A User's Guide to Ermine

Edition r140, 6 April 2014

Stephen Compall

This manual is for Ermine, which is a well-typed, nonstrict, functional programming language. (Edition r140, 6 April 2014)

Table of Contents

1 What is Ermine?

Ermine is a well-typed, nonstrict, functional programming language. Its key implementations are currently developed as a public free software project. Details about this project may be found on the web at the *ermine-language* group.

Most of the features of Ermine, from how it looks to how it behaves, are borrowed from Haskell. If you already know Haskell, you can get started writing Ermine with only minor changes; most of what you know about designing programs in Haskell applies directly to Ermine.

That said, it is not necessary to know Haskell, or any other programming language, to successfully use Ermine.

1.1 What's new about it?

Most of Ermine's authors understood Haskell very well before they decided to build Ermine. Why did they do so?

Ermine features a runtime-agnostic compiler architecture written in Haskell. By *runtime-agnostic*, we mean that the compiler's output is intended to be consumed by interpreters or compilers for JVM, Javascript, the CLR, or even native code. The Ermine compiler is also an example of how to build an elegant pure compiler using the most cutting-edge Haskell features and libraries.

The original version of Ermine, known as *Ermine Legacy*, grew up as a system for highly customizable and type-safe data processing, integrated with a Java codebase. It demonstrates the value of type system research in business-oriented data processing, and that it is not necessary to sacrifice language features and highly generic programming for interoperability to build a programming language that can integrate with feature-starved virtual machines like the JVM.

Besides sharing several of the type system extensions to Haskell implemented by GHC, Ermine features a *rowtype* system. So far, the main use of this new type system feature has been in a type-safe multi-source database query engine.

1.2 Who is it for?

The audience of Ermine, and so of this book, has always been twofold.

Some of its users are interested in collecting, analyzing, and inspecting data from various sources, but aren't necessarily programmers. The type system here is key: in an Ermine program, types provide a guide to what data is available where, and how it can be combined. Nonstrictness, and lack of side-effects, frees the Ermine user and integrator from concerns about order of operations.

As these features guide you toward correct Ermine programs, it is not truly necessary for you to be an expert in program design to use Ermine successfully. Building new programs is a matter of declaring data sources and following the types as you combine them. Modifying existing programs is a matter of convincing the type system that your changes make sense.

The others in the audience are functional programmers who wish to build elegant, highly generalized, well-typed programs, who abhor boilerplate, and who want advanced type system features and high-quality inference to help them in these goals.

1.3 About the book

This book is a "ground up" tutorial, based on understanding the simplest of concepts and building from there, not on digging through larger examples. It is suitable for users unfamiliar with any programming language.

If you come from a programming tradition outside the purely functional world, imagine that you are a new programmer, and keep an open mind; some of your ideas about programming may need to be "unlearned" to create beautiful Ermine programs.

However, we will try not to dwell on points longer than needed to explain them. Introductory sections are meant to be read in order, so I hope that if you do not understand a section, you will try giving it a more careful read, or ask for help from another programmer or the Ermine community, before continuing.[1]

The exception to this is *stretch* chapters, marked accordingly, which you may return to later, as you feel comfortable doing so. You may attempt them immediately, perhaps as a test of how far you can stretch your knowledge, but do not worry if you find them impenetrable at first; feel free to move on.

I hope this book is also useful to you if you are already well-acquainted with Haskell or functional programming, though. It can be frustrating to wade through basic material that feels like so much old hat to you, so in addition to not dwelling on things, I have broken up the introductory material with stretch chapters as just described. You may wish to skim the introductory material, trying examples as you go, or simply skip to the stretch chapters, referring back as you have basic syntax questions.

[1] The main place for live Ermine discussion is the '`#ermine`' channel on '`irc.freenode.net`'.

2 A Taste of Ermine

This book describes the Ermine language and standard library; we begin with the most basic features, building on earlier concepts to explore more complex concepts as we go.

However, we're not really going to get started until the next chapter. Instead, I'd like to take a quick peek at what's to come in each chapter, in particular at some samples of Ermine code we'll be looking at.

The point isn't to understand everything included in this chapter; you can start to get a feel for Ermine, what it can do, and the kind of things that are possible with it, just by seeing it.

TODO: *A representative sample from each chapter.*

2.1 A taste of beginning

You'll learn how programs are constructed of the simplest of ideas, combined into more complex ideas. You'll also learn how thinking in harmony with Ermine helps you build safer and more general programming components, and try out two key tools for generality: the function and the polymorphic type. See Chapter 3 [At the Beginning], page 4.

This sample has not yet been tested; it may contain undiscovered errors or inaccuracies in expected output.

```
twoPlus x = (+) 2 x

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

fourPlus = compose twoPlus twoPlus
```

2.2 A taste of data

With bootstrapping done, you'll dive into the core of building Ermine programs: defining data types, and defining functions to operate on the data they describe. You'll encounter most of the simple, but useful data types common to most Ermine programs.

You'll see how to build unfinished programs and have Ermine tell you which parts are missing, and try out the first version of a tool for discovering where generalities can be extracted from your Ermine programs.

This sample has not yet been tested; it may contain undiscovered errors or inaccuracies in expected output.

```
data Stream a = SCons a (Stream a)

fortyTwos = SCons 42 fortyTwos

foldrS : (a -> b -> b) -> Stream a -> b
foldrS f (SCons a sa) = f a (foldrS f sa)
```

3 At the Beginning

The key concept in the organization of Ermine programs is the *definition*. As an Ermine user, you come up with an idea, give it a name, and write a definition that describes what the name means in your program.

It is possible to create your entire program as a single definition, but this is very impractical; instead, you write definitions of ideas that are much simpler than your program, and other definitions that explain how these ideas can work together. As such, an Ermine program is a set of definitions.

3.1 First Definition

You organize your definitions into files, called *modules*, mostly as you see fit; you will be working with single modules at a time for now.

Create a file[1] called `ChapterBeginning.e` with these contents:

```
module ChapterBeginning where

ft = 42
```

You've just defined the module called `ChapterBeginning`, in which the name `ft` means `42`.

Save the file, and load it in the Ermine console.

```
>> :load path/to/ChapterBeginning.e
 ⊣ Importing module 'ChapterBeginning' (0.02 seconds)
>>
```

Now you can ask Ermine for the meaning of the definition `ft`.

```
>> ft
⇒ res0 : Int = 42
```

3.2 Types and Definitions

However, there's a deeper meaning to `ft`. We can ask Ermine for its *type*, which explains what sort of thing `ft` is.

```
>> :t ft
⇒ Int
```

We say that "`Int` is `ft`'s type", where `Int` is short for "integer".[2] A type has the strength of scientific proof: when Ermine gives a type, this is something it *knows* about that code. It is also how *people* think about their definitions in Ermine. These types reveal truths about the definitions you write that are not often apparent simply by reading the definitions.

---

[1] You will not want to use a word processor; they are designed for prose, not programming. If you are comfortable with a programmer's text editor already, use that. Otherwise, gedit is a good, super-simple editor to start with. You can pick View→Highlight Mode→Sources→Haskell to have gedit pleasantly color your Ermine code; most programmers work with such "syntax highlighting".

[2] In Ermine Legacy, `Int` is actually "integer between −2,147,483,648 and 2,147,483,647, inclusive". The reason for this is not very interesting right now.

3.3 Trying things out

The Ermine console, into which you loaded `ChapterBeginning.e`, is known as the REPL.[3]

In a definition like `ft = 42`, the right-hand side (RHS) of the equals sign is an *expression*. In Ermine, every expression has a type, and definitions can be used when forming expressions.

You have seen two expressions so far: `ft`, and `42`. The REPL works on expressions and simplifies them to a final form, showing the result. This is an easy way to try out expressions without putting them into a module, and it is how you will experiment throughout the rest of this book.

```
>> 42
⇒ res2 : Int = 42
>> ft
⇒ res3 : Int = 42
```

Here, `42` has already been simplified to a final form, so it is simply displayed. However, `ft` can be simplified, by looking at its definition, to `42`, so Ermine does that first.

The '`:t`' command you have already seen actually operates on expressions, not definitions. It just happens that when you write a definition, the expression that is just that definition's name has the same type.

```
>> :t 42
⇒ Int
>> :t ft
⇒ Int
```

3.4 Another type

Here is another expression and another type.

```
>> :t "hello, world"
⇒ String
>> "hello, world"
⇒ res4 : String = hello, world
```

Here, `String` means some written text, such as this sentence.[4] Add this definition to your module:

```
module ChapterBeginning where

ft = 42

hw = "hello, world"
```

Now you can reload the module and see the new definition.

```
>> :r
⊣ Reloaded one dirty module (one exotic file) while retaining 90 modules (0.01 seconds)
>> :t hw
⇒ String
```

[3] Read-Eval-Print Loop, for reasons not interesting for the purposes of this book.

[4] `String` is *not* an alias for `[Char]` as it is in Haskell.

```
>> hw
⇒ res0 : String = hello, world
```

There will be places in your program where `String` is called for, and places where `Int` is called for. For example, it makes sense to multiply an `Int` by another `Int`. However, it does not make sense to multiply a `String` by another `String`. So if you wish to multiply, you know—and Ermine knows—that a `String` is not appropriate; likewise, if you want to combine two `String`s, you know—and Ermine knows—that multiplication is not an appropriate thing to do.

3.5 Thinking in types

Though you are thinking about very simple types now, you will use the same way of thinking about how to combine pieces of your programs, even as your programs get very sophisticated. This harmony between how *you* think about your programs, and what *Ermine* knows about your programs, is the key to discovering their beauty. This is called *thinking in types*.

Recall that programs consist of many definitions: each definition has a type. Recall that you must describe how the definitions fit together; if you think about them in terms of their types, and use the types to prove that the way you have fit them together makes some kind of sense, then by checking types, Ermine checks that your thinking is sound.

In this way, you can reach very high levels of sophistication in your definitions, secure in the knowledge that Ermine's type checker will steer you in the right direction. The surprising outcome of this effect is that as your types become more carefully considered, your programs will become both more beautiful and safer.[5]

3.6 Generalization

The essence of fine Ermine programs is *generality*. There is no need for a custom-built tool for a job if you can assemble general-purpose pieces to do it. Even if you have only one specific task in mind, it pays in several ways to discover how your specific program can separate into generalized components that are later combined.

Your journey through the rest of this book, and the broader world of Ermine, will be one of discovering new generalizations. Ermine was conceived by way of new research in program generality; this research is ongoing, so nobody knows all the methods of generalization you might one day use!

Let's start with one of our most effective generalization tools. A *function* takes a piece of data, transforms it in some way, and produces a result. The result may be of the same type, or a different type.

We spoke earlier of "combining" `Int`s and `String`s. Functions are the tools you use for these combinations. Ermine comes with many modules of generalized functions that you can use to build your programs. One of them is in a definition called `(+)`.

```
>> :t (+) ft
⇒ Int -> Int
```

`(+)` needs *two* `Int`s; when you supply one as an *argument*, the type shows you that you need one more `Int`, and the function will "take an `Int` to an `Int`". You say the type of

[5] We will explore this idea in Section 3.11 [Safety in polymorphism], page 10.

the function `(+) ft`, `Int -> Int`, out loud as "int to int". You say that you are *calling* a function when you supply an argument to obtain a result.

Let's try supplying something that doesn't make sense.

```
>> :t (+) ft hw
error  <interactive>:1:1: error: failed to unify type Int with type String
error  (+) ft hw<EOF>
error  ^
```

Here's your first type error; it's just Ermine saying that what you tried to do makes no sense.

For now, don't worry about the meaning of type errors. Just treat it as a gentle indicator that you should try something different. We'll talk about what they mean, and how they guide you to the right changes in faulty code, in Chapter 6 [Understanding type errors], page 31.

So let's follow Ermine's advice and supply things that do make sense.

```
>> :t (+) ft 1
⇒ Int
>> (+) ft 1
⇒ res1 : Int = 43
>> :t (+) ft ((+) ft 1)
⇒ Int
>> :t (+) ((+) ft 1) ((+) ft 1)
⇒ Int
```

Here you have used `(+)` multiple times, even using it to produce new arguments for other uses of `(+)`. The whole thing is an expression; each argument is also an expression, and you can use parentheses to disambiguate.

3.7 Defining functions

Your own definitions will almost all be functions, and they will be defined using expressions that refer to other functions.[6] some of these others will have been defined by you, and some will be supplied by Ermine.

