Chapter 07: Hindley-Milner and Me

What's Your Type?

If you're new to the functional world, it won't be long before you find yourself knee deep in type signatures. Types are the meta language that enables people from all different backgrounds to communicate succinctly and effectively. For the most part, they're written with a system called "Hindley-Milner", which we'll be examining together in this chapter.

When working with pure functions, type signatures have an expressive power to which the English language cannot hold a candle. These signatures whisper in your ear the intimate secrets of a function. In a single, compact line, they expose behaviour and intention. We can derive "free theorems" from them. Types can be inferred so there's no need for explicit type annotations. They can be tuned to fine point precision or left general and abstract. They are not only useful for compile time checks, but also turn out to be the best possible documentation available. Type signatures thus play an important part in functional programming - much more than you might first expect.

JavaScript is a dynamic language, but that does not mean we avoid types all together. We're still working with strings, numbers, booleans, and so on. It's just that there isn't any language level integration so we hold this information in our heads. Not to worry, since we're using signatures for documentation, we can use comments to serve our purpose.

There are type checking tools available for JavaScript such as Flow or the typed dialect, TypeScript. The aim of this book is to equip one with the tools to write functional code so we'll stick with the standard type system used across FP languages.

Tales from the Cryptic

From the dusty pages of math books, across the vast sea of white papers, amongst casual Saturday morning blog posts, down into the source code itself, we find Hindley-Milner type signatures. The system is quite simple, but warrants a quick explanation and some practice to fully absorb the little language.

```
// capitalize :: String -> String
const capitalize = s => toUpperCase(head(s)) + toLowerCase(tail(s));

capitalize('smurf'); // 'Smurf'
```

Here, `capitalize` takes a `String` and returns a `String`. Never mind the implementation, it's the type signature we're interested in.

In HM, functions are written as `a -> b` where `a` and `b` are variables for any type. So the signatures for `capitalize` can be read as "a function from `String` to `String`". In other words, it takes a `String` as its input and returns a `String` as its output.

Let's look at some more function signatures:

```
// strLength :: String -> Number
const strLength = s => s.length;

// join :: String -> [String] -> String
const join = curry((what, xs) => xs.join(what));

// match :: Regex -> String -> [String]
const match = curry((reg, s) => s.match(reg));

// replace :: Regex -> String -> String -> String
const replace = curry((reg, sub, s) => s.replace(reg, sub));
```

`strLength` is the same idea as before: we take a `String` and return you a `Number`.

The others might perplex you at first glance. Without fully understanding the details, you could always just view the last type as the return value. So for `match` you can interpret as: It takes a `Regex` and a `String` and returns you `[String]`. But an interesting thing is going on here that I'd like to take a moment to explain if I may.

For `match` we are free to group the signature like so:

```
// match :: Regex -> (String -> [String])
const match = curry((reg, s) => s.match(reg));
```

Ah yes, grouping the last part in parenthesis reveals more information. Now it is seen as a function that takes a `Regex` and returns us a function from `String` to `[String]`. Because of currying, this is indeed the case: give it a `Regex` and we get a function back waiting for its `String` argument. Of course, we don't have to think of it this way, but it is good to understand why the last type is the one returned.

```
// match :: Regex -> (String -> [String])
// onHoliday :: String -> [String]
const onHoliday = match(/holiday/ig);
```

Each argument pops one type off the front of the signature. `onHoliday` is `match` that already has a `Regex`.

```
// replace :: Regex -> (String -> (String -> String))
const replace = curry((reg, sub, s) => s.replace(reg, sub));
```

As you can see with the full parenthesis on `replace`, the extra notation can get a little noisy and redundant so we simply omit them. We can give all the arguments at once if we choose so it's easier to just think of it as: `replace` takes a `Regex`, a `String`, another `String` and returns you a `String`.

