Some discrete math in Haskell

William Chargin

9 January 2016

Introduction

This program ports a specific Mathematica notebook to Haskell. In doing so, we hope to write a more easily understandable program.

The code comprises three modules. In *CAS*, we implement a *very* basic CAS so that we can write and evaluate symbolic expressions. In *Liese*, we port some code provide by Dr. Liese as a preamble to the problem. Finally, in *Main*, we present the solution to the problem.

The reader is not assumed to have any Haskell knowledge. We'll see how this goes. . .

A quick note on notation

This document is written in Literate Haskell; the LaTeX source for the document is also the Haskell source! (The Haskell compiler knows to ignore anything outside a `\begin{code} \end{code}` environment if it detects LaTeX input.)

Usually, Literate Haskell does quite a good job with the formatting. However, it does rather poorly with infix operator sections. Thus, we explicitly explain the following notations:

- the expression $(2/)$ is called an *left operator section*, and represents a function that will divide 2 by its argument, so it is the same as $\lambda x \rightarrow 2 \, / \, x$;
- the expression $(/2)$ is called an *right operator section*, and represents a function that will divide its argument by 2, so it is the same as $\lambda x \rightarrow x \, / \, 2$;
- the expression $x \,`\mathit{div}`\, y$ is entered `x 'div' y`, and is the same as $\mathit{div}\ x\ y$; that is, it treats *div* as an infix operator and applies it to two operands;
- the expression $(`\mathit{div}`3)$ is written `('div' 3)`, and it is a *right infix operator section*, which does exactly what you think it would do (it is equivalent to $\lambda x \rightarrow x \,`\mathit{div}`\, 3$).

The only really confusing one of these is the last, where the lack of space between the trailing ` and the 3 can make these expressions difficult to read. Sorry about that!

1 Some basic Haskell

This document doesn't assume any knowledge of Haskell. Let's introduce some basic concepts.

> You're welcome to skip this whole section, if you want.

1.1 Functions

Haskell is all about *functions*. As in mathematics, a function takes input values to output values, where the input values are in the domain and the output values are in the codomain. In Haskell, we write $f :: \alpha \to \beta$ to indicate that f has domain α and codomain β.

Here's a function:

$$\begin{array}{l} \mathit{double} :: \mathit{Int} \to \mathit{Int} \\ \mathit{double}\ x = 2 * x \end{array}$$

The *type annotation* on the first line is not required; Haskell will infer it if you omit it. (In fact, Haskell will always infer it; if you include it, Haskell will complain if yours is wrong.)

We could also write this as follows:

$$\begin{array}{l} \mathit{double} :: \mathit{Int} \to \mathit{Int} \\ \mathit{double} = \lambda x \to 2 * x \end{array}$$

The term $\lambda x \to 2 * x$ is called a *lambda expression*; it is used to represent a notion of a mapping without assigning it a name. Of course, in this case, we do immediately assign it a name. But that need not be the case, as we will see soon.

1.1.1 Functions are values

A key concept is that functions are values just as much as *Int*s are. So we can write something like this:

$$\begin{array}{l} \mathit{multiplyBy} :: \mathit{Int} \to (\mathit{Int} \to \mathit{Int}) \\ \mathit{multiplyBy}\ x = (\lambda y \to x * y) \end{array}$$

Thus, *multiplyBy* expects an *Int*, and returns *a function* from *Int* to *Int*.

In fact, this is so common that we insist that **the** $(\to)$ **type operator is right-associative**, so $a \to (b \to c)$ can be written just as $a \to b \to c$.

The reason this is common is because it enables multi-argument functions: a "function of two arguments" is really just a function that returns another function. Here's another example:

$$\begin{array}{l} \mathit{addThree} :: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \to \mathit{Int} \\ \mathit{addThree}\ x = \lambda y \to \lambda z \to x + y + z \end{array}$$

And, of course, this is tedious, so we may just as easily write:

$$\begin{aligned}&\mathit{addThree}' :: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \to \mathit{Int}\\&\mathit{addThree}'\ x\ y\ z = x + y + z\end{aligned}$$

1.1.2 Applying functions

Suppose we want to add 5, 6, and 12 using $\mathit{addThree}'$ from above. First, we need to invoke $\mathit{addThree}'$ with argument 5 to get a function. Then, we call the resulting function with 6 to get a second function. Finally, we call this last function with 12 to get the result.

So we want to write $((\mathit{addThree}'\ 5)\ 6)\ 12$.

Of course, this is equally tedious, so we insist that **function application is left-associative**, so we can write the above as $\mathit{addThree}'\ 5\ 6\ 12$.

1.1.3 Partial application

Partial application falls very naturally out of the above. Consider this example:

$$\begin{aligned}&\mathit{distance} :: \mathit{Float} \to \mathit{Float} \to \mathit{Float}\\&\mathit{distance}\ a\ b = \mathit{sqrt}\ (a * a + b * b)\\&\mathit{distanceFrom0} :: \mathit{Float} \to \mathit{Float}\\&\mathit{distanceFrom0} = \mathit{distance}\ 0\end{aligned}$$

In the definition of $\mathit{distanceFrom0}$, we just take our existing function and apply it to only one argument. This leaves us with a function from Float to Float, which does exactly what we want.

This pattern is very common:

$$\begin{aligned}&\mathit{double} &&= (*)\ 2\\&\mathit{reciprocal} &&= (/)\ 1 \quad \text{-- note that we're specifying the first (left) operand}\end{aligned}$$

1.1.4 Higher-order functions

Suppose we have a list of integers—in particular, suppose $\mathit{xs} :: [\mathit{Int}]$. How might we double every element in this list? We could write

$$\begin{aligned}&\mathit{double}\ x &&= x * 2\\&\mathit{doubleAll}\ \mathit{xs} &&= \mathit{map}\ \mathit{double}\ \mathit{xs}\end{aligned}$$

Here, map is a built-in function. Its type is $\mathit{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta]$. That is, provided with *any* function from some type α to another type β, and also a list of elements whose type is the domain of the function (i.e., α), map will produce the result of applying the function to every value in the list.

We call map a **higher-order function** because it accepts a function—in this case, of type $\alpha \to \beta$—as an argument.

Here's another higher-order function in use:

```
evenNumbersUnder100 :: [Int]
evenNumbersUnder100 = filter even [1 .. 100]
```

The $\mathit{even} :: \mathit{Int} \to \mathit{Bool}$ function is a built-in predicate, and $\mathit{filter} :: (\alpha \to \mathit{Bool}) \to [\alpha] \to [\alpha]$ takes just those elements of a list matching the given predicate.

We can combine these, e.g., to write

```
square :: Int → Int
square x = x * x

under1000 :: Int → Bool
under1000 x = x < 1000

allTheNumbers :: [Int]
allTheNumbers = [1 ..]

squaresUnder1000 = filter under1000 (map square allTheNumbers)
```

Note that we need parentheses in the higher-order function type signatures; if we wrote $\mathit{notMap} :: \alpha \to \beta \to [\alpha] \to [\beta]$, then *notMap* would just be a function taking three arguments—an α, a β, and an $[\alpha]$—none of which is a function.

1.2 Data

The other concept that's as important as functions is data: the construction and use of data types.

You've already seen types like *Int*, *Bool*, and *String*. But you can make your own types, too. An *algebraic data type* is a type with one or more *variants*. Suppose we want to represent vehicles, which can either be cars or bicycles. Suppose further that we want to keep track of the make and model of each car and the number of gears on each bike. We can use the following declarations:

```
data Vehicle = Car String String
             | Bike Int

myCar :: Vehicle
myCar = Car "BMW" "Jetta"

myBike :: Vehicle
myBike = Bike 18
```

There are two important things to note here. First, we've only added one type: *Vehicle*. Note that we *don't* write $\mathit{myCar} :: \mathit{Car}$ or $\mathit{myBike} :: \mathit{Bike}$, because *Car* and *Bike* are not types.

What are they, then? Well, if *Car* `"BMW"` `"Jetta"` :: *Vehicle*, and `"BMW"` :: *String* and `"Jetta"` :: *String*, then we must have $\mathit{Car} :: \mathit{String} \to \mathit{String} \to \mathit{Vehicle}$. Similarly, $\mathit{Bike} :: \mathit{Int} \to \mathit{Vehicle}$. So **data constructors are just functions**.

1.2.1 Pattern matching

Let's write a function that gets the cost of a vehicle. Everyone knows that vehicles with longer makes and models are more valuable, so we'll use the formula $cost_{car} = 10 \cdot length_{make} + length_{model}$ and $cost_{bike} = 5 \cdot gears$. That is:

```
cost :: Vehicle → Int
cost (Car make model) = 10 ∗ length make + length model
cost (Bike gears)     = 5 ∗ gears
```

Note that we've written *two* definitions for *cost*: one for each variant of the input argument. In fact, this is a pretty common practice:

```
factorial :: Int → Int
factorial 0 = 1
factorial n = n ∗ factorial (n − 1)
length′ :: [a] → Int
length′ []           = 0
length′ nonEmptyList = 1 + length′ (tail nonEmptyList)
                       -- (where tail [1, 2, 3] ≡ [2, 3])
```

You can use the _ wildcard to match any pattern and ignore its value:

```
sameType :: Vehicle → Vehicle → Bool
sameType (Car _ _) (Car _ _) = True
sameType (Bike _) (Bike _)   = True
sameType _ _                 = False   -- Car and Bike or vice versa

data TrafficLight = Red | Yellow | Green
shouldGo :: TrafficLight → Bool
shouldGo Red = False
shouldGo _   = True   -- for both Yellow and Green
```

We'll use pattern-matching extensively.