Let's use `(+)` to define a few functions, importing the `Num` module to get access to `(+)`. Update your `ChapterBeginning.e` as follows:

```
module ChapterBeginning where

import Num

ft = 42

hw = "hello, world"

twoPlus x = (+) 2 x
```

[6] Defining functions and giving them names at the top level is not the only way to create functions that you need. See Section 7.2 [Lambda expressions], page 45.

```
twoPlusInfix x = 2 + x
```

Now, reload, and try out `twoPlus`.

```
>> :r
⊣ Reloaded one dirty module (one exotic file) while retaining 90 modules (0.02 seconds)
>> :t twoPlus
⇒ Int -> Int
>> :t twoPlus 3
⇒ Int
>> twoPlus ft
⇒ res0 : Int = 44
```

The `x` on the left says where the argument appears in a call to the function; you can talk about `x` in the RHS, and it will be replaced with the argument you supplied.

Recall that Ermine simplifies to a final form by looking up definitions and translating from the left side of the `=` to the right. Here's what that looks like for `twoPlus ft`, step by step.[7]

```
twoPlus ft
(+) 2 ft
<(+) 2 called with ft>
<(+) called with 2, called with ft>
<(+) called with 2, called with 42>
44
```

`twoPlusInfix` is exactly like `twoPlus`, but demonstrates that expressions like `(+) x y`, when the function has a special name, can be rewritten as `x + y`. This is called *infix notation*.

3.8 Multiple arguments

When you defined `twoPlus`, you saw it had the type `Int -> Int`. However, when you looked at the type `(+) ft` earlier, you saw it had the same type! It is not necessary to have a placeholder on the left-hand side of a definition to create a function.

```
>> :t (+) : Int -> Int -> Int
⇒ Int -> Int -> Int
```

Here after the colon, you've written an assertion about what you expect the type to be; since the assumption is correct, it just gets written back to you. This reads "int to int to int". It means:

- A function that takes an `Int` to...
  - a function that takes an `Int` to an `Int`.

When you supply an `Int` to `(+)`, it produces a function. Just like `Int` and `String`, functions with types like `Int -> Int` really exist in Ermine, and may be given as arguments or returned from functions.

You saw this earlier when you checked the type of `(+) ft`, which was `Int -> Int`. But `twoPlus` has the same type!

[7] If you are familiar with languages outside the Haskell tradition, the fact that `ft` is replaced so late may surprise you. This *nonstrictness* is by design.

To the end of `ChapterBeginning.e`, add these definitions:

```
twoPlusPF = (+) 2

twoPlusab x y = twoPlus ((+) x y)
```

Let's see how `twoPlus` and `twoPlusPF` compare.

```
>> :r
⊣ Reloaded one dirty module (one exotic file) while retaining 90 modules (0.03 seconds)
>> :t twoPlus
⇒ Int -> Int
>> :t twoPlusPF
⇒ Int -> Int
```

Try calling them; they are indistinguishable.

`twoPlusab` shows how you can define your own functions with types that look like `(+)` in that they produce intermediate functions when we supply arguments to them.

```
>> :t twoPlusab
⇒ Int -> Int -> Int
>> :t twoPlusab 1
⇒ Int -> Int
>> twoPlusab 1 1
⇒ res0 : Int = 4
```

3.9 Combining functions

You've created several new functions by using the functions `(+)` and your own `twoPlus` in different ways. You can see how `twoPlus` could be a reusable component for other definitions, such as for `twoPlusab`.

It's time to go deeper. Add this definition to `ChapterBeginning.e`.[8]

```
compose f g x = f (g x)
```

You've called two functions, `f` and `g`, in the RHS here, but they do not come from other definitions; they are arguments. What would the type of this be?[9]

```
>> :r
⊣ Reloaded one dirty module (one exotic file) while retaining 90 modules (0.04 seconds)
>> :t compose
⇒ forall a b c. (a -> b) -> (c -> a) -> c -> b
```

3.10 Reading generic types

The types you've encountered so far, `Int`, `String`, and function types denoted by `->`, have been symbolic or capitalized because they represent particular known types. But with `compose`, you've generalized one step further than you ever have before: `compose` has *many* possible types.

---

[8] Ermine provides `compose` for you; it is called `(.)`. Your definition here is just as good, though.

[9] If you're familiar with polymorphic functions in languages like Java, two things to note here: this level of generality, including all 3 type "parameters", was discovered automatically by Ermine, and polymorphism is not limited to functions. See Section 4.8 [Value polymorphism], page 17.

These *polymorphic* types are important because it would be silly to write a `compose` that only worked on `Int`s, one that worked on `String`s, one that worked on `Int`s at the ends but `String` in the middle, and so on.

Start by adding an expected type, or *signature*, for `compose` to `ChapterBeginning.e`. This can serve as a form of documentation, and may be easier to read than what Ermine just told you.

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

Compared to the type Ermine figured out by itself, `(a -> b) -> (c -> a) -> c -> b`, we've renamed the *type variables* `a`, `b`, and `c` to `b`, `c`, and `a`, respectively, and put parentheses around the final `a -> c`. To Ermine, this means exactly the same thing; it's just easier to understand like this.

What does this type mean? If we decide what `a`, `b`, and `c` are, and we know how to take `b` to `c`, and also how to take `a` to `b`, we know how to take `a` to `c` by just using both of those functions. And that is exactly what the definition looks like: we feed `g x` to `f`.

```
>> :t compose ((+) 42)
⇒ forall a. (a -> Int) -> a -> Int
```

By supplying the `f` argument, you set the type variables `b` and `c` to `Int`, because the type of the argument you supplied is `Int -> Int`, which fits the pattern of `b -> c` in our polymorphic type in just that way. The result is still polymorphic, because it still makes sense to set the type variable `a` to anything you like.

Most of the functions we discuss throughout the rest of this book will be polymorphic. Moreover, every new type system feature we discuss from now on will involve polymorphism in some way. Looking at the level of polymorphism in your program's definitions is a good smoke-test for how well you have generalized.

3.11 Safety in polymorphism

It's obvious how a polymorphic `compose` is more generally usable than one that only works on `Int` functions. Let's see why that is.

Keep in mind that the *only* thing `compose` is supposed to do is feed the result of one function into another.

Add these definitions to `ChapterBeginning.e`:

```
composeInt : (Int -> Int) -> (Int -> Int) -> (Int -> Int)
composeInt f g x = f (g x)

fourPlus = compose twoPlus twoPlus

fourPlusInt = composeInt twoPlus twoPlus
```

Now, try to do something naughty with `composeInt`. Perhaps you could add 5 to the result of `f`. Maybe you could just not use `f` or `g`, or both, in the RHS. Use `:r` to see if the Ermine type checker accepts your changes. If Ermine will accept it, try to make the same changes to `compose`.

The point is to try and screw up `fourPlusInt`, *only* by changing `composeInt`, and likewise with `fourPlus` and `compose`. Obviously, they are expected to add four to their arguments. Try to get a different result than `7` in the below examples.

```
>> :r
⊣ Reloaded one dirty module (one exotic file) while retaining 90 modules (0.07 seconds)
>> fourPlus 3
⇒ res0 : Int = 7
>> fourPlusInt 3
⇒ res1 : Int = 7
```

Well, how did you do?

I'll spoil matters here and say that while `composeInt` is easy to screw up, `compose` is *unbreakable*.[10] You used type variables instead of `Int` in `compose`'s type, so Ermine refused to let us inject arbitrary addition operations.

In other words, to use `(+)`, you must *prove* that you are working with `Int`. By making our type polymorphic, you have explicitly said "these types could be anything", so you do not have the necessary proof at this point to do addition.

You can't even put `f` and `g` in the wrong order in `compose`. In other words: as your definitions become more general and polymorphic, Ermine will discover more and more possible mistakes in them, and guide you away from them with the type checker.

The upshot is that, even though `fourPlusInt` and `fourPlus` do the same thing, and have the same type `Int -> Int`, `fourPlus` is more reliable, because you used more general components to put it together, even though it is not more general itself.

Moreover, a polymorphic type tells you more about the definition or expression it is talking about; with a little practice, you will see types like `compose`'s type and automatically know what the code **must** be doing, without even looking at the definition!

[10] Advanced users: I'm ignoring strictness and bottoms because we aren't *nearly* that far yet.

4 Working with Data

`Int`, `String`, and functions are all well and good. What if your data is more complicated, though? Working with `Int` and `String` is important, but there's more to *data modeling* in Ermine. Ermine gives you the tools to take simple data types, like the ones we've named, and build more complex types out of them, each representing different possible arrangements of data.

In this chapter, you'll define some simple data types, and discover how to build highly general functions associated with them.

4.1 Two things

We can start with a very simple data type. Create a new file, `ChapterData.e`, and write into it:

```
module ChapterData where

data PairII = PairIIOf Int Int
```

Now load it into the REPL and see what you have.

```
>> :load path/to/ChapterData.e
⊣ Importing module 'ChapterData' (0.02 seconds)
>> :t PairIIOf
⇒ Int -> Int -> PairII
```

All of your previous definitions have defined `Int`s, `String`s, and functions. They are called *term definitions*. This one, `PairII`, is a *data definition*, and Ermine can tell that because you started it with the magic word `data`.

It defines *two* things: the type `PairII`, which is just as much a type as `Int` or `String`, and the value `PairIIOf`, whose type you can see above.[1]

`PairIIOf` takes two `Int`s and yields a `PairII`; in other words, that's where we can actually get values of this type.

```
>> PairIIOf 42 ((+) 1 1)
⇒ res0 : PairII = (PairIIOf 42 2)
```

`PairIIOf` takes data as arguments and produces a piece of data of our new type, so we call it a *data constructor*.

4.2 Taking data out

Of course, you have to get the `Int`s out of the pair if you want to work on them again.

You can take them out directly with *pattern matching*. Add this import and definition to `ChapterData.e`.

```
import Num

addProducts (PairIIOf w x) (PairIIOf y z) = ((*) w x) + ((*) y z)
```

[1] Both are required to start with uppercase letters. Not coincidentally, term definition names are required to start with lowercase letters.

Where simple argument names would appear, as they first did in Section 3.7 [Defining functions], page 7, you have things in parentheses that look sort of like the RHS of `PairII` as you defined above. This matches up with the way you *called* `PairIIOf` to create the argument being given to `addProducts`.

```
>> :r
⊣ Reloaded...
>> :t addProducts
⇒ PairII -> PairII -> Int
>> addProducts (PairIIOf 2 3) (PairIIOf 4 5)
⇒ res0 : Int = 26
```

This is a somewhat inflexible way of working with your data types, though. Typically, when writing an assortment of functions for working with your data types, you'll write only a few functions that actually use destructuring, called *primitives*, and then write your other functions in terms of those simpler functions.

The simplest primitive is a function called the *catamorphism*, typically named the same as the type but starting with a lowercase letter. For data types unlikely to change structure, it can be very useful for any code working with that data type. Define a catamorphism for `PairII` like this in `ChapterData.e`.

```
pairII f (PairIIOf x y) = f x y

addProductsCata x y = (pairII (*) x) + (pairII (*) y)
```

```
>> :r
⊣ Reloaded...
>> :t pairII
⇒ forall a. (Int -> Int -> a) -> PairII -> a
>> addProductsCata (PairIIOf 2 3) (PairIIOf 4 5)
⇒ res0 : Int = 26
```

Based on this, what do you think the type of `(*)` is? What does the type variable `a` get set to in the calls to `pairII` made by `addProductsCata`?

4.3 Two other things

What if you want pairs of things that aren't `Int`s? You could try extending `ChapterData.e` like so.

```
data PairSS = PairSSOf String String
data PairIS = PairISOf Int String
data PairISSS = PairISSSOf PairIS PairSS
```

Now you have pairs of strings, a pair of an int and a string, and a pair of an int/string pair and a string/string pair. Your catamorphism and other functions for `PairII` won't work for any of these, so you have to define all those again, too.

But wait, just as we have polymorphic functions, we also have *polymorphic data types*. In fact, a polymorphic function is just one of many possible polymorphic data types. Add these definitions to `ChapterData.e`.

```
data Pair a b = PairOf a b
```

```
pair f (PairOf x y) = f x y
```

And, in the Ermine console:

```
>> :r
 ⊣ Reloaded...
>> PairOf 3 6
⇒ res0 : Pair Int Int = (PairOf 3 6)
>> PairOf 5 "hello"
⇒ res1 : Pair Int String = (PairOf 5 "hello")
>> :t PairOf 3
⇒ forall b. b -> Pair Int b
>> :t pair
⇒ forall a b a1. (a -> b -> a1) -> Pair a b -> a1
```

Just as the type variables in `compose` were set to types as you fixed them with arguments (see Section 3.10 [Reading generic types], page 9), so the type variables in `Pair` are set as you give data of particular types to the `PairOf` data constructor.[2]

TODO: *Likewise, when you use* `pair` *and fix* `a`*,* `b`*, and* `c`*...*

Try rewriting `addProducts` for two arguments of type `Pair Int Int`, once with pattern matching, once with the `pair` catamorphism.

4.4 A Practical Formula for Generality

In Section 3.6 [Generalization], page 6, we mentioned that no one knows all possible techniques of generalizing programs, even the experts. It will get easier with practice, but when beginning to program, it can be hard to spot even simple, well-known places where you can generalize.

