`". In fact, GADTs are merely syntactic sugar over type equalities. We can also declare `Expr` as a traditional Haskell datatype as follows:

```
data Expr_ a
  = (a ~ Int)  => LitInt_  Int
  | (a ~ Bool) => LitBool_ Bool
  | (a ~ Int)  => Add_ (Expr_ Int) (Expr_ Int)
  | (a ~ Bool) => Not_ (Expr_ Bool)
  | If_ (Expr_ Bool) (Expr_ a) (Expr_ a)
```

When viewed like this, it's a little easier to see what's happening behind the scenes. Each data constructor of `Expr_` carries along with it a type equality constraint. Like any constraint inside a data constructor, Haskell will require the constraint to be proven when the data constructor is called.

As such, when we pattern match on a data constructor which contains a constraint, this satisfied constraint *comes back into scope.* That is, a function of type `Expr a ->`

`a` can return an `Int` when pattern matching on `LitInt`, but return a `Bool` when matching on `LitBool`. The type equality constraining `a` only comes back into scope after pattern matching on the data constructor that contains it.

We will explore the technique of packing constraints inside data constructors in much greater generality later.

Though GADT syntax doesn't offer anything novel, we will often use it when defining complicated types. This is purely a matter of style as I find it more readable.

5.3 Heterogeneous Lists

Necessary Extensions

```
{-# LANGUAGE ConstraintKinds      #-}
{-# LANGUAGE DataKinds            #-}
{-# LANGUAGE GADTs                #-}
{-# LANGUAGE ScopedTypeVariables  #-}
{-# LANGUAGE TypeApplications     #-}
{-# LANGUAGE TypeFamilies         #-}
{-# LANGUAGE TypeOperators        #-}
{-# LANGUAGE UndecidableInstances #-}
```

Necessary Imports

```
import Data.Kind (Constraint, Type)
```

One of the primary motivations of GADTs is building inductive type-level structures out of term-level data. As a working example for this section, we can use GADTs to define a heterogeneous list—a list which can store values of different types inside it.

To get a feel for what we'll build:

GHCi

```
> :t HNil
HNil :: HList '[]

> :t True :# HNil
True :# HNil :: HList '[Bool]

> let hlist = Just "hello" :# True :# HNil

> :t hlist
hlist :: HList '[Maybe [Char], Bool]

> hLength hlist
2
```

The `HNil` constructor here is analogous to the regular list constructor `[]`. `(:#)` likewise corresponds to `(:)`. They're defined as a GADT:

```
data HList (ts :: [Type]) where  . . . . . .  ❶
  HNil :: HList '[]  . . . . . .  ❷
  (:#) :: t -> HList ts -> HList (t ': ts)  . .  ❸
infixr 5 :#
```

At ❶, you'll notice that we've given `HList`'s `ts` an explicit kind signature. The type parameter `ts` is defined to have kind `[TYPE]`, because we'll store the contained types inside of it. Although this kind signature isn't strictly necessary—GHC will correctly infer it for us—your future self will appreciate you having written it. A good rule of thumb is to annotate *every* kind if *any* of them isn't `TYPE`.

`HList` is analogous to the familiar `[]` type, and so it needs to define an empty list at ❷ called `HNil`, and a cons operator at ❸ called `(:#)`.[2] These constructors have carefully chosen types.

`HNil` represents an empty `HList`. We can see this by the fact that it takes nothing and gives back `ts` ∼ `'[]`—an empty list of types.

The other data constructor, `(:#)`, takes two parameters. Its first is of type `t`, and the second is a `HList ts`. In response, it returns a `HList (t ': ts)`—the result is this new type has been consed onto the other `HList`.

This `HList` can be pattern matched over, just like we would with regular lists. For example, we can implement a length function:

```
hLength :: HList ts -> Int
hLength HNil      = 0
hLength (_ :# ts) = 1 + hLength ts
```

But, having this explicit list of types to work with, allows us to implement much more interesting things. To illustrate, we can write a *total* `head` function—something impossible to do with traditional lists.

```
hHead :: HList (t ': ts) -> t
hHead (t :# _) = t
```

The oddities don't stop there. We can deconstruct any length-3 `HList` whose second element is a `Bool`, show it, and have the compiler guarantee that this is an acceptable

[2] Symbolically-named data constructors in Haskell must begin with a leading colon. Anything else is considered a syntax-error by the parser.

(if strange) thing to do.

```
showBool :: HList '[_1, Bool, _2] -> String
showBool (_ :# b :# _ :# HNil) = show b
```

Unfortunately, GHC's stock deriving machinery doesn't play nicely with GADTs—it will refuse to write `Eq`, `Show` or other instances. But we can write our own by providing a base case (for `HNil`), and an inductive case.

The base case is that two empty `HList`s are always equal.

```
instance Eq (HList '[]) where
  HNil == HNil = True
```

And inductively, two consed `HList`s are equal only if both their heads and tails are equal.

```
instance (Eq t, Eq (HList ts)) => Eq (HList (t ': ts))
 ↪  where
  (a :# as) == (b :# bs) = a == b && as == bs
```

Exercise 5.3-i

Implement `Ord` for `HList`.

Exercise 5.3-ii

Implement `Show` for `HList`.

The reason we had to write two instances for `Eq` was to assert that every element in the list also had an `Eq` instance. While this works, it is rather unsatisfying. Alternatively, we can write a closed type family which will fold `ts` into a big CONSTRAINT stating each element has an `Eq`.

```
type family AllEq (ts :: [Type]) :: Constraint where
  AllEq '[]       = ()                    ❶
  AllEq (t ': ts) = (Eq t, AllEq ts)      ❷
```

As `AllEq` is our first example of a non-trivial closed type family, we should spend some time analyzing it. `AllEq` performs type-level pattern matching on a list of types, determining whether or not it is empty.

If it is empty—line ❶—we simply return the unit CONSTRAINT. Note that because of the kind signature on `AllEq`, Haskell interprets this as CONSTRAINT rather than the unit TYPE.

However, if `ts` is a promoted list cons, we instead construct a CONSTRAINT-tuple at ❷. You'll notice that `AllEq` is defined inductively, so it will eventually find an empty list and terminate. By using the `:kind!` command in GHCi, we can see what this type family expands to.

GHCi

```
> :kind! AllEq '[Int, Bool]
AllEq '[Int, Bool] :: Constraint
= (Eq Int, (Eq Bool, () :: Constraint))
```

`AllEq` successfully folds [TYPE]s into a CONSTRAINT. But

there is nothing specific to `Eq` about `AllEq`! Instead, it can be generalized into a fold over any CONSTRAINT `c`. We will need `-XConstraintKinds` in order to talk about polymorphic constraints.

```
type family All (c :: Type -> Constraint)
                (ts :: [Type]) :: Constraint where
  All c '[]       = ()   . . . . . . . . . . . . . . .  ❶
  All c (t ': ts) = (c t, All c ts)   . . . . . . .  ❷
```

With `All`, we can now write our `Eq` instance more directly.

```
instance All Eq ts => Eq (HList ts) where
  HNil      == HNil      = True
  (a :# as) == (b :# bs) = a == b && as == bs
```

Exercise 5.3-iii

Rewrite the `Ord` and `Show` instances in terms of `All`.