1.2.2 Recursive data types

Suppose we want to write a data type for a list of integers (and suppose we "forgot" about the built-in list type). A list can either be empty or not. If it's not, it has a first element and the rest of the list, and the rest of the list is just another list! That is,

```
data IntList = Empty
             | NonEmpty Int IntList
emptyList :: IntList
emptyList = Empty
```

```
oneTwoThree :: IntList
oneTwoThree = NonEmpty 1 (NonEmpty 2 (NonEmpty 3 Empty))
```

The interesting thing here, of course, is that an *IntList* can have another *IntList* as its field. This makes it a recursive data type. Think about this if it confuses you. We'll use these extensively, too.

1.2.3 Parametric data types

Suppose we now want a *StringList*. We could create another *StringList* type, and maybe rename the old constructors *EmptyIntList* and *NonEmptyIntList* so the names don't clash. But this isn't a good solution; any functions we make will have to be duplicated. And of course we'd need to put in even more work to make a *BoolList*, etc.

Instead, what about a list of elements of some arbitrary type?

```
data List α = Empty
            | NonEmpty α (List α)

emptyIntList :: List Int
emptyIntList = Empty

helloWorld :: List String
helloWorld = NonEmpty "hello" (NonEmpty "world" Empty)
```

Here, α is a *type parameter*. Note that *List Int* and *List String* and *List Bool* are all types. For that matter, *List* (*List String*) is a type! But, crucially, *List* by itself is *not* a type—it is a *type constructor*. It needs a type parameter, α, to create a concrete type.

(In fact, we say that $\mathit{List} :: * \rightarrow *$, where :: here is read as "has kind"; kinds are like types of types, and $*$ is the kind of concrete types, so $* \rightarrow *$ is the kind of a type constructor that takes one concrete type (α) and returns another ($\mathit{List}\ \alpha$).)

1.2.4 More

There's a lot more to say about data types, but we won't.

1.3 More

There's a lot more to say about Haskell, but we won't. Hopefully, the syntax will be readable enough. From time to time, when we do want to explain something, you may come across notes like these:

> **Notes**
> Please note this note. This note is a sample note, so I suppose it's a note about notes.

These are notes about the Haskell language itself.

With no further ado, we'll get right to it!

2 The CAS

We'll start with a digression: we need to develop a mini-CAS. All we need to support is the following:

- we must be able to refer to symbolic variables;
- we must be able to combine variables, constants, and operators to form expressions; and
- given scalar values for all the variables, we must be able to provide a value for the entire expression (i.e., evaluate it).

We will see that this task is quite easy for Haskell.

First, the following **module** declaration indicates that we're creating a new Haskell module called *CAS*. This is how we'll use our CAS in our other modules.

```
module CAS where
```

We'll need these utilities later on; pay them no heed...

```
import Control.Monad (liftM, liftM2)
```

2.1 Symbols

We'll start by defining a data type for symbols. For our purposes, it'll be sufficient to have plain variables (e.g., x) and symbols with subscripts (e.g., x_3). So we're defining a data type with two variants:

```
data Symbol = Var Char
            | Symbol `Sub` Int
              deriving (Eq, Show)
```

For example, we might use a variable *Var* 'x', or its subscript form *Var* 'x' ‘*Sub*‘ 3. Note that, because the *Sub* constructor accepts any symbol, we can take subscripts of subscripts: e.g., *Var* 'x' ‘*Sub*‘ 3 ‘*Sub*‘ 6 is valid.

Infix data constructors
Note that the second variant is defining a data constructor named *Sub*, which takes two arguments: a *Symbol* and an *Int*. We could have also written *Sub Symbol Int* for this variant. However, we choose to write it in infix notation because that's how it will most frequently be used: you could, of course, write *Sub* (*Var* 'x') 3, but *Var* 'x' ‘*Sub*‘ 3 is more faithful to the pronunciation.

The first **deriving** declaration is important: it says that we have a total equivalence relation on *Symbol*, so that the operation *Var* 'x' ≡ *Var* 'x' is well-defined. (The compiler automatically figures out the equivalence relation and implements it for us.) The second one just lets us display *Symbol*s for debugging or output.

2.2 Expressions

Next, we need to define some basic arithmetic expressions. For generality's sake, we'll let these be over an arbitrary set of scalars; thus, instead of just describing an *Expression*, we'll describe, say, an *Expression Int*.

As before, we'll start by creating a type with a few variants. I'll write the full definition and then explain each variant in turn:

```
data Expression a = Constant a
                  | Literal Symbol
                  | MonOp (a → a) (Expression a)
                  | BinOp (a → a → a) (Expression a) (Expression a)
                  | NOp ([a] → a) [Expression a]
```

The first variant is reasonably clear: a constant term requires a scalar value of type *a*. So if our scalars are the integers, we can write *Constant* 10; if our scalars were, say, strings, we could write *Constant* `"hello"`.

Finding types
You can verify that *Constant* 10 :: *Expression Int* and *Constant* `"hello"` :: *Expression String* by typing `:type Constant 10` or `:type Constant "hello"` into GHCi, the interactive Haskell interpreter. (Launch it with `ghci`.)

Another important core term type is a symbol as an expression; e.g., the first term in the expression $x + 3$. This is what the second variant expresses. So we could write *Literal* (*Var* `'x'`) or *Literal* (*Var* `'x'` ‘*Sub*‘ 3).

Next, we could go ahead and add variants specifically for sums, products, etc. But we're going to need a bunch of them, so we might as well define them for arbitrary unary, binary, and *n*-ary operations.

These three variants work as follows. The *MonOp* variant requires two parameters: a function of type $a \to a$ (remember, *a* is the set of scalars, so this might be, e.g., *Int* → *Int*), and an expression to which to apply that function. So, we might write something like this:

```
double :: Int → Int
double x = 2 * x   -- or double = (2*)

baseExpr :: Expression Int
baseExpr = Literal (Var 'x')

doubledExpr :: Expression Int
doubledExpr = MonOp double baseExpr
```

In a sense, this lets us "pull up" a normal scalar function into a function that acts on an expression.

Similarly, *BinOp* takes a binary function and two expressions. So we could easily write *sumExpr* = *BinOp* (+) *exprA exprB*.

The case of *NOp* is slightly more interesting, because it takes a function whose input is a list of scalars and whose output is a single scalar, and also takes a list of expressions. So, for example, because *sum* :: [*Int*] → *Int* is a built-in function, we can write *NOp sum* [*exprA*, *exprB*, *exprC*, *exprD*]. This is clearly generalizable to *NOp product*, etc.

2.3 Environments

To evaluate an expresion, we'll need to keep track of the variable values. We'll force the user to provide an *Environment*, which is just a partial function from symbols to scalars:

```
type Environment a = Symbol → Maybe a
```

Note that we're using **type** to define a type alias; that is, anywhere we write *Environment a*, we could just as well write *Symbol* → *Maybe a*.

Also note that the codomain of an *Environment a* is a *Maybe a*; this represents a value that could be absent. For example, if we only know the values for *Var* `'x'` and *Var* `'y'`, then we could write an environment as follows:

```
myEnvironment :: Environment Int
myEnvironment (Var 'x') = Just 12
myEnvironment (Var 'y') = Just 33
myEnvironment _         = Nothing
```

This indicates a piecewise-defined function where we've specified values for *Var* `'x'` and *Var* `'y'` and let everything else be absent (*Nothing*).

Of course, we expect the user to define all the variables we need, so if we ever actually come across a *Nothing* while evaluating, we should complain loudly.

Creating simple environments more easily

We can also create a helper function to convert a list of bindings to an environment function. That is, we'd like to be able to write

```
myEnvironment′ = environmentFromList [(Var 'x', 12), (Var 'y', 33)]
```

and have it behave the same as *myEnvironment*.

We can take advantage of the built-in *lookup* function, which works as follows:

```
lookup 10 [(10, "dog"), (30, "cat")] ≡ Just "dog"
lookup 20 [(10, "dog"), (30, "cat")] ≡ Nothing
```

So our definition might be

```
environmentFromList :: [(Symbol, a)] → Environment a
environmentFromList table = λsymbol → lookup symbol table
```

But this is an unnecessary use of a lambda function; remember that *Environment a* ≡ *Symbol* → *Maybe a*, so we can actually write

```
environmentFromList :: [(Symbol, a)] → Symbol → Maybe a
environmentFromList table symbol = lookup symbol table
```

or

```
environmentFromList :: [(Symbol, a)] → Environment a
environmentFromList table symbol = lookup symbol table
```

There's one more simplification we can make. We can now see that this function literally is *lookup* except that its arguments are flipped. The built-in *flip* combinator, whose type is $\mathit{flip} :: (a \to b \to c) \to (b \to a \to c)$ and whose implementation is $\mathit{flip}\ f\ x\ y = f\ y\ x$, lets us write the final version:

```
environmentFromList :: [(Symbol, a)] → Environment a
environmentFromList = flip lookup
```

> **Point-free style**
> Note that we defined *environmentFromList* without actually referring to its parameters; instead, we applied a higher-order function (*flip*) to an existing function (*lookup*). This is called *point-free style*; the terminology arises from thinking of types as topological spaces and values are points in those spaces, so we're not referring to the "points" in the definition.