You don't have to get it perfectly right the first time you write a program. All you have to do is be on the lookout for common patterns.

When you defined the variants of `Pair`, they all were written very similarly, with similar types of data constructors and catamorphisms. Here are the data constructor types:

```
PairIIOf   : Int    -> Int    -> PairII
PairSSOf   : String -> String -> PairSS
PairISOf   : Int    -> String -> PairIS
PairISSSOf : PairIS -> PairSS -> PairISSS
```

Not only do the catamorphisms also look very similar to each other, their definitions are almost exactly the same.

When this happens to your data types, you can often generalize by a simple experimental formula.

1. Add one or more type parameters to the left of the `=` in the data definition, as `a` and `b` were added to make `Pair`.
2. Use those type parameters on the RHS in place of specific types like `Int`. The parameters represent the differences between the different variants of `Pair`, in this example, the separate types of the two pieces of data that go in the pair.

[2] Ermine provides `Pair`, but it is typically spoken out loud as "two-tuple" and written as `(,)` in code. Its catamorphism is, unusually, called `uncurry`.

3. Go to where you are using the type in your signatures, and fill in the new parameters. For example, you may replace `PairII` with `Pair Int Int`, or `Pair a b` for sufficiently generic code.

Doing this in your code is called *parameterization*.

As you fix up your functions for working on data types, you'll notice they get more polymorphic too; `pair` picked up two type variables when you made the data definition polymorphic.

4.5 Parameterizing functions

This basic formula for parameterization works even better on your functions than it does on data types. It's very easy to write your functions for one specific case, and then later realize that you've written multiple functions with almost the same contents.

So you can follow a very similar experimental formula.

1. Add one or more parameters to the left of the `=` in the term definition.
2. Use those parameters on the RHS in place of specific expressions like `42`. The parameters represent the differences between the different variants of the function.
3. Go to where you are calling this function, and fill in the new parameters.

One of the most exciting things about Ermine is how its relatively rare features[3] combine to make this technique work in almost all code.

4.6 Holes and more pair functions

Define these functions in `ChapterData.e`.

```
mapFirst : (a -> c) -> Pair a b -> Pair c b
mapFirst f (PairOf a b) = _

mapSecond : (b -> c) -> Pair a b -> Pair a c
mapSecond f (PairOf a b) = _

bimapPair : (a -> c) -> (b -> d) -> Pair a b -> Pair c d
bimapPair f g (PairOf a b) = _
```

The `_` at the end of each term definition is a *hole*. As you're writing Ermine code, you'll realize that you need this or that particular expression, but don't feel like writing it yet. Or maybe you're not sure what type of expression you have to supply in a particular spot. See Section 6.5 [Using holes while solving errors], page 35, later on, will delve into an example of using holes in development.

Can you guess what the type of each hole above is? See if you're right:

```
>> :r

⊣ Remembered terms:

ChapterData.e:22:27: (Pair !c !b) inferred from
```

[3] Like nonstrictness and referential transparency.

```
mapFirst f (PairOf a b) = _
                          ^

ChapterData.e:25:28: (Pair !a !c) inferred from
mapSecond f (PairOf a b) = _
                           ^

ChapterData.e:28:30: (Pair !c !d) inferred from
bimapPair f g (PairOf a b) = _
                             ^

Reloaded...
```

This serves not only as a helpful note about what type you need, but also that you haven't finished this part of your program. The number after the filename tells you what line number you can find the hole on.

Now you can refine a little bit. Let's start with `bimapPair`. You know you need a `Pair`, and the argument `Pair a b` can't be used, because it's the wrong type. So you definitely need a new one, made with `PairOf`. Replace the definition in `ChapterData.e`:

```
bimapPair f g (PairOf a b) = PairOf _ _
```

I'll leave off the other `map` functions and just show the new holes.

```
>> :r

⊣ Remembered terms:

ChapterData.e:28:37: !c inferred from
bimapPair f g (PairOf a b) = PairOf _ _
                                    ^

ChapterData.e:28:39: !d inferred from
bimapPair f g (PairOf a b) = PairOf _ _
                                      ^

Reloaded...
```

Now, can you fill in the hole on the left? Here's a hint: you have, at hand, a thing of type `a -> b`, and also a thing of type `a`.

What goes in the rest of the holes? Fill these in, so you have no more "remembered terms" notes, before continuing. When experimenting, be assured that once Ermine accepts your changes here with no errors, you've given the correct definitions.

4.7 Possibilities

We've seen how to combine two pieces of data into one kind of value that you can get if you have both an `a` *and* a `b`. What if you have an `a` *or* a `b`?

Define these types and functions with holes in `ChapterData.e`.

```
data Which a b = This a | That b
```

Let's see what Ermine says about this.

```
>> :r
⊣ Reloaded...
>> :t This
⇒ forall a b. a -> Which a b
```

```
>> :t That
⇒ forall b a. b -> Which a b
>> This 42
⇒ res0 : forall b. Which Int b = (This 42)
>> That "greetings"
⇒ res1 : forall a. Which a String = (That "greetings")
```

By using the `|`, you've provided two data constructors, instead of one.[4] You can have many, just as you can have more than one or two arguments per data constructor.

4.8 Value polymorphism

We have also, for the first time, encountered *polymorphic values* that aren't functions. A `Which Int b` can be used anywhere a `Which Int String`, `Which Int (Pair String String)`, and so on might be called for.

For a data type, you can tell which data constructors will produce polymorphic values by looking at which type variables are used. `This` doesn't use `b`, so it is polymorphic in `b`. `That` doesn't use `a`, so it is polymorphic in `a`.

It is just like the function case you first encountered in Section 3.10 [Reading generic types], page 9. Polymorphic functions are just another kind of polymorphic value! You can tell that a function is going to be polymorphic over some bit of data if it doesn't do anything with the data that would require it to know what type it is. `compose` is polymorphic because all it does is the third argument to a function *you* provide, then hand the result of that to another function *you* provide, then give that back, without ever looking at any of those things itself.

4.9 Multiple definitions

Add these definitions to `ChapterData.e`.

```
which : (a -> c) -> (b -> c) -> Which a b -> c
which this that (This a) = _
which this that (That b) = _
```

Curiously, you seem to have two definitions of `which`.

Since encountering pattern matching in Section 4.2 [Taking data out], page 12, it's just been a way to take data out after it was given to a data constructor like `PairOf`. But unlike your previous data definitions, `Which` has two.

If I have, in hand, a value of type `Which Int String`, there are two possibilities at the outset: it could be a `This`, containing an `Int`, or it could be a `That`, containing a `String`. Ermine doesn't know in advance, and nor do you; if you knew in advance, you would just use `Int` or `String` as required by your circumstances.

Therefore, `which` needs to deal with both circumstances, and here's where the "matching" part of pattern matching comes in. When simplifying calls to `which`, once a value is

[4] Ermine provides instead `data Either a b = Left a | Right b`, and the same catamorphism, called `either`.

in hand, Ermine will look at each definition, starting with the first one you list. The first one that uses the same data constructor as the value will be used.[5]

Furthermore, just as you have two data constructors, your catamorphism needs to take two functions as arguments, one to handle the data in `This`, the other in `That`.

Let's see this in action.

```
>> :r

⊣ Remembered terms:

ChapterData.e:33:28: !c inferred from
which this that (This a) = _
                           ^

ChapterData.e:34:28: !c inferred from
which this that (That b) = _
                           ^

Reloaded...
```

Go ahead and fill in the holes. Ermine won't let you give the wrong answer here.

```
>> :r
⊣ Reloaded...
>> which ((+) 3) ((*) 5) (This 40)
⇒ res0 : Int = 43
>> which ((+) 3) ((*) 5) (That 100)
⇒ res1 : Int = 500
```

Let's get a little more practice with `which`. Add this definition to `ChapterData.e`, reload, and fill in the holes. Don't worry, this is another example where Ermine won't let you give the wrong answer.

```
bimapWhich : (a -> c) -> (b -> d) -> Which a b -> Which c d
bimapWhich f g = which (_ This f) (_ That g)
```

```
>> :r

⊣ Remembered terms:

ChapterData.e:37:25: ((a -> Which a b) ->
(!a1 -> !c) -> !a1 -> Which !c !d) inferred from
bimapWhich f g = which (_ This f) (_ That g)
                       ^

ChapterData.e:37:36: ((b -> Which a b) ->
(!b1 -> !d) -> !b1 -> Which !c !d) inferred from
bimapWhich f g = which (_ This f) (_ That g)
                                  ^

Reloaded...
```

[5] I won't elaborate further here about the vagaries of pattern matching, because they aren't important yet. If you are looking to jump ahead, it's pretty much *exactly* like Haskell's pattern matching: the same strictness model, irrefutable patterns, (non-pattern) guards, and wildcards are all present with the same syntax.

Here is a hint: there is a polymorphic function that you have already defined that will work for both of these holes.

Before moving on, it's worth your while to compare the types of `bimapWhich` and `bimapPair` (see Section 4.6 [Holes and more pair functions], page 15).

4.10 A simpler `Which`

For whatever reason, you are more likely to need to say in your programs something like "I might have a `String` here, but maybe not" than things like "either I have an `Int` or a `String`". So you typically use a simpler data type to handle this circumstance.

```
data Umm a = Nada | GotIt a

umm : b -> (a -> b) -> Umm a -> b
umm nada gotit Nada = _
umm nada gotit (GotIt a) = _

mapUmm : (a -> b) -> Umm a -> Umm b
mapUmm f = _
```

Ermine provides this one for you, as with all the other data types you've defined so far, though it is called `Maybe`, with `Nothing` and `Just` as the data constructors, and `maybe` as the catamorphism.

This one's a little trickier than it looks, though.

```
>> :r

⊣ Remembered terms:

ChapterData.e:42:23: !b inferred from
umm nada gotit Nada = _
                      ^

ChapterData.e:43:28: !b inferred from
umm nada gotit (GotIt a) = _
                           ^

ChapterData.e:46:12: (Umm !a ->
Umm !b) inferred from
mapUmm f = _
           ^

Reloaded...
>> Nada
⇒ res0 : forall a. Umm a = Nada
>> GotIt 3
⇒ res1 : Umm Int = (GotIt 3)
```

`Nada` results in a polymorphic `Umm` value, so the one `Nada` works everywhere you need an `Umm` of whatever. This is a blessing and a curse. By simplifying the fact of "I don't have one of those", it's easy to *accidentally* say "I don't have one of those" when you *really* meant to say that you do have one. This makes it possible to write wrong things for the second and third holes above.

Even though you aren't protected from all wrong ideas here, you're protected from *most* of them. So you can formulate a simple *test*, where you can try out a few things in the REPL on your definition to see if it's correct.[6]

Fill in the holes for `umm`, and then try it out on data of both cases. Make sure you get the same answers for both calls below.

```
>> :r
 ⊣ Reloaded...
>> umm Nada GotIt (Nada)
⇒ res0 : forall a. Umm a = Nada
>> umm Nada GotIt (GotIt 33)
⇒ res1 : Umm Int = (GotIt 33)
```

Even though it is possible to mess up `umm` without Ermine's type checker catching the problem, it still catches most problems, so this simple test, which is just "you get the same thing out of `umm Nada GotIt` that you put into it", catches what problems can happen. This kind of rule, which always looks something like "you always get out what you put into it", is called an *identity*.

Next, fill in the hole for `mapUmm`. You may expand it into two pattern matches, but for a better single-line definition refer back to `bimapWhich` for inspiration. Test `mapUmm`'s identity, `mapUmm id`, too, as follows.

```
>> :r
 ⊣ Reloaded...
>> mapUmm id (Nada)
⇒ res0 : forall a. Umm a = Nada
>> mapUmm id (GotIt 33)
⇒ res1 : Umm Int = (GotIt 33)
```

Section 3.11 [Safety in polymorphism], page 10, introduced the idea of increased polymorphism making us safer. Part of that is that the more polymorphic your function is, the less testing you have to do to be assured that it is correct. "I don't have to test at all" is just one example of that.

4.11 More things than you can count

Here are some types that represent two, three, and four `Int`s respectively.

```
Pair Int Int
Pair Int (Pair Int Int)
Pair Int (Pair Int (Pair Int Int))
```

This sort of thing is useful if you have such an exact number of things to work with. What if you might have zero, or five, or 42, or infinitely many of them, and you don't know yet?

Add these definitions to `ChapterData.e`.[7]

[6] The Haskellian might wonder about QuickCheck-like possibilities here. It won't happen for Ermine Legacy, for which a necessary feature for sanity, typeclass dictionaries, will remain absent. This should be possible with the new Ermine runtime universe, though.

