Data Types in Haskell

Joseph C Osborn

April 28, 2025

Outline

Types

So far we've seen a few types in Haskell:

- `Int`, `Integer`, `Bool`
- `Char`
- Lists: `[Char]`, `[a]` for some a
- Tuples, e.g. `([Char], Int)` or `(a, b, c)`
- Function types: `a -> b`, `Int -> Int -> Int`

As well as some type *classes* like `Ord` or `Eq` or `Num`.

The data statement

Haskell lets you define your own types too:

```
data Suit = Hearts | Spades | Clubs | Diamonds
data Rank = A | K | Q | J | Ten | Nine | ...
data Card = Card Suit Rank
```

More Examples

```
data Nat = O | S Nat
data MyList a = MyNil | MyCons a (MyList a)

data SortedList a = Sorted [a]
sort :: (Ord a) => [a] -> Sorted [a]
```

Syntax

- `data TypeName (TypeParameters...)  =`
  - `Constructor (Parameters...)`
  - `| Constructor (Parameters...)`

Exercise

1. Define a datatype for days of the week.
2. Define a datatype for class meeting times (e.g. CSCI 054 meets on Mondays and Wednesdays from hour 13, minute 15 until hour 14, minute 30).
3. Define a datatype for a class, including its name and meeting times.

Matching

Remember from earlier in the course: construction is dual to matching!

```
is_face_card (Card _ K) = True
is_face_card (Card _ Q) = True
is_face_card (Card _ J) = True
is_face_card _ = False
```

Matching

```
insert x (Sorted []) = Sorted [x]
insert x (Sorted (y:l))
  | x <= y = Sorted (x:y:l)
  | otherwise = let Sorted s = insert x (Sorted l) in Sortec
```

This tells the caller—if you assume insert is correct, and you have evidence `l` is sorted, the output will also give you evidence `l` is sorted.

Exercise

Write a function using your class datatype from the earlier exercise to calculate how many minutes of time you meet for a class during the week.
You can write a helper function if you want.

```
contact_minutes :: Class -> Int
```

Maybe

Remember this function?

```
list_min [] = error "oh no"
list_min [x] = x
list_min (x:xs) = min x (list_min xs)
```

It's very unsatisfying that it crashes on an empty list.

Maybe

I prefer this one:

```
list_min :: (Ord a) => [a] -> Maybe a
list_min [] = Nothing
list_min (x:xs) =
  case list_min xs of
    Nothing -> Just x
    Just rest -> min x rest
```

Haskell's `Maybe a` type has two variants: `Nothing` and `Just a`. (Some programming language libraries call this type `Option`, with variants None and Some.)

Maybe

Our new list_min can never throw an error, no matter what list we give it.
But the tradeoff is that now we need to write a case around the call to list_min:

```
case list_min l of
  Nothing -> "Empty list"
  Just x -> "Smallest element found"
```

Still, it's much better to have to write error handling code than to maybe crash sometimes.

Maybe

If we were very sure the list weren't empty, we could define:

```
unwrap :: Maybe a -> a
unwrap (Just x) = x
unwrap Nothing = error "empty Maybe!"

(unwrap (list_min [1,2,3])) * 2
```

Maybe

But we can also do work on the Maybe, to avoid excessive unwrapping:

```
fmap (\x -> x * 2) (list_min l)
```

This will return double the minimum element of `l` if it exists (as `Just 14` or whatever), or else `Nothing` if there was no minimum element.
It's a lot like `map` for lists, but for arbitrary mappable types.

find_{by}

Implement `find_by :: (a -> Bool) -> [a] -> Maybe a`, which returns the first a in the list satisfying the test. It's ok to use either `filter` or recursion.

Either

In case a computation might fail in more than one way, Haskell also provides `Either`:

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

For example, reading a file from disk might give `Left String` or `Right FileNotFound` or `Right FileAccessForbidden` error or `Right FileTooBig`, from a type like:

```
data FileReadError = FileNotFound |
FileAccessForbidden | FileTooBig | ...
```

Some languages call this type Result with variants Ok, Err.

$\text{find}_{\text{unique by}}$

Given the error type `data FindError a = EmptyList | NotUnique([a])`, define `find_unique_by :: (a -> Bool) -> [a] -> Either a (FindError a)`, which returns one of:

1. The first found element in the list, if it's the only one passing the test;
2. `Right EmptyList`, if the list is empty; or
3. `Right NotUnique(dupes)` if there is more than one element that passes the test.

Either

We can also define tools like `fmap` for Either, that do things like "map the Right variant to a different type", or "map the `Left a` variant to an `Either c d`, producing either `Left c` or `Right d` depending on the result", or "If this is a `Left x`, produce `Just x`, otherwise produce `Nothing`".

These *combinators* can be really useful for avoiding code with lots of cases and ifs! You can code on the error-free path for the most part using combinators, and handle errors at the end of the chain.