Evaluation

We now have all the pieces necessary to write the evaluator. First, the type signature:

```
eval :: Environment a → Expression a → Either String a
```

The first two components should be no surprise: we created the *Environment*s to be used in the evaluator, and we obviously need an expression to evaluate.

The return type may be new to you. It's an *Either String a*, which can either be an error message (of type *String*) or the scalar result (of type *a*). Here are two values of that type:

```
valueError :: Either String Int
valueError = Left "you forgot the flux capacitor"

valueSuccess :: Either String Int
valueSuccess = Right 71
```

(You can remember which is which because *Right* is the right answer!)

> **Defining *Either***
> The (built-in) definition of *Either* is simply:

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

Now we can implement the evaluator in a piecewise manner. First, we address constant terms. When we evaluate a constant term, we don't care about the environment, and the computation will always succeed. So we can just write the rule

```
eval _ (Constant n) = Right n
```

where the _ indicates that we discard the first argument.

The next variant of an *Expression* is a *Literal*, like *Literal* (*Var* `'x'`). Here, we'll need to look something up in the environment:

```
eval env (Literal x) = case env x of
  Just val → Right val
  Nothing → Left $ "no such symbol: " ++ show x
```

Recall that the environment is just a function, so we apply that function to the variable symbol and then inspect the result. If we got a successful value, that's the return value, so we wrap it up in a *Right* and we're done. Otherwise, the user failed to provide a value, so we return an error indicating which variable was missing. Here, we use the *show* function, which converts any *Show*able value into a string so that we can add it to our error message.

The $\$$ function

This is a built-in function that we use to eliminate parentheses. Function application is left-associative, so $foo\ bar\ baz\ quux \equiv ((foo\ bar)\ baz)\ quux$. This is necessary because all functions are *curried*: a "two-argument" function is really a function that returns another function. For example, $take\ 3\ [1\,..\,10] \equiv [1,2,3]$. But this decomposes to $(take\ 3)\ [1\,..\,10]$, so $take\ 3$ is actually a function that can take the first three elements of any list. So $take$ must have type $Int \rightarrow ([a] \rightarrow [a])$, which is what we mean when we write $take :: Int \rightarrow [a] \rightarrow [a]$.

But if we want to write, say, $length\ (filter\ null\ (map\ myFunc\ list))$, then the function application isn't purely left-associative (as indicated by the parentheses).

The $\$$ function is simply defined as $f \mathbin{\$} x = f\ x$, but it has very low precedence and is defined to be right-associative. So you can write

```
length $ filter null $ map myFunc list
```

to mean the same as the original expression.

All the remaining variants are interesting because they involve sub-expressions. For example, let's think about how we'd evaluate *BinOp* (+) *lhs rhs*:

1. Evaluate *lhs* as a sub-expression.

2. If the evaluation failed, return the error immediately. Otherwise, get the resulting value; call it *va*.

3. Evaluate rhs as a sub-expression.
4. If the evaluation failed, return the error immediately. Otherwise, get the resulting value; call it vb.
5. Compute $\mathit{va} + \mathit{vb}$, and wrap it up in a Right.

This sounds like a lot of complicated logic, so it may surprise you that the answer is a one-liner:

$$\mathit{eval}\ \mathit{env}\ (\mathit{BinOp}\ \mathit{op}\ a\ b) = \mathit{liftM2}\ \mathit{op}\ (\mathit{eval}\ \mathit{env}\ a)\ (\mathit{eval}\ \mathit{env}\ b)$$

It's clear that $\mathit{eval}\ \mathit{env}\ a$ and $\mathit{eval}\ \mathit{env}\ b$ correspond to steps 1 and 3 in our plan above. The magical $\mathit{liftM2}$ function is the "glue" that carries out the rest of the steps. In particular, this function says:

> I see that your function returns an $\mathit{Either}\ \mathit{String}\ a$, so I'll treat it as a computation that can either fail or succeed. You gave me a binary operator—op—that applies to the scalars, and two results of sub-expressions, either of which may be an error. I'll try to apply them if I can; otherwise, I'll return the first error.

In other words, $\mathit{liftM2}$ does *exactly* what we want!

A taste of monads
The real beauty of $\mathit{liftM2}$ is that it's more general than this! In fact, it can work in any *monad*; see later for a crash course.

Similarly, of course, liftM works on unary functions:

$$\mathit{eval}\ \mathit{env}\ (\mathit{MonOp}\ \mathit{op}\ a) = \mathit{liftM}\ \mathit{op}\ (\mathit{eval}\ \mathit{env}\ a)$$

We need a slightly different kind of glue for the NOp case. Here's what we'll write:

$$\mathit{eval}\ \mathit{env}\ (\mathit{NOp}\ \mathit{op}\ \mathit{as}) = \mathit{liftM}\ \mathit{op}\ \$\ \mathit{mapM}\ (\mathit{eval}\ \mathit{env})\ \mathit{as}$$

Let's look at the right-most sub-expression first: $\mathit{mapM}\ (\mathit{eval}\ \mathit{env})\ \mathit{as}$.

First of all, $\mathit{eval}\ \mathit{env}$ is the result of partially applying eval in the current environment; that is, $\mathit{eval}\ \mathit{env} :: \mathit{Expression}\ a \to \mathit{Either}\ \mathit{String}\ a$ is a function that will evaluate any expression with a fixed environment.

Then we call mapM. Here, mapM has type

$$\begin{aligned} \mathit{mapM}\ ::\ & (\mathit{Expression}\ a \to \mathit{Either}\ \mathit{String}\ a) \\ & \to [\mathit{Expression}\ a] \\ & \to \mathit{Either}\ \mathit{String}\ [a] \end{aligned}$$

so it can either have a single error message or a list of results. The call to mapM will apply this function to each of the as; if any of them fails, it'll return that error message, and otherwise it will return a list of all the results. So, all together, the type of this sub-expression is

```
mapM (eval env) as :: Either String [a]
```

Finally, we have this function *op* :: [*a*] → *a*, provided by the user, and we have an *Either String* [*a*] that we'd like to apply it to. Again, *liftM* is helpful: it "lifts" our normal function into an *Either*-like function, so *liftM op* :: *Either String* [*a*] → *Either String a*. Then we can apply this directly to the result from *mapM* and we have the result of evaluation!

If all of this monadic stuff makes little sense to you yet, don't worry. There's a lot of type-juggling, and moving into and out of different levels of abstraction (*Either*s, in this case). It takes a while to get used to, but is extremely powerful once you're comfortable.

Anyway, this completes the definition for our evaluator, and that's all we need for our mini-CAS!

2.4 Example

Let's write $(x_1 + x_2 + x_3)^2 + x_3^4$.

```
xsub :: Int → Symbol
xsub i = Var 'x' `Sub` i

xsubs = map (Literal ∘ xsub) [1, 2, 3]
  -- Note: xsub :: Int → Symbol
  -- and Literal :: Symbol → Expression a,
  -- so Literal ∘ xsub :: Int → Expression a
  -- is a function that generates expressions over any scalar set.

squaredQuantity = MonOp (↑2) $ NOp sum xsubs

x3tothe4 = MonOp (↑4) (Literal $ xsub 3)

expr = BinOp (+) squaredQuantity x3tothe4
```

Then we can evaluate it with some variable assignments:

```
env = environmentFromList [(xsub 1, 10), (xsub 2, 20), (xsub 3, 30)]
result = eval env expr   -- comes out to Right 813600
```

Note that we can write nice composable units like *xsub* for building up larger expressions, and that we can use existing Haskell functions like *map* to do the same.

3 Dr. Liese's code

Dr. Liese provided some Mathematica code. Let's write it in Haskell.

```
module Liese where
```

First off, we'll need to pull in the `PermutationGroup` module. This provides functions for, e.g., converting permutations to cycle decompositions and vice versa, finding the order of a permutation group, etc. By importing this, we save ourselves from having to rewrite all that (and of course this functionality is built into Mathematica).

```
import Math.Algebra.Group.PermutationGroup
  (Permutation, toCycles, elts)
```

> **Import lists**
> The parenthesized import list indicates that we only want those types and functions; this makes it easy to tell what comes from where when reading the code. If you've imported ten modules from various sources and see a function you haven't heard of, it can be difficult to determine where it comes from if there aren't any import lists!

We're importing the *Permutation* type, and two functions. The *toCycles* function takes a *Permutation* to its cycle decomposition. The *elts* function is like Mathematica's `GroupElements` function: its type is [*Permutation*] → [*Permutation*], and it finds the closure of a permutation group given its generators.