[7] `Stream` is provided by Ermine in the `List.Stream` module.

```
data Stream a = SCons a (Stream a)

show4 (SCons a (SCons b (SCons c (SCons d ignored)))) = (a, b, c, d)

fortyTwos = SCons 42 fortyTwos

mapS : (a -> b) -> Stream a -> Stream b
mapS f (SCons a sa) = _

countFrom : Int -> Stream Int
countFrom n = SCons n (countFrom _)
```

The only data constructor for `Stream a`, `SCons`, requires another `Stream a`. Where will it come from?[8]

```
>> :r

⊣ Remembered terms:

ChapterData.e:56:23: (Stream !b) inferred from
mapS f (SCons a sa) = _
                      ^

ChapterData.e:59:34: Int inferred from
countFrom n = SCons n (countFrom _)
                                 ^

Reloaded...
>> :t fortyTwos
⇒ Stream Int
>> show4 fortyTwos
⇒ res0 : (Int, Int, Int, Int) = (42,42,42,42)
```

Here you've used the *4-tuple* type `(,,,)`, which lets you put the types in between the points there and acts just like `Pair` but for four elements instead of two. Also available are 2-tuples `(,)`, which you can use instead of `Pair`, 3-tuples, and so on up to some larger number.[9] `show4` extracts the first four elements of the *infinite* stream.

This is saying "the infinite stream of 42s is a single 42 followed by the infinite stream of 42s." Logically, this is the minimum you can say about the stream and still expect to get something. What if you left off the 42 at the beginning?

```
fortyTwos = fortyTwos
```

This is saying "the infinite stream of 42s is the infinite stream of 42s." This is meaningless.

```
>> :r
⊣ Reloaded...
```

[8] Programmers outside the Haskell tradition should note here that `fortyTwos` is *not* "calling itself" in this example. Its definition merely includes itself.

[9] The point of the large number, which might be 14 for all I know, is that if your tuple is this large, you need to stop using a tuple and define a custom data type. Also, attention Pythonistas: tuples are *not* glorified lists.

```
>> :t fortyTwos
⇒ forall a. a
```

The inferred type of `forall a. a` means this value can never be computed. After all, there is no single value of all types.

You could say `SCons 42 (SCons 42 fortyTwos)`, which would also work. Then, however, you're saying "the infinite stream of 42s is two 42s followed by the infinite stream of 42s". There's no benefit to saying more than you must, here, though.

What about "the infinite stream of 42s is the infinite stream of 42s followed by 42"? The problem with this is that once you have an `SCons` in hand, it has yet another `SCons` representing the rest of the stream, and so on. You can never get to the end; it's supposed to be infinite, after all.

Fill in the hole in `countFrom`'s definition so you get this result.

```
>> :r
⊣ Reloaded...
>> show4 (countFrom 5)
⇒ res0 : (Int, Int, Int, Int) = (5,6,7,8)
```

Next, fill in the hole in `mapS` so you get these results.

```
>> :r
⊣ Reloaded...
>> show4 (mapS id (countFrom 7))
⇒ res0 : (Int, Int, Int, Int) = (7,8,9,10)
>> show4 (mapS id fortyTwos)
⇒ res1 : (Int, Int, Int, Int) = (42,42,42,42)
```

4.12 Into the abyss

This very useful function on `Stream` is worth musing on.

```
foldrS : (a -> b -> b) -> Stream a -> b
foldrS f (SCons a sa) = f a (foldrS f sa)
```

First, it looks very much like a catamorphism. For *recursive data types* that refer to themselves, like `Stream`, you will commonly define a fold that looks like this, instead of a catamorphism that simply reproduces the data constructor's argument types. Like the data type, the function is also *recursive*; it uses itself as part of its own definition.

It's very much like your catamorphisms in that supplying the data constructors as arguments will form an identity. Here are the catamorphism-derived and `foldrS`-derived identities you can reach so far; you've already seen `umm`.

```
>> pair PairOf (PairOf 3 2)
⇒ res0 : Pair Int Int = (PairOf 3 2)
>> which This That (This 5)
⇒ res1 : forall b. Which Int b = (This 5)
>> which This That (That 7)
⇒ res2 : forall a. Which a Int = (That 7)
>> umm Nada GotIt (Nada)
⇒ res3 : forall a. Umm a = Nada
>> umm Nada GotIt (GotIt 11)
```

```
⇒ res4 : Umm Int = (GotIt 11)
>> show4 (mapS id (countFrom 13))
⇒ res5 : (Int, Int, Int, Int) = (13,14,15,16)
>> show4 (foldrS SCons (countFrom 17))
⇒ res6 : (Int, Int, Int, Int) = (17,18,19,20)
```

This is always a good test to include for your data structures: can you define a function over your data type which, when supplied with the data constructors for your data type, forms an identity?

But there's something else going on here. Check out the signature for `foldrS` again.

```
foldrS : (a -> b -> b) -> Stream a -> b
```

Where does the `b` come from that we give to the first argument? Well, it comes from the fold over the second, third, and so on elements of the stream. What about that fold, where does it get `b`? From the fold over the third, fourth, and so on elements of the stream.

Imagine that you are standing on a field. You are not on earth, or even in this universe, because this land is flat, and extends forever in every direction. You stand upon a very straight boulevard. You look ahead on the road, and it extends forever, straight in the direction you look.

No matter how good your eyesight, you cannot help but perceive the road as narrowing to a vanishing point at some long, but not infinite distance away. That does not mean the road is not infinite, merely that you have not *realized* the infinite extent of it. *That* is where the `b` comes from.

This is the sort of thing Ermine is supposed to make sane.

4.13 Many things

So you can represent three things, four things, and an infinite number of the same things. What about an *unknown* number? Could be none, could be 1,024. Could even be an infinite number again.[10]

`Stream` is just one of a variety of interesting *recursive data types*, by which we mean data types that refer to themselves. Here is another one, a slight variant of `Stream`, to add to `ChapterData.e`.[11]

```
data Things a = XNil | XCons a (Things a)

foldrThings : (a -> b -> b) -> b -> Things a -> b
foldrThings f n XNil = n
foldrThings f n (XCons a la) = f a (foldrThings f n la)

mapThings : (a -> b) -> Things a -> Things b
mapThings f = foldrThings _ _
```

[10] We will look at structures that can't be infinitely large later. This isn't as useful a constraint on your data as it sounds, though.

[11] `Things`, `XNil`, and `XCons` are provided by Ermine as `List`, `Nil`, and `(::)`. Notably for Haskell users, `[]` is **not** a valid replacement for `List`, `[Int]` is **not** a valid way to say `List Int`, and `:` and `::` have swapped places in the language.

Pretty much everything in this chapter is here: it's a generic type, with multiple data constructors, multiple values in one of them, and is recursive like `Stream`. We also have the usual fold and map functions. Like `foldrS`, `foldrThings` is recursive, and the structure of the fold is modeled closely after the data constructors.

```
>> :r

⊣ Remembered terms:

ChapterData.e:71:27: (!a ->
Things !b -> Things !b) inferred from
mapThings f = foldrThings _ _
                          ^

ChapterData.e:71:29: (Things !b) inferred from
mapThings f = foldrThings _ _
                            ^

Reloaded...
>> XCons 1 (XCons 5 XNil)
⇒ res0 : Things Int = (XCons 1 (XCons 5 XNil))
>> XCons 42 (XCons 1 (XCons 5 XNil))
⇒ res1 : Things Int = (XCons 42 (XCons 1 (XCons 5 XNil)))
>> XNil
⇒ res2 : forall a. Things a = XNil
```

By convention, we consider, in the value `XCons 42 (XCons 1 (XCons 5 XNil))`, the `42` to be the "first" element. But who is to say the `XNil` truly marks the end, rather than the beginning, of the list? It is ultimately up to you, and what suits your program best.

The unique properties of Ermine, inherited from Haskell, make this structure much more useful than it might seem to be. So you will encounter it, and use it, **all the time**.

Fill in the holes in `mapThings`'s definition, so it satisfies the identity `mapThings id` with these tests.

```
>> :r
⊣ Reloaded...
>> mapThings id (XCons 1 (XCons 5 XNil))
⇒ res0 : Things Int = (XCons 1 (XCons 5 XNil))
>> mapThings id XNil
⇒ res1 : forall a. Things a = XNil
```

4.14 On `List`'s `foldr`

TODO: *I'm not sure I like this section. It seems audience-inappropriate. If I delete it, comments have to be introduced eventually.*

If you have used other programming languages, you may be surprised by this interesting property of Ermine's simplification model, first seen in Section 3.7 [Defining functions], page 7.

```
constU : a -> b -> Umm a
constU a b = GotIt a
```

```
-- You may want to write down what each function
-- is for in a little human-language comment
-- above it, with -- in front, like this:

-- | First list element if GotIt, Nada otherwise.
headThing : Things a -> Umm a
headThing = foldrThings constU Nada
```

The property is this: **this is a perfectly reasonable way to write this function**. It is optimal even for infinitely long lists.

TODO: *If I keep this section, talk about* `constU`.

4.15 Many different things

Now, you might say, "what if I want a list that can contain both `Int`s and `String`s? I only have the one type variable!" That is no problem. It's the type `Things (Which Int String)`. Just as you can combine term definitions to form more complex programs, you can combine generic types to form types that precisely describe the structure of your data.

Let's use left-to-right =-substitution to see what that means.

```
Things (Which Int String)
data ThingsWIS = XNil | XCons (Which Int String) ThingsWIS
data ThingsWIS = XNil | XCons (This Int | That String) ThingsWIS
```

And we can distribute the `|` in parentheses,

```
data ThingsWIS = XNil2
               | XConsThis Int ThingsWIS
               | XConsThat String ThingsWIS
```

If you have an `Things Int` in hand, it's easy enough to use `mapThings This` to get to a list of the right type, `forall b. Things (Which Int b)`.

This might be too restrictive, though. What about a list that goes, `Int`, `String`, `Int`, `String`, and so on? Also easy. That's `Things (Pair Int String)`. It's no different, in terms of data, to have two things at a time in the `Pair` than directly in the list. Let's substitute!

```
Things (Pair Int String)
data ThingsPIS = XNil3 | XCons3 (Pair Int String) ThingsPIS
```

And the single data constructor of `Pair`, `PairOf`, can explode into the containing data constructor.

```
data ThingsPIS2 = XNil4 | XCons4 Int String ThingsPIS2
```

The key to beautiful data design is *seeing* the combinations you should use. We will explore that soon, after an excursion.

5 Stretch: Universals and Existentials

This chapter is aimed at readers already familiar with Haskell or advanced type systems, wanting to delve deeper into Ermine. You may try it now, but feel free to come back to it later once you have more of the basics down.

You've seen how Ermine's type-printing format uses `forall` to talk about the various type variables. It was only later in Ermine's development that it became *possible* to have the type variable scope inferred in an explicit polymorphic type; that's why, if you browse through the standard library, you'll see most functions declare the type variables with `forall` explicitly.

In this chapter, we'll just jump into the polymorphism-related features. I'm not going to attempt a full introduction here; that would require a whole other book!

5.1 Haskell extensions

Those Haskell extensions implemented in Ermine mostly have to do with type system polymorphism. Of type-system-oriented Haskell extensions, Ermine implements:

EmptyDataDecls
LiberalTypeSynonyms
TypeOperators
: More or less exactly as in Haskell, for exactly the same reason.

ExistentialQuantification
: TODO: *I don't yet know whether these will capture typeclass dictionaries.*

KindSignatures
PolyKinds
: More important in Ermine, for which the built-in atomic kind `row` appears in many programs.

ImpredicativeTypes
RankNTypes
: Both universal and existential quantification encode in the same way in data constructors. Working with higher-ranked and impredicative term definition types is quite different, though.

ScopedTypeVariables
: As stated, not just an extension, but a historical necessity.

5.2 Limits and alleviations in Legacy

I don't know when you'll be able to write data constructors that tote typeclass dictionaries around in Ermine. It seems likely that you'll continue to be unable to remember satisfaction of row constraints.

This sample has not yet been tested; it may contain undiscovered errors or inaccuracies in expected output.

```
>> data Joinable rel t =
|>   forall r s. (RUnion2 t r s) => Joinable (rel r) (rel s)
```

error TODO: *blah blah higher-ranked row constraints*

As for typeclass dictionaries, they don't matter, because we can hand those around like normal values anyway. Here's how `Functor` is defined in `Control.Functor`.[1]

This sample has not yet been tested; it may contain undiscovered errors or inaccuracies in expected output.

```
module ChapterUniv where

data Functor f = Functor (forall f. (a -> b) -> f a -> f b)
```

And `Monad`.

This sample has not yet been tested; it may contain undiscovered errors or inaccuracies in expected output.

```
data Monad m = Monad (forall a. a -> m a)
                     (forall a b. m a -> (a -> m b) -> m b)
```

This sample has not yet been tested; it may contain undiscovered errors or inaccuracies in expected output.

```
>> :load "path/to/ChapterUniv.e"
⊣ Reloading...
>> :k Functor
⇒ (* -> *) -> *
>> :t Functor
⇒ forall f. (forall a b. (a -> b) -> f a -> f b) -> Functor f
```

Extractor functions on these are provided, with the names `fmap`, `unit`, and `bind`, all taking these "dictionaries" as the first argument. When you pattern match on these, they retain the polymorphism you'd expect.

This sample has not yet been tested; it may contain undiscovered errors or inaccuracies in expected output.

```
>> unit2 (Monad u _) = (u, u)
>> :t unit2
⇒ forall m a b. Monad m -> (a -> m a, b -> m b)
```

5.3 Higher-ranked functions

In Haskell, it's easier to box up a universal in a new type, as done with the `Functor` and `Monad` constructors above. But in Ermine, it's even more important.

Let's say you try to write `mapply`, one of the `Functor` module combinators, by exploding the `Functor`.

This sample has not yet been tested; it may contain undiscovered errors or inaccuracies in expected output.

```
mapply : (forall a. (a -> b) -> f a -> f b)
      -> f (c -> d) -> c -> f d
mapply map ffuns c = map apc ffuns
```

[1] Not `Data.Functor`, as you might expect from Haskell.

```
  where apc f = f c
```

This sample has not yet been tested; it may contain undiscovered errors or inaccuracies in expected output.

```
>> :r
```

error TODO: *blah blah failed to unify*

This is because explicit types don't get along with inferred types in Ermine in the same way they do in Haskell. First, the term is typed; then, the inferred type is checked to make sure it's subsumed by the explicit type.

If you are used to working around the monomorphism restriction in Haskell by attaching explicit types, this ordering may surprise you. Keep in mind, though, that Ermine *has never had* a monomorphism restriction.

Instead, you have to write this, and all higher-ranked or impredicative types used in terms, as follows.

This sample has not yet been tested; it may contain undiscovered errors or inaccuracies in expected output.

```
mapply (map : some f. forall a b. (a -> b) -> f a -> f b)
       ffuns c = map apc ffuns
  where apc f = f c
```

Curiously, what you choose for the `f` variable name is completely arbitrary. It doesn't refer to an explicit outer variable scope; as I've implied, *there isn't one*. It's just that you have to declare free variables so the typer will know which things to unify when the polymorphic argument gets used in some context. It'll complain if you don't get this right.

We'll see another example, with a different and more exciting error message, in Section 5.5 [Church encoding], page 29.

5.4 Monoid checking with existentials

Let's make a quick test set for testing left identity of monoids.

This sample has not yet been tested; it may contain undiscovered errors or inaccuracies in expected output.

```
import Bool
import Control.Functor
import Control.Monoid
import List
import Num

data MonoidTest r =
  forall a. MonoidTest (Monoid a) (a -> a -> r) a

tests : List (MonoidTest Bool)
tests = [MonoidTest (Monoid 0 (+))   (==) 42,
         MonoidTest (Monoid [] (++)) (==) [1,2,3]]

check : MonoidTest r -> r
```

```
check (MonoidTest m eq v) = eq v (mappend m (mempty m) v)

checkAll : List Bool
checkAll = fmap listFunctor check tests
```

This sample has not yet been tested; it may contain undiscovered errors or inaccuracies in expected output.

```
>> :r
⊣ Reloading...
>> :t MonoidTest
⇒ forall a r. Monoid a -> (a -> a -> r) -> a -> MonoidTest r
>> checkAll
⇒ res0 : List Bool = [True, True]
```

Ermine does have an `exists` keyword, but you can't use it to make impredicative existential types; you must box them up as you would in Haskell via higher-ranked `forall`.

5.5 Church encoding

Something curious happens when you try to write the catamorphism for `MonoidTest`.

This sample has not yet been tested; it may contain undiscovered errors or inaccuracies in expected output.

```
monoidTest f (MonoidTest m eq v) = f m eq v
```

This sample has not yet been tested; it may contain undiscovered errors or inaccuracies in expected output.

```
>> :r
error  TODO: blah blah escaped skolems
```

TODO: *index escaped skolem error message*

This is the equivalent of "rigid type variable" errors you see when trying to do the same thing in Haskell. The solution is the same; types with boxed existentials give rise to higher-ranked catamorphisms.

This sample has not yet been tested; it may contain undiscovered errors or inaccuracies in expected output.

```
monoidTest (f: some r z. forall a.
               Monoid a -> (a -> a -> r) -> a -> z)
           (MonoidTest m eq v) = f m eq v
```

Of course, it's also possible to write `MonoidTest` without existentials at all, via boxed universals like `Functor` has.[2]

This sample has not yet been tested; it may contain undiscovered errors or inaccuracies in expected output.

```
import Function                 -- for flip
```

[2] If you're a newcomer trying to forge ahead through this chapter, you might notice here that catamorphisms have an even deeper relationship with their data types than we've seen so far.

```
data MonoidTest2 r =
  MonoidTest2 (forall z. (forall a. Monoid a -> (a -> a -> r)
                                    -> a -> z)
                      -> z)

oneToTwo : MonoidTest r -> MonoidTest2 r
oneToTwo mt = MonoidTest2 (flip monoidTest mt)

twoToOne : MonoidTest2 r -> MonoidTest r
twoToOne (MonoidTest2 f) = f MonoidTest
```

This sample has not yet been tested; it may contain undiscovered errors or inaccuracies in expected output.

```
>> :r
⊣ Reloading...
>> :t MonoidTest2
⇒ forall r.
    (forall z.
      (forall a. Monoid a -> (a -> a -> r) -> a -> z) -> z)
    -> MonoidTest2 r
```

As an exercise, it's worth seeing what happens when you try to rewrite `oneToTwo` using `compose`, which is, incidentally, defined in the `Function` module you just imported as `(.)`.

6 Understanding type errors

So far, you have written several generalized functions where either it was impossible to screw up and load the function into Ermine, or a simple identity test, in combination with the type system, was sufficient to ensure you had gotten it right.

You will never stop seeing type errors as long as you write code in Ermine. Experienced Ermine programmers use type errors as a guide when they're making changes: they make a change in one place, reload, and then take the resulting type errors as hints of what other places in the code need to change.

That is all they are: hints. Ermine cannot solve type errors on its own; it cannot even reliably tell you *where* the underlying change should be made. With some practice, you'll start to recognize the probable causes of type errors when they happen, and by comparing the type error with the actual code, be able to fix what *really* needs to be fixed.

Ill-typed programs are like the noise drowning out the signal; they vastly outnumber well-typed programs, which are islands of sanity in this ocean of chaos. So it is impractical to discuss all possible ill-typed situations in which you could find yourself as you explore Ermine, and we will not attempt to do so. Consider that these are a few hints and shortcuts that might prove useful in your journey.

6.1 Too much help

The Scala programming language's typer attempts to be more "helpful" than Ermine's typer, in some ways. There are many more categories of type errors in Scala, and it will sometimes suggest a fix to do.

For example, when you write certain pieces of code, like this, it will make the suggestion as shown.[1]

```
final case class Box[A](a: A)
def box[A <: B](a: A): Box[B] = {
  val b = Box(a)
  b
}
```

TODO: *covariant blah blah*

Except that the fix it suggests isn't really the fix. The fix is to define `box` as follows.

```
def box[B, A <: B](a: A): Box[B] =
  Box(a)
```

This is fine, if you have the confidence and the desire to disagree with the typer, which isn't asking for nothing, because after all, it's the *typer*, which should *know what it's talking about, right?*[2]

[1] This isn't a Scala book, so I'm only presenting a very trivial example.

[2] "Make it covariant" is asking a lot more of you than it seems to be implying. It's also, in effect, asking you to be okay with typer bugs such as SI-2066. There's much disagreement in the Scala community, but I've found myself in the "don't use variance" camp for a while, and later personal discoveries like SI-2066 have only reinforced that.

So what if you're not? The types in your program manifest as a set of logical premises and conclusions. If the typer merely reports where the logic fell apart, and in what way, then you only have one assumption: your premises are faulty. But the assumption of the "helpful" suggestion is that your premises are faulty *and* you are supposed to fix them *in a particular way*.

The typer cannot really know that, though. Overly helpful type errors encourage over-reliance on the advice they give, especially when it is so often wrong, ultimately making your programs less well-founded and general.

6.2 Building functions with type errors

Let's see how it feels to use type errors as a guide to building up a simple program. Create a file `ChapterErrors.e` with this code.[3]

```
module ChapterErrors where

import Bool
import Eq
import List

-- | Occupations, and some places you're likely to find their
-- practitioners.
data JobCluster = JobCluster String String

-- | Getters for 'JobCluster'.
occupation, location : JobCluster -> String
occupation (JobCluster j _) = j
location (JobCluster _ o) = o

-- | Some places you might find some people.
commonJobs : List JobCluster
commonJobs = [JobCluster "actor" "Hollywood",
              JobCluster "hacker" "Cambridge",
              JobCluster "financier" "London"]

-- | The first element in a list matching 'p'.
findElt p (x :: xs) = p x || findElt p xs
```

Load it up, and everything seems fine.[4]

```
>> :load path/to/ChapterErrors.e
⊣ Importing module 'ChapterErrors' (0.08 seconds)
>> map occupation commonJobs
⇒ res0 : List String = ["actor","hacker","financier"]
>> map location commonJobs
```

[3] I imported `Bool` because my imaginary super-helpful Ermine editor told me I should import it to get access to `(||)`. Serves me right.

[4] Haskellians will be surprised that no exhaustiveness warning popped up for `findElt`. Ermine Legacy doesn't have exhaustiveness warnings. TODO: *Will Ermine-hs?*

```
⇒ res1 : List String = ["Hollywood","Cambridge","London"]
```

TODO: *mapThings, Things, forward ref *_Bracket*

Now, add a function that searches your `commonJobs` "database" for the location associated with a particular occupation. In keeping with the principle of building up programs from more general components, it'll use the general-purpose `findElt` function on lists.

```
-- | Find a likely location for the given job.
oftenFoundAt job = location (findElt jobis commonJobs)
  where jobis j = job == occupation j
```

To which you see:

```
>> :r
error  ChapterErrors.e:14:1: error: failed to unify type JobCluster with type Bool
error  location (JobCluster _ o) = o
error  ^
```

How did an error end up in the perfectly sensible definition of `location`, just by adding code elsewhere? Well, it didn't; Ermine actually spotted the error when trying the new *call* to `location`. Legacy is a little quirky about type error locations. If you see an absurd error location like this, check the places where you're using the definition Ermine is pointing at.

6.3 Explicit types test sanity

TODO: *"explicit types" is not the right term here.*

But the error isn't *really* in `oftenFoundAt`. It's in `findElt`. Take a look at the documentation comment supplied: "The first element in a list matching `p`." What is its actual type?

```
>> :t findElt
⇒ forall a. (a -> Bool) -> List a -> Bool
```

That's clearly wrong. The first **element** is an `a`, not a `Bool`, so it should result in `a`, not `Bool`. But this function is *some* well-typed function, so Ermine just accepted it. It can't understand your English-language documentation to figure out that the type doesn't match what you meant.

As you work with Ermine more and more, you'll write more types first, before writing term definitions to go along with them. This is just Section 3.5 [Thinking in types], page 6, at work. Ermine doesn't require you to write most types at all, though; it can figure out everything on its own. If you give it a term definition that just happens to line up, even though it's not the one you want, it'll assume you know what you want, and only complain later when you try and *use* the term where you expected it to have a different type.

Moreover, the example here is simple! You could go many steps down the path of bad types, where a wrong type in some definition percolates through several other definitions, only to manifest much later. As you checked `findElt`'s type above, you need to check your definitions' types as you go. A seemingly beautiful program with a shaky, ill-typed foundation will eventually sink into the mire.

You don't have to keep checking your types' sanity, though; you only have to do it once, and write down what you expect as an explicit type as introduced in Section 3.10 [Reading generic types], page 9, so Ermine will check it for you.

I've used explicit types so far to test your definitions, because I know exactly what definition I'm pointing you towards, and what its type would be. You don't have to know exactly what you want to benefit from explicit-type-based testing. You can extend the simple formula, described in Section 4.5 [Parameterizing functions], page 15, to build explicit types.

1. Write a function definition, and load it into Ermine.
2. Ask Ermine for its type with `:t`.
3. If that type makes sense, copy it into your Ermine code as-is as an explicit type.
4. If the type doesn't make sense, copy it as before, make the necessary changes to the type, and update your function until it works. If you're not sure what changes need to happen to the type, skip this step.

So, grab that signature, change the last `Bool` to `a`, comment out the definition of `oftenFoundAt` with `{-` and `-}` as shown, and reload.