Of course, we also want our own CAS.

```
import CAS
```

3.1 *cycleOrders*

We start by replicating the "cycle type" function from the Mathematica code; this function, given a permutation group and its order, finds the length of each of the disjoint cycles.

```
cycleOrders :: Ord a ⇒ Permutation a → Int → [Int]
cycleOrders grp n = nontrivials ++ trivials
  where
    cycles      = toCycles grp
    numTrivials = n − length (concat cycles)
    trivials    = replicate numTrivials 1
    nontrivials = map length cycles
```

Note that *toCycles* is a function from that library we imported, whose type signature is *toCycles* :: *Permutation a* → [[*a*]].

Also, *replicate* :: *Int* → *a* → [*a*] is a built-in function; it's defined such that, e.g., *replicate* 3 'x' = ['x', 'x', 'x'].

So, for example, the following should hold (up to order of the resulting list, which we don't really care about):

```
import Math.Algebra.Group.PermutationGroup (fromCycles)
cycleOrders (fromCycles [[2, 3, 1], [4, 5]]) 7 ≡ [3, 2, 1, 1]
```

3.2 *getP*

Using our mini-CAS, we can also implement the *getP* function, where we want *getP n k* to equal $x_1^n + \cdots + x_k^n$. Personally, I think that this is a lot clearer than the original Mathematica code: we're creating an expression that's a *sum* of a bunch of powers of x_i to the nth power for various $i \in \{1, \ldots, k\}$ and constant n.

```
getP :: Int → Int → Expression Int
getP n k = NOp sum [BinOp (↑) (xsub i) (Constant n) | i ← [1 . . k]]
  where
    xsub i = Literal $ Var 'x' `Sub` i
```

(The ↑ operator, written ˆ, is exponentiation.)

List comprehensions
The expression in brackets is a *list comprehension*. Here are two simple examples:

```
firstFiveSquares = [x ∗ x | x ← [1 . . 5]]
trianglePoints   = [(x, y) | x ← [1 . . 10], y ← [1 . . x]]
```

The format is made to resemble set-builder notation.

3.3 *powerSym*

We want to define

$$\mathit{powerSym}(\lambda, k) = \prod_{\lambda_i \in \lambda} \mathit{getP}(\lambda_i, k),$$

so we do just that!

```
powerSym :: [Int] → Int → Expression Int
powerSym lambda k = NOp product [getP li k | li ← lambda]
```

3.4 *getZG*

It looks like this one is

$$\mathit{getZG}(G, n, k) = \frac{1}{|G|} \sum_{\sigma \in G} \mathit{powerSym}(\mathit{cycleOrders}(\sigma, n), k);$$

even though I’m not entirely sure what that represents, it’s not too difficult to write.

```
getZG grp n k = MonOp (‘div‘length elements) $ NOp sum terms
  where
    elements     = elts grp
    terms        = map term elements
    term element = powerSym (cycleOrders element n) k
```

And that’s it—we’re done with these functions! Finally, we can write the main program to solve the original problem.

4 The main program

First, we import the things we've created so far, as well as some standard libraries.

```
module Main where
import Liese
import CAS
import Math.Algebra.Group.PermutationGroup
  (_S, Permutation, (.^), fromList, elts)
import Data.List (elemIndices)
```

Note that *_S* generates the symmetric group of a given order, and (.^) applies a permutation to an element (like *x .^ perm*).

> **On capitalization**
> Capitalization is semantically important in Haskell: only data and type constructors may start with an uppercase letter. (The compiler enforces this.) Because *_S* is not a data constructor (it's not a variant of any **data** type; it's just a convenient function), this library prefixes it with an underscore to avoid this constraint.

Also, *elemIndices* :: *a* → [*a*] → [*Int*]; combined with its name, that type signature should be sufficient for you to determine what it does.

We'll start by defining a few type aliases for our own use, since we talk about edges on the graph a lot.

```
type Edge = (Int, Int)
type Edges = [Edge]
```

This is the first difference from the Mathematica implementation: we explicitly use *pairs* of integers to represent edges instead of an arbitrary array of integers [*Int*]. This way, the type system ensures that values like (3) and (5, 6, 7) cannot possibly be called edges.

Next, this particular problem is all about S_7, but we might as well make that arbitrary.

```
groupNumber :: Int
groupNumber = 7
```

The Mathematica code used a `Subsets` function to generate the sorted edges. I think it's simpler, though, to just use a two-dimensional list comprehension. This also avoids confusion due to the fact that we want the edges to be sorted, but the name sub*sets* seems to imply that that need not be the case (even though it probably is).

```
sortedEdges :: Edges
sortedEdges = [(a, b) | a ← [1 .. groupNumber], b ← [a + 1 .. groupNumber]]
```

We can use the built-in _S function to get the generating elements for the symmetric group of our order.

```
group :: [Permutation Int]
group = _S groupNumber
```

Next comes the *applyPermutationToEdges* function. This is actually built in to the `PermutationGroup` module as the (−ˆ) operator. However, we implement it just to show that there's no magic going on.

```
applyPermutationToEdges :: Permutation Int → Edges
applyPermutationToEdges perm = map applyEdge sortedEdges
  where
    applyEdge (a, b) =
      let (a', b') = (a .ˆ perm, b .ˆ perm)
      in  if a' < b'
            then (a', b')
            else (b', a')
```

Destructuring

Note that *applyPermutationToEdges* is a function of just one argument, but we're defining it by writing *applyEdge* $(a, b) =$ .... Here, the (a, b) refers not to two different variables, but to one variable; we're expecting that this variable is a two-element tuple and extracting its elements as a and b. That is, we could equivalently have written

```
applyPermutationToEdges :: Permutation Int → Edges
applyPermutationToEdges perm = map applyEdge sortedEdges
  where
    applyEdge edge =
      let a = fst edge
          b = snd edge
      in  if a' < b'
            then (a', b')
            else (b', a')
```

where *fst* and *snd* are built-in functions that get the first and second elements of a pair.

This technique (the shorter version) is called *destructuring*. It's a special case of *pattern matching*, which is very powerful.

If we wanted this to be a bit more general, we could write

```
import Data.List (sort)
applyPermutationToEdges :: Permutation Int → Edges
applyPermutationToEdges perm = map applyEdge sortedEdges
```

```
    where
      applyEdge (a, b) =
        let [a', b'] = sort [a .^ perm, b .^ perm]
        in  (a', b')
```

to allow this to be more easily extended to arbitrary “edge-like” things. But we won’t do that because we don’t care about that case.

Then, finally, we have *getPermutationFromEdgeList*:

```
getPermutationFromEdgeList :: Permutation Int → Edges → Permutation Int
getPermutationFromEdgeList perm edges =
  let permutedEdges      = applyPermutationToEdges perm
      findPositions edge = edge `elemIndices` edges
  in  fromList $ concatMap findPositions permutedEdges
```

Note that *concatMap* = *concat* ∘ *map* :: $(a \rightarrow [b]) \rightarrow [a] \rightarrow [b]$.

> Stop and think about that type signature. It tells you a lot. Can you tell exactly what it will do just from the type? Can you implement it?

For the finale, we create our group and get the results!

```
g = [getPermutationFromEdgeList perm sortedEdges | perm ← elts group]
main = do
  print $ length $ elts g            -- 5040
  let zgExpression = getZG g 21 2
  let env = const (Just 1)           -- let x_i = 1   ∀i
  print $ eval env zgExpression      -- Right 1044
```

A Monads

Okay, here we go. This will not be comprehensive, but perhaps it will be useful.

I have to introduce a new colored box for this section:

> **Exercise.** This is an exercise. Do it before proceeding.

(The answers will often immediately follow the exercises.)

The core unit of computation is the function. But, to some degree, functions represent an idealized view of the world. In many cases, we want our computations to have some more structure. For example, the following facts are all true:

- Some computations may sometimes fail to return a value; e.g., looking for an item in a list if it's not actually there.
- Some computations may sometimes return a helpful error message instead of the expected result; e.g., parsing a file that may be corrupt.
- Some computations may be nondeterministic; e.g., rolling a die.
- Some computations may be nondeterministic and with non-uniform probability; e.g., rolling two dice and computing their sum.
- Some computations may require reading from some global configuration file that would be awkward or annoying to pass to every function; e.g., running a physics simulation with lots of parameters.
- Some computations may want to perform I/O; e.g., writing to a file or getting user input.
- Some computations may want to update some persistent internal state; e.g., evaluating a computer program in a language that allows mutation of variables.

Monads are[1] an abstraction over the notion of computation itself, and provide a way to easily express and compose computations that have structures like these and others.

A.1 A motivating example: genealogy

Suppose we have a family tree. In a family tree, each person has two biological parents: a mother and a father.

However, this being the Real World, we may sometimes lack genealogical data for one or more of the parents. Suppose we have functions like this:

$$\begin{aligned}&\mathit{getMother} :: \mathit{Person} \to \mathit{Maybe\ Person}\\&\mathit{getFather} :: \mathit{Person} \to \mathit{Maybe\ Person}\end{aligned}$$

[1] Well, this is one way to view them, anyway, and it works for right now.

Recall that the *Maybe* type indicates a possible absence of a value. So maybe *getMother isaac* ≡ *Just sarah*, but *getFather isaac* ≡ *Nothing* don't know who Isaac's father was.