TODO: *Check whether needing to comment out oftenFoundAt is a Legacy quirk or also exists in Haskell. I imagine it comes from ordering as described in Section 5.3 [Higher-ranked functions], page 27.*

```
-- | The first element in a list matching 'p'.
findElt : forall a. (a -> Bool) -> List a -> a
findElt p (x :: xs) = p x || findElt p xs

{-
-- | Find a likely location for the given job.
oftenFoundAt job = location (findElt jobis commonJobs)
  where jobis j = job == occupation j
-}
```

```
>> :r
error ChapterErrors.e:24:1: error: failed to unify type !a with type Bool
error findElt p (x :: xs) = p x || findElt p xs
error ^
```

By introducing a type error in `findElt`, which compiled successfully, you've made progress, because you're closer to a *correct* definition, which matters more.

Closer, but not quite there: `findElt` doesn't handle the case where no matching element is found. You don't need to see the final definition to know that; the type is enough to tell you that. That is, it is not possible to write a function with the type `(a -> Bool) -> List a -> a` that handles the not-found case. You can fix that later, though; this is good enough for now.[5]

6.4 Choosing the right function

I mentioned that `findElt` didn't match up with the documentation. Since Ermine accepted it anyway, and gave it a type, is it meaningful? Sure. It's even in the standard library.

```
>> :t any
⇒ forall a. (a -> Bool) -> List a -> Bool
```

[5] The answer will *not* be "provide a default `a`", though.

Yes, sometimes it can be useful to be able to say "are there any entries in this list that match this `p`? You don't have to tell me which one, just whether there is one." Like all good general functions, you can build neat things with this one. Note: `(.)` means `compose`, so this code sample is like `compose any (compose any any)`.

```
>> :t any . any . any
⇒ forall a. (a -> Bool) -> List (List (List a)) -> Bool
```

Can you guess what that does, based on the type?

So, it's a reasonable function. *It's just not the one you want, right now.* Thinking in types can point us in the right direction, though. Browsing `List.e` in the Ermine standard library brings up a couple candidates that you might reuse to quickly build the `findElt` that you want.

```
>> :t filter
⇒ forall a. (a -> Bool) -> List a -> List a
>> :t find
⇒ forall a. (a -> Bool) -> List a -> Maybe a
```

You can use either of these, and one other function in either `List.e` or `Maybe.e`, to finish `findElt`. It should pass these two tests, besides typing.[6]

```
>> findElt ((==) 42 . fst) [(42,1),(2,2),(42,3)]
⇒ res2 : (Int, Int) = (42,1)
>> findElt ((==) 42 . fst) [(1,1),(42,2),(42,3)]
⇒ res3 : (Int, Int) = (42,2)
>> findElt ((==) 42 . fst) [(1,1)]
error  res4 : (Int, Int) =
error    <error: match error at...>
```

If you're having a little trouble, continue on for more tools to help.

6.5 Using holes while solving errors

If you're not sure what to do to resolve a type error, you can use the holes, first seen in Section 4.6 [Holes and more pair functions], page 15, to "fix" the error and continue with other type errors. This isn't a proper fix, but it can help to clarify matters.

Let's say you chose to use `filter` to write `findElt`, as mentioned in the previous section. Well, its signature is pretty much like `findElt`, so what happens when you just use it directly?

```
findElt p xs = filter p xs
```

```
>> :r
error  ChapterErrors.e:24:1: error: failed to unify type !a with type (List !a)
error  findElt p xs = filter p xs
error  ^
```

Well, the main problem seems to be the return type being different. You can shim that with a `_`.

```
findElt p xs = _ (filter p xs)
```

[6] Yes, we really want to use `find`'s type for `findElt`'s type, hence the error in the tests below.

```
>> :r

Remembered terms:

ChapterErrors.e:24:16: (List !a -> !a) inferred from
findElt p xs = _ (filter p xs)
               ^
```

This isn't a type error. You can keep going, reintroduce `oftenFoundAt` as commented out in Section 6.3 [Explicit types test sanity], page 33, and keep refining your program. Ermine will keep reporting the `_` location, as a reminder that you're not done, though. Because if you try to test this part of the program:

```
>> findElt ((==) 42 . fst) [(1,1),(42,2),(42,3)]
error runtime error: Hole(ChapterErrors.e(24:16))
error   (of class com.clarifi.reporting.ermine.Hole)
error Use ":stack" to see a stack trace
```

Find the right function with the shape of `List !a -> !a`, put it where the `_` is as in prior hole experiments, and you'll have a finished definition.[7]

You should find the right function, thereby solving the exercise in Section 6.4 [Choosing the right function], page 34, before proceeding.

6.6 Refine by adding type errors

If you have completed `findElt` and uncommented `oftenFoundAt`, you can try it out.

```
>> oftenFoundAt "hacker"
⇒ res6 : String = "Cambridge"
>> oftenFoundAt "financier"
⇒ res7 : String = "London"
```

It seems to work pretty well, for the examples in the `commonJobs` database.

```
>> oftenFoundAt "lawyer"
error res8 : String =
error   <error: match error at...>
```

Well, this isn't very helpful. This error is an *untyped* error called *bottom*; you can't use it in your program, detect it, or recover from it in any sane way. It essentially means that your program crashed.

In Section 6.3 [Explicit types test sanity], page 33, I mentioned that `findElt` doesn't handle the case where no matching element is found. By changing `findElt`'s type, to include a meaningful value when no matching element is found, your program becomes more "well-typed", as well as becoming more *total*, meaning that its functions have fewer opportunities to hit "bottom", producing meaningful results in more cases.

[7] You've seen `!` a few times in type errors and hole reports at this point. The type variables they indicate are not related to type variables you've chosen in explicit types; they're placeholders Ermine uses when checking types. You're seeing Ermine report things about what it sees when it hasn't finished running; these *skolems* never appear explicitly in complete, typed programs.

6.7 A better `findElt` type

The "didn't find a match" case of `findElt` can be elegantly represented with the `Umm` data type, which you encountered in Section 4.10 [A simpler `Which`], page 19. `Nada` represents the "didn't find a match" case, and `GotIt` represents the "found a match" case.

Ermine provides this for you, already, as the `Maybe` data type. The "got it" data constructor is `Just`, and the other is `Nothing`.

```
>> :t Nothing
⇒ forall a. Maybe a
>> :t Just
⇒ forall a. a -> Maybe a
```

The "maybe I have it, maybe I don't" you're looking for here is in the result type of `findElt`. So add it to `ChapterErrors.e`.

```
-- Add this to the imports at the top of the file.
import Maybe

-- Then, for findElt's type:
findElt : forall a. (a -> Bool) -> List a -> Maybe a
```

You updated the type, but didn't update the `findElt` term definition. What happens?

```
>> :r
error  ChapterErrors.e:15:1: error: failed to unify type JobCluster
error                                with type (Maybe JobCluster)
error  location (JobCluster _ o) = o
error  ^
```

Ah, more of this. Hide `oftenFoundAt` away again.

```
>> :r
error  ChapterErrors.e:25:1: error: failed to unify type (Maybe
error  !a) with type !a
error  findElt p xs = solution64 (filter p xs)
error  ^
```

That's better. I've hidden the solution for the hole shown earlier, but the principle is the same. Try introducing a hole, that would adapt that solution to `filter p xs`.

```
findElt p xs = _ solution64 (filter p xs)
```

That is, it takes the solution and the filter as its arguments.

```
>> :r

Remembered terms:

ChapterErrors.e:25:16: ((List a -> a) ->
List !a1 -> Maybe !a1) inferred from
findElt p xs = _ solution64 (filter p xs)
               ^
```

This can be read as "convert a `List a -> a` to a `List a -> Maybe a`." The problem is that `List a -> a` is nontotal by nature; we can't fix it this way. It's the wrong function.

So put the hole from the exercise back.

```
findElt p xs = _ (filter p xs)
```

Now the results are more promising.

```
>> :r

Remembered terms:

ChapterErrors.e:25:16: (List !a -> Maybe !a) inferred from
findElt p xs = _ (filter p xs)
               ^
```

There's a good function in `List.e` of this type that you can use. Here's your usual repertoire of tests, very closely related to the old ones.

```
>> :r
⊣ Reloaded one dirty module...
>> findElt ((==) 42 . fst) [(42,1),(2,2),(42,3)]
⇒ res0 : Maybe (Int, Int) = (Just (42,1))
>> findElt ((==) 42 . fst) [(1,1),(42,2),(42,3)]
⇒ res1 : Maybe (Int, Int) = (Just (42,2))
>> findElt ((==) 42 . fst) [(1,1)]
⇒ res2 : Maybe (Int, Int) = Nothing
>> findElt (const True) []
⇒ res3 : forall a. Maybe a = Nothing
```

6.8 Holes in the wrong places

Thus far, I've shown several examples of using holes as placeholders for code to be filled in later. However, all of these examples worked because I put the hole somewhere that could sanely be filled in with code and produce a correct definition.

Ermine can tell you in some cases when a hole couldn't possibly be in the correct place. This is usually because it failed somewhere that has nothing to do with the hole's position. That doesn't mean, just because you receive a "remembered terms" message, that it's in the right place.

Consider this example attempt to adapt `findElt` to the new type.

```
findElt : forall a. (a -> Bool) -> List a -> Maybe a
findElt p xs = _ (solution64 (filter p xs))
```

Ermine will allow this, and report a type for the hole.

```
>> :r

⊣ Remembered terms:

ChapterErrors.e:25:16: (!a -> Maybe !a) inferred from
findElt p xs = _ (solution64 (filter p xs))
               ^
```

And there's a function that looks just like that, `Just`. So what happens with the tests in Section 6.7 [A better `findElt` type], page 37?

```
>> :r
⊣ Reloaded one dirty module...
>> findElt ((==) 42 . fst) [(42,1),(2,2),(42,3)]
⇒ res0 : Maybe (Int, Int) = (Just (42,1))
>> findElt ((==) 42 . fst) [(1,1)]
error  res1 : Maybe (Int, Int) = (Just
error    <error: match error at...>)
```

Not what you're looking for. There's nothing sane you can put in this position to make the tests pass; the `solution64` simply must be replaced.

6.9 Fixing downstream functions

Now that `findElt` is sane, reintroduce `oftenFoundAt`.

```
>> :r
error  ChapterErrors.e:15:1: error: failed to unify type JobCluster
error    with type (Maybe JobCluster)
error  location (JobCluster _ o) = o
error  ^
```

You have this unfortunate report again, but we know how to think about this: the problem is in where `location` is being used. `findElt` gives back a `Maybe JobCluster`, but `location` expects a `JobCluster` as an argument.

The problem is adapting the `location` function to work with the result produced by `findElt`, which you can represent with a hole.

```
oftenFoundAt job = _ location (findElt jobis commonJobs)
  where jobis j = job == occupation j
```

Note that the hole isn't followed by a parenthesized call; it receives `location`, and then `findElt jobis commonJobs` as two arguments, and is expected to mediate them. How should it do that?

```
>> :r

⊣ Remembered terms:

examples/ChapterErrors.e:28:20: ((JobCluster -> String) ->
Maybe JobCluster -> a) inferred from
oftenFoundAt job = _ location (findElt jobis commonJobs)
                   ^
```

From experimenting with `oftenFoundAt` in Section 6.6 [Refine by adding type errors], page 36, you might *think* `a` should be `String`. But maybe it should be `Maybe String`. By leaving it `String`, you deny other functions built upon `oftenFoundAt` the opportunity to handle the case where no database entry was found. `findElt` now gives you the power to yield some "default" value, like `"no entry found"`, but that's very ad hoc and makes for much less generality.

So the type you want is:

```
(JobCluster -> String) -> Maybe JobCluster -> Maybe String
```

In Section 4.10 [A simpler `Which`], page 19, you wrote and tested just the function needed, for the `Umm` type, your equivalent to Ermine's standard `Maybe`. Here was its signature.

```
mapUmm : (a -> b) -> Umm a -> Umm b
```

Where `a = JobCluster` and `b = String`, this is the right type! Ermine defines it for you as `fmap maybeFunctor`, which calls out a pattern we'll talk more about in Section 7.5 [Admitting functors], page 48.

```
-- Add this to the imports at the top of the file.
import Control.Functor

oftenFoundAt job = fmap maybeFunctor location (findElt jobis commonJobs)
  where jobis j = job == occupation j
```

Let's see that work.

```
>> :r
⊣ Reloaded one dirty module...
>> :t oftenFoundAt
⇒ String -> Maybe String
>> oftenFoundAt "hacker"
⇒ res0 : Maybe String = (Just "Cambridge")
>> oftenFoundAt "lawyer"
⇒ res1 : Maybe String = Nothing
```

So `String -> Maybe String` seems like a good type for this function. If it had users expecting it to be `String -> String`, you'd now fix them, and so on.

6.10 Further refinement

Is `String -> Maybe String` good enough, though? Well, that's up to you.

What would change if you wanted to handle multiple locations for each `JobCluster`? If you wanted to represent that hackers are commonly found in San Francisco, as well as Cambridge, would you have multiple entries in the `commonJobs` database labeled "hacker", or have the second element of the `JobCluster` data constructor be a `List String`?