Now suppose we want to create the function *getPaternalGrandfather* :: *Person* → *Maybe Person*. If we didn't have all this *Maybe* business, we could just write *getPaternalGrandfather* = *getFather* ∘ *getFather*. But instead we have to write

```
getPaternalGrandfather person =
  let maybeFather = getFather person in
    case maybeFather of
      Just father → getFather father
      Nothing     → Nothing
```

Exercise. What if we wanted to find the father's father's father—that is, the paternal great-grandfather? Can you write *getPPGreatgrandfather* :: *Person* → *Maybe Person*?

We might write

```
getPPGreatgrandfather person =
  let maybeFather = getFather person in
    case maybeFather of
      Just father →
        let maybeGrandfather = getFather father in
          case maybeGrandfather of
            Just grandfather → getFather grandfather
            Nothing          → Nothing
      Nothing     → Nothing
```

Clearly, this gets worse when we want to get, say, a great^n-grandmother. This is hardly elegant.

We'll try to come up with a clean solution for this and similar problems.

A.1.1 Digression: *fmap*

We'll start by noting the following: For any type a, there is a type *Maybe* a, and for any function $f :: a \to b$, there's a function that *fmap* f :: *Maybe* a → *Maybe* b. In particular,

```
fmap :: (a → b) → (Maybe a → Maybe b)
fmap f Nothing = Nothing
fmap f (Just x) = Just (f x)
```

For example:

```
double      = (∗) 2
doubleMaybe = fmap double
```

$$\begin{aligned} &\mathit{doubleMaybe}\ (\mathit{Just}\ 10) \equiv \mathit{Just}\ 20 \\ &\mathit{doubleMaybe}\ \mathit{Nothing}\ \ \equiv \mathit{Nothing} \end{aligned}$$

As you can see, the *fmap* function lets us use functions *inside* a *Maybe*.

A.1.2 A helper function: *join*

Let's write a function:

$$\begin{aligned} &\mathit{join} :: \mathit{Maybe}\ (\mathit{Maybe}\ a) \to \mathit{Maybe}\ a \\ &\mathit{join}\ (\mathit{Just}\ (\mathit{Just}\ x)) = \mathit{Just}\ x \\ &\mathit{join}\ (\mathit{Just}\ \mathit{Nothing}) = \mathit{Nothing} \\ &\mathit{join}\ \mathit{Nothing} \qquad\qquad = \mathit{Nothing} \end{aligned}$$

From this, we can see that $\mathit{join}\ (\mathit{Just}\ (\mathit{Just}\ 10)) = \mathit{Just}\ 10$ and the *join* of anything with a *Nothing* anywhere is *Nothing*. In a sense, *join* "squashes out" one level of *Just*s and only gets the value at the very end.

Why is this useful? Well, it may not look like it, but we now have all the tools to construct *getPaternalGrandfather*!

A.1.3 Take two

We can now do two things:

- We can take a normal function and slap a *Maybe* onto its domain and codomain using *fmap*.
- We can squash a $\mathit{Just}\ (\mathit{Just}\ a)$ into a $\mathit{Just}\ a$ using *join*.

Exercise. Can you use this to write *getPaternalGrandfather*? Think about the types: what do you have? what do you want? what can you do?

(If this sounds remarkably like a proof strategy, by the way, then you'll love the Curry–Howard correspondence!)

Well, $\mathit{getFather} :: \mathit{Person} \to \mathit{Maybe}\ \mathit{Person}$...
...so $\mathit{fmap}\ \mathit{getFather} :: \mathit{Maybe}\ \mathit{Person} \to \mathit{Maybe}\ (\mathit{Maybe}\ \mathit{Person})$...
...so $\mathit{fmap}\ \mathit{getFather} \circ \mathit{getFather} :: \mathit{Person} \to \mathit{Maybe}\ (\mathit{Maybe}\ \mathit{Person})$...
...so $\mathit{join} \circ \mathit{fmap}\ \mathit{getFather} \circ \mathit{getFather} :: \mathit{Person} \to \mathit{Maybe}\ \mathit{Person}$...
and all of a sudden, we have a function of type $\mathit{Person} \to \mathit{Maybe}\ \mathit{Person}$ that suspiciously contains two uses of *getFather*.

Is this function *getPaternalGrandfather*? You bet!

You're supposed to pause and think between paragraphs, by the way. Go back and read that again.

A.1.4 A thousand words

Maybe this helps:

$$\begin{tikzcd}
Person \arrow[r, "getFather"] & Maybe\ Person \arrow[r, bend left, "fmap\ getFather"] & Maybe\ Maybe\ Person \arrow[l, bend left, "join"]
\end{tikzcd}$$

We can iterate that cycle as many times as we like. If we ever hit a *Nothing*, the *join* will make sure that we keep getting *Nothing*s.

Note that this diagram is most certainly *not* commutative! If it were, then we would need *join* ∘ *fmap getFather* (or at its least restriction to the range of *getFather*) to be the identity map, which would render it rather useless.

What just happened?

Was that "monads"? Well, kind of. The *Maybe* type constructor is *a* monad. The real power of monads is that there are many things that behave similarly. Let's look at another one!

A.2 Another example: nondeterminism

Suppose we want to model computations that could return zero or more values instead of just one. Suppose further that we don't care about the probabilities, or we assume they're all uniform; we just want to know which are possible.

How might we model this? Well, we can just return a list of all the possibilities!

```
rollDie :: [Int]
rollDie = [1 .. 6]
```

We're now treating lists as models of nondeterministic computations. You probably want to see a function, so here you go:

```
    -- rollDn: e.g., roll a D20
rollDn :: Int → [Int]
rollDn n = [1 .. n]
```

We've seen that all the hard stuff happens when we try to compose these things. So let's do that. Suppose we have all the dice from D1 to Dn. Let's roll a Dn, and then roll another die of the number we just got. So if we get a 14 when we roll the Dn, the next die will roll will be the Dn.

Of course, by "roll," we mean "define a computation of nondeterministic result," where the "result" we care about is, say, the value of the last die, with repetition (so if there are k ways to get it, we include it k times).

A.2.1 The bad way

As before, we'll start with the brute-force way:

```
    -- Given a maximum die number n, roll two dice as described above.
rollTwice :: Int → [Int]
rollTwice n =
    -- Find all the values we could get for the second die...
  let secondDieTypes  = rollDn n
    -- ...then, for each one of them, roll that die...
      secondDieValues = map rollDn secondDieTypes
    -- ...and, finally, flatten the results.
  in  concat secondDieValues
```

Now, though, what if we wanted to add one more iteration?

```
    -- Given a maximum die number n, roll three dice like described above.
rollThrice :: Int → [Int]
rollThrice n =
  let secondDieTypes  = rollDn n
      secondDieValues = map rollDn secondDieTypes
      flatSecondDies  = concat flatSecondDies
      thirdDieTypes   = flatSecondDies
      thirdDieValues  = map rollDn thirdDieTypes
  in  concat thirdDieValues
```

To be honest, this isn't *awful*—at least it's not nested like the *Maybe* example was—but it sure isn't great. For example, isn't obvious how to define *rollKTimes k n*.

Hey, here's an idea—*fmap* and *join* were useful with *Maybe*. How about with lists?

A.2.2 *fmap* for lists

When we introduced *fmap*, we said that it let us "use functions inside a *Maybe*." That is, we had *fmap* :: (*a* → *b*) → (*Maybe a* → *Maybe b*).

Now we have lists, so we want to "use functions inside a list," and we probably want *fmap* :: (*a* → *b*) → ([*a*] → [*b*]).

But we already have that! In fact, we've just used it! It's *map*!

```
fmap :: (a → b) → ([a] → [b])
fmap = map
```

A.2.3 *join* for lists

When we introduced *join*, we said that it let us "squash out" one level of *Just*s, and we had *join* :: *Maybe* (*Maybe a*) → *Maybe a*.

So we probably want *join* :: [[*a*]] → [*a*], and we want to squash out a level of lists without losing any information (e.g., we don't want to just say "always take the first list and discard the rest").

Again, this function already exists, and we've already used it! This time, it's *concat*, which flattens one level of lists!

```
join :: [[a]] → [a]
join = concat
```

A.2.4 Rewriting with *fmap* and *join*

Let's try it again.

```
rollTwice :: Int → [Int]
rollTwice n = join $ fmap rollDn (rollDn n)


rollThrice :: Int → [Int]
rollThrice n = join $ fmap rollDn $ join $ fmap rollDn (rollDn n)
```

Well, at least it looks a bit nicer. That last line—with the repeated *join* $ *fmap rollDn*— is just *begging* to be improved further, and indeed we will. But we'll need a few more tools to do that.

Here's that diagram again:

$$\begin{tikzcd} Int \arrow[r, "rollDn"] & {[Int]} \arrow[r, bend left, "fmap\ rollDn"] & {[[Int]]} \arrow[l, bend left, "join"] \end{tikzcd}$$

A.3 A new friend: *bind*, or (≫=)

I'm about to introduce something "new" that's not really new at all. Here it goes:

```
bind x f = join $ fmap f x
x ≫= f   = bind x f   -- or just (≫=) = bind
```

Do you see that this just gives a name to the pattern we've been using?

```
getPaternalGrandfather p = getFather p ≫= getFather
getPPGreatgrandfather p = getFather p ≫= getFather ≫= getFather
rollTwice n = rollDn n ≫= rollDn
rollThrice n = rollDn n ≫= rollDn ≫= rollDn
```

A.3.1 What does *bind* do?

Even though we have the definition above, *bind* is not very helpful to us if we had no intuition. Let's add it to our catalog of monadic functions:

- *join* squashes one level of a monad (like *Maybe* or lists);
- *fmap* pulls a function into a monad;
- (≫=) **does an action that might introduce "too much" monadness** (e.g., too many levels of *Just*s or nested lists) **and then squashes out the extra stuff,** so, as a result, you don't end up any more or less monad-y than how you started.

A.3.2 Why do we care?

At the very least, the rewritten versions of *getPPGreatgrandfather* and *rollThrice* are nicer to look at! This also lets us not have to visit that second-level list in our diagram:

$$\begin{tikzcd} Int \arrow[r, "rollDn"] & {[Int]} \arrow[loop right, "(\gg\!= rollDn)"] \end{tikzcd}$$

More importantly, it now becomes clearer how we would develop the following functions, which we spoke of earlier:

```
getNthPaternalGrandfather :: Int → Person → Maybe Person
rollKTimes :: Int → Int → [Int]
```

We can do so as follows:

```
  -- Get someone's nth pater-n-nal great^{n−2}-grandfather.
  -- That is, if n = 0, the result is Just the person;
  -- if n = 0, it is the same as the result of getFather; etc.
getNthPaternalGrandfather :: Int → Person → Maybe Person
getNthPaternalGrandfather 0 p = Just p
getNthPaternalGrandfather n p = previous ≫= getFather
  where
    previous = getNthPaternalGrandfather (n − 1) p
```

Alternatively, we could have done the following, which is perhaps more reminiscent of the diagram above:

```
getNthPaternalGrandfather′ :: Int → Person → Maybe Person
getNthPaternalGrandfather′ n p = iterate (≫=getFather) p !! n
```

Note that $iterate\ f\ x = [x, f\ x, f\ (f\ x), ...]$, and (!!) is the list index operator (e.g., ["a", "b", "c", "d", "e"] !! 3 = "d"), so this constructs an (infinite) list of all the fathers and takes the nth one.

Infinity and laziness

Infinite lists are totally kosher in Haskell, because Haskell is a *lazily evaluated language*. Basically, this means that the Haskell runtime never actually does anything until you actually need the result *right now* (e.g., to print to the screen or write to a file).

So when you write $\mathit{iterate}\ f\ x$, or even just $[1\,..]$, the runtime says, "sure, I can do that"; it's only when you actually try to use that value that anything is computed.

In this case, the list indexing operator only needs to traverse the first n elements of the list. So the runtime keeps advancing through the elements of the list, *generating them as it needs to*, until it finds the one you're looking for, at which point it stops.

(In fact, it doesn't even evaluate the other elements of the list if it doesn't have to! In this case, it does, because each element depends on the previous, but you can run, e.g., $[\mathit{error}$ "bad", "good"$]\,!!\,1$ and get "good" because the error is never encountered.)

A.3.3 What is *bind*'s type?

In the examples above, I conveniently omitted the type signature for *bind*. Is it $\mathit{bind} :: [a] \rightarrow (a \rightarrow [b]) \rightarrow [b]$, or $\mathit{bind} :: \mathit{Maybe}\ a \rightarrow (a \rightarrow \mathit{Maybe}\ b) \rightarrow \mathit{Maybe}\ b$?

The answer is—both! Let's not cover this too deeply right now, but suffice it to say that if m is a monad, then we can use $\mathit{bind} :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$. In general, we write

$$(\gg\!=) :: \mathit{Monad}\ m \Rightarrow m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$$

Examples of monads are *Maybe* and $[\cdot]$ (lists).

If this is confusing and you don't care, ignore it.

A.4 Monads in the CAS

Here's one final example before we go back to more generalized abstract nonsense. The monad we're using in our CAS—the place this all started—isn't either *Maybe* or $[\cdot]$.

The monad we're using is *Either String*. This represents a computation that might succeed, or might return an error message that's a *String*.

Monads need exactly one type parameter

Note that we said *Maybe* is a monad, but we always talk about a *Maybe Int* or a *Maybe Person*. We said that $[\cdot]$ is a monad, but we always talk about $[\mathit{Int}]$ or $[\mathit{String}]$. That's because *Maybe* and $[\cdot]$ each require one type parameter to form a concrete type that can take on values. We say that *Maybe* and $[\cdot]$ have kind $* \rightarrow *$.

By constrast, with *Either*, we talk about things like *Either String Int*. That is, *Either* is not a type, nor is *Either String*; we need *two* type parameters. So

Either has kind $* \to * \to *$.

However, just as we can partially apply functions—like $double = (*)\ 2$—we can partially apply type constructors. So $Either :: * \to * \to *$; we can apply it once to get $Either\ String :: * \to *$, and again to get $Either\ String\ Int :: *$.

Thus, *Either String* is a type constructor that requires one type parameter, so it is eligible to be a monad, if it satisfies certain properties—in fact, it does.

A.4.1 *fmap* and *join*

Exercise.

1. What should the types of *fmap* and *join* be now?
2. How should we implement *fmap* and *join* to be consistent with the intuition we've developed, while losing as little information as possible?

Did you try it? (The answers will appear on the next page.)

For the first exercise, you should probably have found the types

- $\mathit{fmap} :: (a \to b) \to (\mathit{Either}\ \mathit{String}\ a \to \mathit{Either}\ \mathit{String}\ b)$, and
- $\mathit{join} :: \mathit{Either}\ \mathit{String}\ (\mathit{Either}\ \mathit{String}\ a) \to \mathit{Either}\ \mathit{String}\ a$.

You can find this by just replacing *Maybe* with *Either String* in the types for *fmap* and *join* in the *Maybe* monad.

For the second question, *fmap* should have been rather straightforward, using pattern matching:

$$\begin{aligned}
&\mathit{fmap} :: (a \to b) \to (\mathit{Either}\ \mathit{String}\ a \to \mathit{Either}\ \mathit{String}\ b)\\
&\mathit{fmap}\ f\ (\mathit{Right}\ v) = \mathit{Right}\ (f\ v)\\
&\mathit{fmap}\ f\ \mathit{leftVal} \quad\ \ = \mathit{leftVal}
\end{aligned}$$

Variable patterns match any value

Note that the *leftVal* in the *fmap f leftVal* pattern isn't directly related at all to the *Left* variant of the type. We could have written $\mathit{fmap}\ f\ x = x$ for that branch and it would have done the same thing. That is, *leftVal* matches any value.

However, if the value were a *Right*, then it would already have matched on the previous pattern. This, combined with the fact that we know the *Either* data type has exactly two variants, proves that *leftVal* really must be a *Left*.

We could also have written $\mathit{fmap}\ f\ (\mathit{Left}\ e) = \mathit{Left}\ e$, but there's no need to destructure and then restructure the value.

For *join*, you could have explicitly covered all three cases:

$$\begin{aligned}
&\mathit{join} :: \mathit{Either}\ \mathit{String}\ (\mathit{Either}\ \mathit{String}\ a) \to \mathit{Either}\ \mathit{String}\ a\\
&\mathit{join}\ (\mathit{Right}\ (\mathit{Right}\ v)) = \mathit{Right}\ v\\
&\mathit{join}\ (\mathit{Right}\ (\mathit{Left}\ e)) \ \ = \mathit{Left}\ e\\
&\mathit{join}\ (\mathit{Left}\ e) \qquad\qquad\ \, = \mathit{Left}\ e
\end{aligned}$$

...or you could have consolidated that a bit:

$$\begin{aligned}
&\mathit{join} :: \mathit{Either}\ \mathit{String}\ (\mathit{Either}\ \mathit{String}\ a) \to \mathit{Either}\ \mathit{String}\ a\\
&\mathit{join}\ (\mathit{Right}\ \mathit{either}) = \mathit{either}\\
&\mathit{join}\ (\mathit{Left}\ e) \qquad\ \ = \mathit{Left}\ e
\end{aligned}$$

Values can't change types

You may be wondering why I wrote $\mathit{join}\ (\mathit{Left}\ e) = \mathit{Left}\ e$ in the last line when I had just said that that's not necessary! The issue is as follows.

Suppose we had written $\mathit{join}\ \mathit{leftVal} = \mathit{leftVal}$. Then we know that *leftVal* must be an *Either String* (*Either String a*). But the result of the function must be just an *Either String a*, of which *leftVal* is not a member.

In other words, the *Left* in the (*Left e*) in the pattern refers to the data

> constructor for the *Either String* (*Either String a*) type, but the *Left* on the right-hand side is the data constructor for the *Either String a* type. We don't have a value of type *Either String a*, so we need to take the *String* from the *Either String* (*Either String a*) and use that to build our result.