What if you wanted to have `oftenFoundAt` accept a database of `JobCluster`s as an argument, rather than always using the single, unchangeable `commonJobs` database? You could try the parameterizing technique, from Section 4.5 [Parameterizing functions], page 15, and see what happens.

We consider it a great positive that changing ideas about your program can be made with changes in your types, resulting in errors elsewhere. You can come up with the basic change you wish to make, experiment by fixing the errors Ermine reports, and then see if the new version of your program gives sensible results when you test.

6.11 Simple mistakes

Now we'll consider some common causes of minor confusion with regard to type errors.

6.11.1 Associativity confusion

Sometimes simple syntactic mistakes can manifest as type errors. Consider these two definitions of `compose` (see Section 3.9 [Combining functions], page 9).

```
compose f g x     = f (g x)
composeBad f g x  = f g x
composeBad' f g x = (f g) x
```

These have types as follows.

```
>> :r
⊣ Reloaded one dirty module...
>> :t compose
⇒ forall a b c. (a -> b) -> (c -> a) -> c -> b
>> :t composeBad
⇒ forall a b c. (a -> b -> c) -> a -> b -> c
>> :t composeBad'
⇒ forall a b c. (a -> b -> c) -> a -> b -> c
```

The two "bad" versions have the same type; the parentheses are in the wrong place, or missing. As it happens, the two "bad" versions have the same meaning in Ermine; four expressions `f a b c` are just like `((f a) b) c`, and so on.

But Ermine didn't tell you these definitions were wrong; they're perfectly sensible, and have use in some contexts. Add an explicit type, though, and you'll see the mismatch:

```
compose, composeBad, composeBad' : (b -> c) -> (a -> b) -> a -> c
```

Which results in:

```
>> :r
error ChapterErrors.e:34:1: error: failed to unify type !c with type (a -> b)
error composeBad f g x  = f g x
error ^
```

This kind of mistake especially plagues infix operators like `+` and `.` such as you'll see in TODO: *forward ref*. It's yet another good reason to add explicit types as you write new functions, as in Section 6.3 [Explicit types test sanity], page 33. When in doubt, and you're perhaps confused about a type error in an expression that could be ambiguous, such as the definition of `composeBad`, add parentheses to tell Ermine explicitly how you want your expression to be understood.

6.11.2 Stretch: Parameterization failures

> This section is aimed at readers already familiar with Haskell or advanced type systems, wanting to delve deeper into Ermine. You may try it now, but feel free to come back to it later once you have more of the basics down.

Here's a simple function that uses `compose` to combine 3 functions.

```
compose3 f g h = compose f (compose g h)
```

Which has type:

```
>> :r
⊣ Reloaded one dirty module...
>> :t compose3
forall c c1 b a. (c -> c1) -> (b -> c) -> (a -> b) -> a -> c1
```

That's about what's expected. Try parameterizing `compose`, following the approach in Section 4.5 [Parameterizing functions], page 15, though:

```
compose3' comp f g h = comp f (comp g h)
```

Which has type:

```
>> :t compose3'
⇒ forall a b. (a -> b -> b) -> a -> a -> b -> b
```

Well then. That *could* be right; what if you add the explicit type you just saw, with an extra for the `comp` argument?

```
compose3' : ((b -> c) -> (a -> b) -> a -> c)
         -> (c -> d) -> (b -> c) -> (a -> b) -> a -> d
```

That gives:

```
>> :r
error ChapterErrors.e:41:1: error: failed to unify type !c with type !b
error compose3' comp f g h = comp f (comp g h)
error ^
```

Not really what you're looking for, then. There is a specialization of `compose3` and `compose3'`'s type that works, though.

```
compose3' : ((b -> b) -> (a -> b) -> a -> b)
         -> (b -> b) -> (b -> b) -> (a -> b) -> a -> b
```

It's worth noting that the inferred, more general type of unannotated `compose3'` conforms to this one. Of course, this function is now pretty useless.

You need Section 5.3 [Higher-ranked functions], page 27 to solve this kind of thing. Ermine will never infer these types; you have to know exactly what you want, and ask for it, using those techniques. I recommend boxing values of higher-ranked type up in a data definition, as seen for `Monad` in Section 5.2 [Limits and alleviations in Legacy], page 26.

7 Refined data types

In Chapter 4 [Working with Data], page 12, you saw the essentials of building data types and their constellations of functions. Those weren't just the basics; that was almost everything there is to representing data in Ermine.

Using data types is a matter of looking at the situation you're in, and recognizing which combination best represents your situation. This isn't an area where research is finished; many of Ermine's unique features are there to help you represent data in ways that are very hard in other languages.

There isn't enough space here to explain every idea you might want to incorporate into your data types. You will discover new techniques by necessity and curiosity; as you build operations over your data types, you'll find situations that are hard to handle, and adjust your data types accordingly. Moreover, as you build your understanding of Ermine's type system, you'll imagine situations where you don't know how to explain what you mean in Ermine types, and seek advice from the community.

We've previously explored a couple advanced techniques in Chapter 5 [Universals and Existentials], page 26. I'm going to step back a bit, now, and look at some of the simpler tools and principles of data types you haven't seen yet.

7.1 Functions are data

The `:t` REPL command is probably the most useful of its commands. We've seen it used to inspect the types of data and functions.

```
>> :t (1, 2)
⇒ (Int, Int)
>> :t (+) 33
⇒ Int -> Int
```

Functions *are* data.

```
>> fp = ((+) 33, (++_String) "hello, ")
>> :t fp
⇒ (Int -> Int, String -> String)
>> fl = [(+) 33, (*) 33]
>> :t fl
⇒ List (Int -> Int)
```

Take one of those out.

```
>> :t fst fp
⇒ Int -> Int
```

That looks like a plain function. Try invoking it:

```
>> fst fp 100
⇒ res3 : Int = 133
```

Because functions are data, anywhere you've used a type variable to say "anything can go here", that means it could be functions too. Above are a pair of functions of different types, as well as a list of functions, `fl`, all of a single type. Aside from use as type parameter values, you can incorporate function types directly into your data type definitions; many important core data types do so.

In the same way that `(Int, String)` is the type of a pair of `Int` and `String`, `Int -> String` is the type of functions from `Int` to `String`. From there, you can see how Section 3.8 [Multiple arguments], page 8 work.

Functions are such an absurdly powerful representation of data that, theoretically, you could abandon *all* of the data type representation tools you've seen so far in favor of using functions for everything. Here's an extension of an exercise from Section 4.10 [A simpler Which], page 19, to illustrate. Create a new file, `ChapterRefined.e`, and write into it:

```
module ChapterRefined where

data Umm a = Nada | GotIt a

-- | Church-encoded version of Umm.
data UmmChurch a = UmmChurch (forall z. z -> (a -> z) -> z)

umm : b -> (a -> b) -> Umm a -> b
umm nada gotit Nada = _
umm nada gotit (GotIt a) = _

-- | The Nada data constructor for Church.
nadaChurch : UmmChurch a
nadaChurch = UmmChurch (n g -> n)

-- | The GotIt data constructor for Church.
gotitChurch : a -> UmmChurch a
gotitChurch a = UmmChurch (n g -> g a)

-- | The UmmChurch catamorphism.
ummChurch : b -> (a -> b) -> UmmChurch a -> b
ummChurch nada gotit (UmmChurch f) = _
```

Don't worry too much about the use of `forall` above; it was introduced in Section 5.2 [Limits and alleviations in Legacy], page 26; it's only important that you see that the data representation entirely consists of functions, nothing else.

You should have solved the definition of `umm` in Section 4.10 [A simpler Which], page 19. Fill in the hole for `ummChurch`, so named because this technique is called *Church encoding*. You'll have a little trouble testing it, though:

```
>> GotIt 3
res0 : Umm Int = (GotIt 3)
>> gotitChurch 3
res1 : UmmChurch Int = (UmmChurch <function>)
>> ummChurch nadaChurch gotitChurch (gotitChurch 3)
res2 : UmmChurch Int = (UmmChurch <function>)
```

The ability to inspect data, such as is done to print values of type `Umm Int` above, is key to the advantages of leaving functions out of your data representation.

Take care when avoiding functions, though: excluding a function from your data structures where one would be appropriate can add extreme complexity to what would be a

simple idea if you used functions as data. You should choose data type representations that let you build the operations around them that you want to have, and that goes for the decision of whether to incorporate functions as well.

7.2 Lambda expressions

I took a bit of a shortcut when defining `ummChurch` and `gotitChurch` above. Namely, the `(n g -> g a)` thing.

A *lambda expression*, often simply called a *lambda*, creates a function without making you give it a name, as in all previous definitions. Everything to the left of the `->` is an argument, and everything to the right is part of the result expression.

In this example, the lambda refers to `a`, which comes from *outside* the lambda expression; we say that it *captures* `a`.

Many function definitions with arguments to the left of the `=` you've seen could have been written using lambda expressions. Here are four equivalent versions of the same function, which you defined in Section 3.7 [Defining functions], page 7.

```
twoPlus x = (+) 2 x
twoPlus x = 2 + x
twoPlus = x -> (+) 2 x
twoPlus = x -> 2 + x
```

Thanks to the convenience of applying fewer arguments than a function "expects" (see Section 3.8 [Multiple arguments], page 8), lambda expressions are rarer in Ermine than in many other languages that support them. They're still an essential tool for building interesting generic operators, though.

For example, `headThing`, as defined in Section 4.14 [On `List`'s `foldr`], page 24, was made a bit awkward by the need to define `constU` for use as the first argument to `foldrThings`.[1] Let's see two definitions, using `List` and `Maybe` in place of `Things` and `Umm`.

```
import List
import Maybe

constU : a -> b -> Maybe a
constU a b = Just a

headThing, headThing' : List a -> Maybe a

headThing = foldr constU Nothing

headThing' = foldr (a b -> Just a) Nothing
```

7.3 Braces and brackets

In Section 6.2 [Building functions with type errors], page 32, you used brackets '`[]`' to build a `List`. Here's another one.

[1] With more experience in Ermine, you'll probably drop the lambda here again and just say `const . Just`, in so-called "point-free" style.

```
>> [1, 2]
⇒ res2 : List Int = [1,2]
```

This isn't built-in syntax; it happened because there are visible definitions of `cons_Bracket` and `empty_Bracket`. Equivalently,

```
>> cons_Bracket 1 (cons_Bracket 2 empty_Bracket)
res2 : List Int = [1,2]
>> :t empty_Bracket
forall a. List a
>> :t cons_Bracket
forall a. a -> List a -> List a
```

It's common, but unnecessary, for a `cons_Bracket` to follow this `a -> f a -> f a` pattern. In many cases where you frequently want to jam a bunch of values of a certain type in a list-like way into a single value, providing "brackets" is a convenient way to offer that.

Let's define an *isomorphism*, a pair of functions that are each others' inverse, and a way to compose a bunch of them.

TODO: *Where was* `(.)` *introduced, if ever?*

```
import Function

-- | Laws: f . g = id, g . f = id
data Iso a b = Iso (a -> b) (b -> a)

empty_Bracket_I : Iso a a
cons_Bracket_I : Iso b c -> Iso a b -> Iso a c

empty_Bracket_I = Iso id id
cons_Bracket_I (Iso bc cb) (Iso ab ba) =
  Iso (bc . ab) _
```

Solve that hole, and add a few more definitions to try this thing out.

```
import Control.Functor
import Num

-- Define a few isomorphisms.
maybeAndChurch : Iso (Maybe a) (UmmChurch a)
maybeAndChurch = Iso (maybe nadaChurch gotitChurch)
                     (ummChurch Nothing Just)

addOrSub1 : Iso Int Int
addOrSub1 = Iso ((+) 1) ((-) 1)

liftIso : Iso a b -> Iso (Maybe a) (Maybe b)
liftIso (Iso ab ba) = Iso (mapm ab) (mapm ba)
  where mapm = fmap maybeFunctor
```

Now you should see:

```
>> []_I
⇒ res0 : forall a. Iso a a = (Iso <function> <function>)
```

```
>> [maybeAndChurch, liftIso addOrSub1]_I
⇒ res1 : Iso (Maybe Int) (UmmChurch Int) = (Iso <function> <function>)
```

Braces '`{}`' are also available, for data types for which "empty" doesn't make sense. Instead, these are equivalent, with the same rules about suffix naming you used to add the `_I` suffix above:

```
{x, y, z}
snoc_Brace (snoc_Brace (single_Brace x) y) z
```

These are less common than brackets, because there is usually a useful empty or *identity* value, but useful for some data types. Note that, unlike with brackets, values are folded up from the left, rather than the right.

7.4 Stretch: Generalized isomorphisms

This section is aimed at readers already familiar with Haskell or advanced type systems, wanting to delve deeper into Ermine. You may try it now, but feel free to come back to it later once you have more of the basics down.