A.5 The missing piece: *unit*, or *return*

There's one more factor common to all monads (indeed, by requirement). I've postponed it because it's not particularly interesting, but we'll need it to proceed further, so here goes.

It's called *unit*. Here's how it's implemented for our three example monads:

```
unit :: a → Maybe a
unit x = Just x
  -- or just unit = Just

unit :: a → [a]
unit x = [x]

unit :: a → Either String a
unit x = Right x
  -- or just unit = Right
```

Do you see what it does? It takes a normal value and wraps it up in a monad in the simplest way possible.

That's it.

The only sticky point is in the name. The Haskell standard library calls it *return*, which I think, because of its connotations, is useful in practice but awful for learning. Let me strongly note the following: **the function *return* is not a control statement** like the `return` you may have seen in (many) other programming languages. It is precisely the function *unit* defined above.

The categorists call it η, which is fine with me.

So now we have the following things that we can use in a monad:

- *return* pulls a normal value up into a monad;
- *join* squashes one level of a monad;
- *fmap* pulls a function into a monad;
- (≫=) does an action that might introduce "too much" monad-ness (e.g., too many levels of *Just*s or nested lists) and then squashes out the extra stuff, so, as a result, you don't end up any more or less monad-y than how you started.

Let's visit a diagram in, say, the *Maybe* monad, supposing we already have a function $f :: a \to b$:

$$\begin{tikzcd}
a \arrow[r, "return"] \arrow[d, "f"] & \mathit{Maybe}\ a \arrow[r, shift left, "return"] \arrow[d, "\mathit{fmap}\ f"] & \mathit{Maybe}\ (\mathit{Maybe}\ a) \arrow[l, shift left, "join"] \arrow[d, "\mathit{fmap}\ (\mathit{fmap}\ f)"] \\
b \arrow[r, "return"] & \mathit{Maybe}\ b \arrow[r, shift left, "return"] & \mathit{Maybe}\ (\mathit{Maybe}\ b) \arrow[l, shift left, "join"]
\end{tikzcd}$$

One very important fact about this diagram is that we have a lot of freedom to move around within the monad—we can always take types to higher levels of monads with *return*, and we can squash multiple monads to fewer with *join*—but **we can never escape the monad completely!** In this case, that means that we can never take a computation that might not have a result (a *Maybe* a) and force it to come up with some type of result on the spot (an a).

Think about it—suppose I give you a value of type *Maybe* β for some β, which you know nothing about. You might even know what that type *is*: it could be defined in some library that's not accessible to you. Is it reasonable for me to ask you for some normal value of type β?

Well, if I happen to have given you a *Just* b for some $b :: \beta$, then, yeah, you could extract it and give it to me. But if I give you a *Nothing*, there's no way you could give me a value! Suppose β actually represents the type of all complete ordered fields not isomorphic to $\mathbb{R}$—you'd be hard-pressed to give me a β then! Plus, even if β were something mundane like *Int*, you couldn't just give me 0 because *you don't know* that β is *Int*; your function has to work for *all* types β.

This diagram, by the way, is a commutative diagram in **Hask**, the category of Haskell types and functions. This diagram and a few others must commute for a functor (here, *Maybe*) to be a monad. The categorists would write it like this:

$$\begin{tikzcd}
a \arrow[r, "\eta_A"] \arrow[d, "f"] & M\ a \arrow[r, shift left, "\eta_{M\ a}"] \arrow[d, "M\ f"] & M\ (M\ a) \arrow[l, shift left, "\mu_a"] \arrow[d, "M\ (M\ f)"] \\
b \arrow[r, "\eta_b"] & M\ b \arrow[r, shift left, "\eta_{M\ b}"] & M\ (M\ b) \arrow[l, shift left, "\mu_b"]
\end{tikzcd}$$

Note that M is a functor (here, M is *Maybe*), and $\eta : 1_{\mathbf{Hask}} \to M$ and $\mu : M^2 \to M$ are natural transformations.

A.6 The real power: general monadic functions

So far, we've looked at three monads in isolation: *Maybe*, $[\cdot]$, and *Either* e (in particular, *Either String*). But, as nice as it is to notice similarities among these type constructors, we haven't yet really taken advantage of those similarities by constructing machinery that can operate on any of them!

Exercise. Actually, that's false. If we take the fundamental units of a

> monad to be *fmap*, *return*, and *join*, what function have we implemented just in terms of those that would work in any monad?

The answer is *bind*—that's why I said it really wasn't anything new, because we can construct it entirely from existing pieces.

> **Exercise.** In actuality, though, Haskell doesn't take *join* to be a fundamental unit; instead, a monad must provide *fmap*, *return*, and *bind*. Can you implement *join* in terms of *bind*, or is there some information lost in the way we approached it (with *join*)?

Well, think about the following definition:

$$\begin{aligned}&\mathit{join} :: \mathit{Monad}\ m \Rightarrow m\ (m\ a) \rightarrow m\ a\\&\mathit{join}\ x = x \ggg\!= \mathit{id}\end{aligned}$$

(Note that $\mathit{id} :: a \rightarrow a$ is the identity function.)

What does this mean? Recall our intuition for *bind* as doing something that introduces "too much" monad-ness and then stripping off the extra stuff. Here, we're saying that we *already have* too much monad-ness, so we just do nothing to the input and then strip off the extra stuff, which is precisely what *join* does.

But we've already talked about *bind*. Let's talk about another.

Let me pose two problems.

- Suppose we have an environment of variable bindings (the *Environment* type from the *CAS* module), and we have a list of *Symbol*s. We want to get all the variables' values, or fail if any of them doesn't exist. So, for example, if we have bindings for x and y, then [*Var* `'x'`, *Var* `'y'`] might map to *Just* $[5, 10]$, but [*Var* `'x'`, *Var* `'z'`] would map to *Nothing* (not [*Just* 5, *Nothing*]).
- Suppose we have a list of symbolic *Expression*s and we want to evaluate them all and collect the results in a list, but if any of them returns an error we should stop and return that error instead. For example, [*exprA*, *exprB*, *exprC*] might map to *Right* [*valA*, *valB*, *valC*], but [*exprA*, *badExpr*, *exprC*] might map to *Left* `"bad expression"`.

Can you see that these problems are similar? They could each be described by the following process:

> Suppose we have a bunch of values of type a; that is, we have a list of type $[a]$. Suppose also that we have a function $f :: a \rightarrow m\ b$ that takes an a and creates a computation that returns a b. I'd like to create a new computation that passes each of my values through this function, evaluates them in sequence, and gathers up the results in a (now monadic) list; that is, my computation should have type $m\ [b]$.

Could we write such a function that would work for *Maybe*s, for lists, and for *Either String*s? That's our goal for this subsection.

Before we proceed, however, does this sound familiar? Can you think of a similar, non-monadic function?

Exercise. Does it? Can you?

The answer to the first part, of course, is, "yes"! We should be quite familiar with a function that does the following!

> Suppose we have a bunch of values of type a; that is, we have a list of type $[a]$. Suppose also that we have a function $f :: a \to b$ that takes an a and returns a b. I'd like to pass each of my values through this function and gather up the results in a list; that is, my result should have type $[b]$.

This function's name, of course, is *map*, and its type is $map :: (a \to b) \to [a] \to [b]$.

We thus want a monadic version of *map*; we'll call it *mapM*. Comparing the two descriptions above, we can see the evident type adjustments to find that we're looking for $mapM :: Monad\ a \Rightarrow (a \to m\ b) \to [a] \to m\ [b]$.

Exercise. In practice, is this different from just *map*? How?

It sure is. If $f :: a \to m\ b$ and $as :: [a]$, then $map\ f\ as :: [m\ b]$, while $mapM\ f\ as :: m\ [b]$. These can be very different. In the former case, any of the computations can fail and we'll still get the results from the rest; e.g., we could get

$[Just\ 3, Nothing, Just\ 7]$,

or

$[Right$ `"baboon"`, $Left$ `"animal escaped"`, $Right$ `"gorilla"`$]$.

Let's implement *mapM* together. What happens if the list we're given is empty? Well, then the computation should always succeed, and it should monadically contain the empty list—e.g., we might want $Just\ []$ or $Right\ []$, because it succeeded but there was nothing to act on. The generic term for "succeed" is *return*. So

$$mapM\ \_\ [] = return\ []$$

(We ignore the function by matching it against an underscore.)

Now the recursive case. Suppose we get a list that's not empty. Then it has a head (first element) and a tail (other elements). Call the list $x : xs$ (where x is the head and xs is the tail).

What do we want? An $m\ [b]$. What do we have? We have $x :: m\ a$, $xs :: [m\ a]$, and $f :: a \to m\ b$. But we also have ourselves—$mapM :: Monad\ a \Rightarrow (a \to m\ b) \to [a] \to m\ [b]$. So we can recursively call ourselves with xs to get an $m\ [b]$.

> As with all recursively defined functions, the proof of termination relies upon the fact that the list is getting smaller at each step; with each recursive call, we're making progress toward the base case of the empty list.

Okay, so $mapM\ f\ xs :: m\ [b]$. We can also clearly obtain $f\ x :: m\ b$. Now we can do the following:

```
mapM f (x : xs) =
  x ≫= λvx →
    mapM f xs ≫= λvxs →
      return (f vx : vxs)
```

Let's tease this apart with a type analysis.

- Consider the outer binding (the one with x on the left-hand side, which starts on the second line of the definition). Its type must be $(\gg\!=) :: m\ \alpha_1 \to (\alpha_1 \to m\ \beta_1) \to m\ \beta_1$ for some values of α_1 and β_1.

  We know that the result of $mapM\ f\ (x : xs)$ must be an $m\ [b]$, so we must have $\beta_1 = [b]$.

  We also know that $x :: m\ a$, so we must have $\alpha_1 = a$.

  Thus, the lambda expression on the right-hand side must have type $a \to m\ [b]$.

- Because the lambda expression has type $a \to m\ [b]$, we know that vx must be of type a, and the return value of the lambda must be of type $m\ [b]$.

- Now we have another binding; call its type $(\gg\!=) :: m\ \alpha_2 \to (\alpha_2 \to m\ \beta_2) \to m\ \beta_2$. Because the binding is the return value of the enclosing lambda, which we know must have type $m\ [b]$, we must have $m\ \beta_2 = m\ [b]$, so $\beta_2 = [b]$.

  The left-hand side of the binding is the result of a recursive call to $mapM$. This recursive call will have return type $m\ [b]$, so we must have $m\ \alpha_2 = m\ [b]$. Thus, $\alpha_2 = [b]$.

  The right-hand side of the lambda must have type $\alpha_2 \to m\ \beta_2$. We know this is $[b] \to m\ [b]$. Thus, its parameter must be $vxs :: [b]$.

- Thus, in the innermost lambda expression (on the last line), we have access to $vx :: a$ and $vxs :: [b]$. The expression $f\ vx$ thus has type b, so we can write $(f\ vx : vxs) :: [b]$. We're almost done.

- The return value of the final lambda must be of type $m\ [b]$, and we just created a $[b]$. We bring this up into the monad by calling $return$.

Here's another use case for $mapM$. Suppose, hypothetically, that we wanted to download a hundred books from Springer while they're free for a limited time.

Suppose we have the URLs for all the books, so $books :: [URL]$. Suppose we also have a function $download :: URL \to IO\ Filename$ that downloads books to a file.

> **The *IO* type**
> This type means that *download* performs an I/O action and then returns the name of the file where the book is stored. For example, the action of getting user input has type *IO String*, because you do some I/O and then get a string.
>
> The *download* I/O action, presumably, downloads the contents of the given URL and saves them to disk, then returns the path to the file on disk.

Then we'd like to download all the books and get their URLs. Here's how we do it:

```
downloadBooks :: IO [FilePath]
downloadBooks = mapM download books
```

> **I/O actions are "sticky"**
> Note that *downloadBooks* is still an *IO* action, because we can't get out of the *IO* monad! When the program is actually run, the Haskell runtime takes case of actually performing the actions; due to laziness, we don't (and can't) do this ourselves.[2]

A.7 The do-notation

Okay: I'm about to introduce something that's very practically useful, but only because it hides the details of a lot of what we've been talking about. It's a form of *syntactic sugar*—that is, a nice syntax that's converted by the compiler to something simpler.

It's called **do**-notation, and even this name is potentially confusing: **it doesn't "do" anything**. In particular, it doesn't perform any I/O actions, it doesn't prevent laziness, and it doesn't change the execution order (these are all common misconceptions).

Let me reiterate: **do**-notation is **exactly the same** as what we've been doing, but it looks nicer.

Here's some nonsense code:

```
longFormExample :: Foo → Maybe Bar
longFormExample foo =
  flux foo ≫= λbar →
    baz foo (quux bar) ≫= λzing →
      return (zing bar)
```

And here's what it looks like in **do**-notation:

```
shortFormExample :: Foo → Maybe Bar
shortFormExample = do
```

[2] Yes, *unsafePerformIO* :: *IO a* → *a* exists. As its name implies, you should think twice before using it.

```
  bar ← flux foo
  zing ← baz foo (quux bar)
  return (zing bar)
```

That is, bindings can be made with ← instead of with (≫=), and remain in scope for the rest of the **do**-block. The last line of the **do**-block is the result.

The evident advantage is that the **do**-notation removes nesting.

Here's our *mapM*:

```
mapM :: Monad m ⇒ (a → m b) → [a] → m [b]
mapM _ [] = return []
mapM f (x : xs) = do
  vx ← x
  vxs ← mapM f xs
  return (f vx : vxs)
```

Exercise. We wrote *join* in terms of *bind* and vice versa earlier. Can we rewrite these in **do**-notation? Do so.

Here's *join* in terms of *bind*:

```
join :: Monad m ⇒ m (m a) → m a
join vx = do
  x ← x
  vx
```

Exercise. Hold on—where's the *bind*?

It's hidden! That's the point of **do**-notation: the bindings are implicit.

A consequence of this is that, yes, we can write *bind* in terms of *join* using **do**-notation, but it won't actually *use* the **do**-notation:

```
bind :: Monad m ⇒ m a → (a → m b) → m b
bind x f = do
  join (fmap f x)
```

A.8 A library of monads

At the very beginning of this section, we mentioned some types of computations. Now we'll list the corresponding monad types.

- Some computations may sometimes fail to return a value; e.g., looking for an item in a list if it's not actually there.

  These computations can be normal functions that return a *Maybe a*. In *Maybe*, *join* takes a *Just* (*Just x*) to *Just x* and anything else to *Nothing*, and *return* takes *x* to *Just x*.

The meaning of *fmap* is: “if the computation succeeded, feed its output into this next one.”

- Some computations may sometimes return a helpful error message instead of the expected result; e.g., parsing a file that may be corrupt.

  A computation that might return an error of type e can be a normal function that returns an $\mathit{Either}\ e\ a$. In $\mathit{Either}\ e$, *join* takes $\mathit{Right}\ x$ to x, and $\mathit{Left}\ e$ to $\mathit{Left}\ e$; *return* takes x to $\mathit{Right}\ x$.

  The meaning of *fmap* is: “if the computation succeeded, feed its output into this next one; otherwise, keep the error around.”

- Some computations may be nondeterministic; e.g., rolling a die.

  These computations can be normal functions that return a $[a]$. In $[\cdot]$, *join* is *concat* (it flattens $[[1,2],[3,4]]$ to $[1,2,3,4]$), and *return* takes x to $[x]$.

  The meaning of *fmap* is: “take all the possible results of the computation and apply this next one to each of them.”

- Some computations may be nondeterministic and with non-uniform probability; e.g., rolling two dice and computing their sum.

  These computations can be normal functions that return a $\mathit{Dist}\ a$, where *Dist* is the distribution monad. It’s basically like a list where each element has a probability. In *Dist*, *join* computes conditional probabilities by multiplication, and *return* takes x to the constant distribution that returns x with probability 1.

  (Speaking of “probability 1”—this monad doesn’t handle cases with “almost never” or “almost surely” as nicely as perhaps you’d like it to. It’ll usually work, sure, but if you run into a case where there are infinitely many probability-0 elements of the distribution, it won’t be able to discard those from the output set, and some computations might run forever even when you think they should terminate. Whether this is desirable or “correct” is up for debate.)

  The meaning of *fmap* is: “transform all the possible values of my distribution in this way, without affecting the probabilities.”

- Some computations may require reading from some global configuration file that would be awkward or annoying to pass to every function; e.g., running a physics simulation with lots of parameters.

  These computations themselves are embodied by the $\mathit{Reader}\ r$ monad. For example, if *gravityDirection* needs to read a *Settings* object, we might have $\mathit{gravityDirection} :: \mathit{Reader}\ \mathit{Settings}\ \mathit{Vector2D}$. In $\mathit{Reader}\ r$, *join* takes a computation that produces a $\mathit{Reader}\ r\ a$ and just produces the result itself (its type is $\mathit{Reader}\ r\ (\mathit{Reader}\ r\ a) \to \mathit{Reader}\ r\ a$), and *return* takes x to the computation that always returns x without consulting the *Reader* object.

The meaning of *fmap* is: "make a new computation that first applies the old computation, which may access the *Reader* object, and then applies this 'pure,' reader-less function to the result."

- Some computations may want to perform I/O; e.g., writing to a file or getting user input.

  These computations themselves are embodied by the *IO* monad. For example, $getLine :: IO\ String$ describes the action of receiving user input; it promises to return a *String*. In *IO*, *join* takes an I/O computation that returns another *IO* action and just produces the latter result itself (its type is $IO\ (IO\ a) \to IO\ a$), and *return* takes x to the computation that always returns x without actually performing any I/O.

  The meaning of *fmap* is: "make a new I/O action that first performs the other I/O action and then applies this 'pure' function to the result."

- Some computations may want to update some persistent internal state; e.g., evaluating a computer program in a language that allows mutation of variables.

  These computations themselves are embodied by the *State s* monad. For example, if an *Environment* describes all the variables available in the current execution context of a Python program, then we might have $eval :: PyExpression \to State\ Environment\ PyValue$.

  The semantics are as with *Reader r*.

For a library of monadic functions, like *mapM*, go into GHCi and write `import Control.Monad`, then `:browse`. That'll get you a bunch.