Let's play with polykinds!

```
import Control.Category

data NT f g = NT (forall a. f a -> g a)

-- fCategory : Category (->) defined for us.

ntCategory : Category NT
ntCategory = Category (NT id) cc
  where cc (NT gh) (NT fg) = NT (gh . fg)
```

Now let's make the generalization.

```
data GIso c a b = GIso (c a b) (c b a)

gisoId : Category c -> GIso c a a
gisoId cat = GIso (idC cat) (idC cat)

gisoComp : Category q -> GIso q b c -> GIso q a b -> GIso q a c
gisoComp cat (GIso bc cb) (GIso ab ba) =
    GIso (cc bc ab) (cc ba cb)
  where cc = comp cat

gisoCat : Category c -> Category (GIso c)
gisoCat cat = Category (gisoId cat) (gisoComp cat)
```

See how all the lovely polykinds are inferred.

```
>> :k Category
⇒ Category : forall a. (a -> a -> *) -> *
>> :k GIso
```

```
⇒ GIso : (a -> a -> *) -> a -> a -> *
>> :t gisoCat
⇒ forall {a} (c: a -> a -> *). Category c -> Category (GIso c)
```

And how everything falls into place.

```
>> :t gisoCat fCategory
⇒ Category (GIso (->))
>> :t gisoCat ntCategory
⇒ forall {a} . Category (GIso NT)
>> :t gisoCat (gisoCat ntCategory)
⇒ forall {a b} . Category (GIso (GIso NT))
```

Pay no attention to those unused kind variables!

7.5 Admitting functors

In Section 4.4 [A Practical Formula for Generality], page 14, you saw that introducing more polymorphism into your data types could help make the functions operating on them more general. However, some care in choosing your type parameters can give you access to a tremendous amount of functionality for free.

You've encountered, and implemented, numerous functions of the following form.

```
mapSecond : (b -> c) -> Pair a b -> Pair a c
mapUmm : (a -> b) -> Umm a -> Umm b
mapS : (a -> b) -> Stream a -> Stream b
mapThings : (a -> b) -> Things a -> Things b
```

These were defined in `ChapterData.e`; load that file in Ermine to follow along.

All of them take a function that transforms the last type parameter in the other argument. There's a strong enough type guarantee that we can't accidentally, say, leave some `b` in the `Pair`.

There's another way to look at these functions, though. Since functions are data, we can treat them all in the same way, after a fashion. See how it looks with some lining up and a little renaming.

```
mapSecond : (a -> b) -> Pair x a -> Pair x b
mapUmm    : (a -> b) -> Umm    a -> Umm    b
mapS      : (a -> b) -> Stream a -> Stream b
mapThings : (a -> b) -> Things a -> Things b
```

We also specified that for each of these functions *f*, `f id` is an identity. In `Control.Functor`, Ermine provides a data type to wrap up functions of this type, with that identity.

```
>> :t Functor
forall (f: * -> *). (forall a b. (a -> b) -> f a -> f b) -> Functor f
```

The *only* requirements for building a valid `Functor` are that you be able to apply the `Functor` data constructor, and that the function you give to it be an identity when given `id`. That's all.

The type of `Functor` has three type variables: `a`, `b`, and `f`. When we call `Functor` with each of those four functions, `a` and `b` will simply be the `a` and `b` of their signatures given above. But what about `f`? Ask what Ermine thinks they are.

```
>> :t Functor mapSecond
⇒ forall a. Functor (Pair a)
>> :t Functor mapUmm
⇒ Functor Umm
>> :t Functor mapS
⇒ Functor Stream
>> :t Functor mapThings
⇒ Functor Things
```

All four of these values of `f`, that is, `(Pair a)`, `Umm`, `Stream`, and `Things`, are a little odd. We can see how by looking back at their data type definitions, specifically, how their types are written:

```
data Pair a b
data Umm a
data Stream a
data Things a
```

So `Functor` chooses an `f` by peeling off the last type parameter for whatever data structure you're mapping with. There are many interesting functions defined for `Functor`, so it's useful to create a `Functor` for your data type if you can. But you can't do that if there's no type parameter to peel off!

Thus, when designing your own data types, an easy way to get an increase in power and flexibility is to add a type parameter if it's missing one, and describe one part of the data with that variable. That's just what you did earlier in Section 4.4 [A Practical Formula for Generality], page 14: `Pair` is simply `PairII` with extra type parameters. That `Functor` can be written for `Pair`, but not `PairII`, merely hints at the extra flexibility you gain by adding type parameters.

7.6 The custom/generic tradeoff

Ermine comes with a collection of generic, general-purpose data types, and various functions to work with them. So there is never a need to write a data type like `Things`, because `List` is already defined for you. Every function you've defined on `Things` is already defined in the `List` module, including the `Functor` above.

`Things` is perfectly serviceable as a way to represent zero or more things you might have. However, if you use it instead of `List`, you don't get to use all the tools that are already available for `List`.

It's so easy to reinvent parts of the core as part of your own data structures accidentally, such that reinvention is a great way to learn how Ermine works. `JobCluster` in Section 6.2 [Building functions with type errors], page 32 was such a reinvention.

```
data JobCluster = JobCluster String String

commonJobs : List JobCluster
commonJobs = [JobCluster "actor" "Hollywood",
              JobCluster "hacker" "Cambridge",
              JobCluster "financier" "London"]
```

This is just pair-of-two-strings again, and can be represented concisely with the already-defined type `(String, String)`, applying the built-in `(,)` type, mentioned in Section 4.3 [Two other things], page 13, to the `String` type twice.

```
jobCluster : String -> String -> (String, String)
jobCluster job w = (job, w)

commonJobs2 : List (String, String)
commonJobs2 = [jobCluster "actor" "Hollywood",
               jobCluster "hacker" "Cambridge",
               jobCluster "financier" "London"]
```

All the same data is present. Moreover, you can use things that are already defined for `List` and `(,)` to put together functions on the `commonJobs2` database.

```
import Pair

jobs : List String
jobs = fmap listFunctor ((JobCluster j _) -> j) commonJobs2

jobs2 : List String
jobs2 = fmap listFunctor fst commonJobs2
```

Ask Ermine what those values are:

```
>> :r
⊣ Reloaded one dirty module...
>> jobs
⇒ res0 : List String = ["actor","hacker","financier"]
>> jobs2
⇒ res1 : List String = ["actor","hacker","financier"]
```

You got only a tiny advantage by having `fst`, a function that takes the first part of a `(,)`-pair, already available. These advantages start to add up, though, and add up very quickly.

One occasional drawback is clarity in type errors. Let's say that you attempted to treat `jobs` as a job cluster database, redoing the extraction.

```
>> fmap listFunctor fst jobs
error <interactive>:1:1: error: failed to unify type (a, b) with type String
error fmap listFunctor fst jobs<EOF>
error ^
```

It's not really clear here that you're trying to treat a `String` as a job cluster database entry, because Ermine only knows that `(,)`-pairs are involved. With the specific structure, this is clear:

```
>> fmap listFunctor ((JobCluster j _) -> j) jobs
error <interactive>:1:1: error: failed to unify type JobCluster with type String
error fmap listFunctor ((JobCluster j _) -> j) jobs<EOF>
error ^
```

Also, suppose we wanted more data to show up in each `JobCluster` entry? We can just add it to the `JobCluster` data type declaration, and fix up the resulting type errors until

it works again. But you can't add anything to `(String, String)`; if you want to change the data, you have to change the type.

I suggest the following compromise: for most data, use ad hoc combinations of predefined data types, like `(String, String)`. For data that is highly prone to structural changes, define new data types, *but reuse as many existing data types as possible*. For example, it's possible to define a full data type for `commonJobs` as follows:

```
data JobsDB = Empty | JobCons String String JobsDB
```

But it's better to reuse `List`, and `(,)`, so that tools for those types can be reused.

```
data JobsDB2 = JobsDB2 (List (String, String))
```

This `JobsDB2` is a simple wrapper around a combination of standard data types. It's easy to implement functions over `JobsDB2` simply by extracting the `List (String, String)` and then wrapping such values back up when we're done, and we still have the freedom to change the `JobsDB2` type.

7.7 Avoid pattern matching, or not

In Section 4.2 [Taking data out], page 12, you saw the most primitive technique for taking data out of its enclosing structure, pattern matching.

Data types like `JobCluster` have various associated functions that might use pattern matching to extract information, such as the definition of `jobs` above. Say you changed `JobCluster` to have another piece of data, though.

```
>> data JobCluster3 = JobCluster3 String String String
|>
>> :t ((JobCluster3 j _) -> j)
error  <interactive>:1:3: error: expected constructor with two arguments
error  ((JobCluster3 j _) -> j)<EOF>
error    ^
```

Even if the function is only interested in the first element, it still needs to be updated with another `_` to mark that it doesn't care about the newly-added third `String`! The problem with adding to data types is that every existing pattern match over them needs to be updated.

However, this is a feature! Frequently, when you are adding more values to a data type, you *want* Ermine to tell you everywhere you're using the data type, because you'll frequently need to handle the newly-added data in those places. The failure to acknowledge part of a data structure, by simply ignoring it, is a source of many bugs.

This isn't true of all types, though. You can write catamorphism-like functions that run only over part of a data structure:

```
>> jc3JobPlace f (JobCluster3 j p _) = f j p
>> :t jc3JobPlace
⇒ forall a. (String -> String -> a) -> JobCluster3 -> a
```

This is like a catamorphism, but drops the new third value. If I had a catamorphism-like function for just those first two values *before*, I can continue to have such a function *after*. So functions that only care about the first two pieces of data, and will only ever care about the first two pieces of data, can use a function like this one instead of pattern matching.

There's a special case of this, where you're interested in only one value of the structure.

```
>> jc3Job f (JobCluster3 j _ _) = f j
>> :t jc3Job
⇒ forall a. (String -> a) -> JobCluster3 -> a
```

There's little point in bothering with `f` in this case.

```
>> jc3Job' (JobCluster3 j _ _) = j
>> :t jc3Job'
⇒ JobCluster3 -> String
```

7.8 Stretch: van Laarhoven lens families

> This section is aimed at readers already familiar with Haskell or advanced type systems, wanting to delve deeper into Ermine. You may try it now, but feel free to come back to it later once you have more of the basics down.

Thanks to higher-kinded types, something similar to the encoding in the lens library can be used to define composable lens families in Ermine.

```
-- | A type alias for a lens-style lens.
type LensFamily f s t a b =
  Functor f -> (a -> f b) -> s -> f t

-- | Sample lens: the snd of a pair.
sndL : LensFamily f (x, a) (x, b) a b
sndL ff f (x, a) = fmap ff ((,) x) (f a)
```

Functions over these lenses take the `Functor` evidence, the `ff` argument above, explicitly and pass it around. So, while an identity and composition can be defined for these lenses, they're not as convenient as the Haskell versions, because there is no implicit `Functor` constraint being propagated automatically:

```
-- | Lens family identity.
idL : LensFamily f a a a a
idL ff = id

infixr 9 .:

-- | Indexed associative composition where idL is a left and right
-- identity.
(.:) : LensFamily f s t q r
    -> LensFamily f q r a b
    -> LensFamily f s t a b
(.:) lf lg ff = lf ff . lg ff
```

So `(.:)` is *almost* `(.)`, but not quite.

Now you have an algebra, so can start putting lenses together.

```
>> :t sndL .: sndL
⇒ forall (f: * -> *) a b x x1.
  Functor f -> (a -> f b) -> (x, (x1, a)) -> f (x, (x1, b))
```

As in lens, you can introduce specific functors to interpret any lens. Here's the basic getter, `^.`:

```
-- | The const functor; used as a utility for ^..
data Const s a = Const s
runConst (Const s) = s
constF = Functor (_ (Const s) -> Const s)

infixl 8 ^.
(^.) : s -> LensFamily (Const a) s s a a -> a
(^.) s lf = runConst $ lf constF Const s
```

And the setter.

```
-- | The identity functor; used as a utility for .~.
data Id a = Id a
runId (Id a) = a
idF = Functor (f (Id a) -> Id (f a))

infixr 4 .~
(.~) : LensFamily Id s t a b -> b -> s -> t
(.~) lf b s = runId $ lf idF (const (Id b)) s
```

I've avoided rank-N types in the encoding above; for tips on what to change to use them, see Section 5.3 [Higher-ranked functions], page 27. It's also possible to use `data` instead of an unboxed function type to make the higher-rank encoding more convenient. Here's what those look like; replace `type LensFamily`'s definition above, and remove the first type argument to `LensFamily` in all cases, to try these forms out.

```
type LensFamily' s t a b =
  forall f. Functor f -> (a -> f b) -> s -> f t

data LensFamily'' s t a b = LensFamily''
  (forall f. Functor f -> (a -> f b) -> s -> f t)
```

Given the current state of Ermine type inference, the last version is probably the most convenient.

Appendix A Acknowledgements

Appendix B GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W