A few last things here:

```
// id :: a -> a
const id = x => x;

// map :: (a -> b) -> [a] -> [b]
const map = curry((f, xs) => xs.map(f));
```

The `id` function takes any old type `a` and returns something of the same type `a`. We're able to use variables in types just like in code. Variable names like `a` and `b` are convention, but they are arbitrary and can be replaced with whatever name you'd like. If they are the same variable, they have to be the same type. That's an important rule so let's reiterate: `a -> b` can be any type `a` to any type `b`, but `a -> a` means it has to be the same type. For example, `id` may be `String -> String` or `Number -> Number`, but not `String -> Bool`.

`map` similarly uses type variables, but this time we introduce `b` which may or may not be the same type as `a`. We can read it as: `map` takes a function from any type `a` to the same or different type `b`, then takes an array of `a`'s and results in an array of `b`'s.

Hopefully, you've been overcome by the expressive beauty in this type signature. It literally tells us what the function does almost word for word. It's given a function from `a` to `b`, an array of `a`, and it delivers us an array of `b`. The only sensible thing for it to do is call the bloody function on each `a`. Anything else would be a bold face lie.

Being able to reason about types and their implications is a skill that will take you far in the functional world. Not only will papers, blogs, docs, etc, become more digestible, but the signature itself will practically lecture you on its functionality. It takes practice to become a fluent reader, but if you stick with it, heaps of information will become available to you sans

RTFMing.

Here's a few more just to see if you can decipher them on your own.

```js
// head :: [a] -> a
const head = xs => xs[0];

// filter :: (a -> Bool) -> [a] -> [a]
const filter = curry((f, xs) => xs.filter(f));

// reduce :: (b -> a -> b) -> b -> [a] -> b
const reduce = curry((f, x, xs) => xs.reduce(f, x));
```

`reduce` is perhaps, the most expressive of all. It's a tricky one, however, so don't feel inadequate should you struggle with it. For the curious, I'll try to explain in English though working through the signature on your own is much more instructive.

Ahem, here goes nothing....looking at the signature, we see the first argument is a function that expects a `b`, an `a`, and produces a `b`. Where might it get these `a`s and `b`s? Well, the following arguments in the signature are a `b` and an array of `a`s so we can only assume that the `b` and each of those `a`s will be fed in. We also see that the result of the function is a `b` so the thinking here is our final incantation of the passed in function will be our output value. Knowing what reduce does, we can state that the above investigation is accurate.

Narrowing the Possibility

Once a type variable is introduced, there emerges a curious property called *parametricity*. This property states that a function will *act on all types in a uniform manner*. Let's investigate:

```js
// head :: [a] -> a
```

Looking at `head`, we see that it takes `[a]` to `a`. Besides the concrete type `array`, it has no other information available and, therefore, its functionality is limited to working on the array alone. What could it possibly do with the variable `a` if it knows nothing about it? In other words, `a` says it cannot be a *specific* type, which means it can be *any* type, which leaves us with a function that must work uniformly for *every* conceivable type. This is what *parametricity* is all about. Guessing at the implementation, the only reasonable assumptions are that it takes the first, last, or a random element from that array. The name `head` should tip us off.

Here's another one:

```
// reverse :: [a] -> [a]
```

From the type signature alone, what could `reverse` possibly be up to? Again, it cannot do anything specific to `a`. It cannot change `a` to a different type or we'd introduce a `b`. Can it sort? Well, no, it wouldn't have enough information to sort every possible type. Can it re-arrange? Yes, I suppose it can do that, but it has to do so in exactly the same predictable way. Another possibility is that it may decide to remove or duplicate an element. In any case, the point is, the possible behaviour is massively narrowed by its polymorphic type.

This narrowing of possibility allows us to use type signature search engines like Hoogle to find a function we're after. The information packed tightly into a signature is quite powerful indeed.

Free as in Theorem

Besides deducing implementation possibilities, this sort of reasoning gains us *free theorems*. What follows are a few random example theorems lifted directly from Wadler's paper on the subject.

```
// head :: [a] -> a
compose(f, head) === compose(head, map(f));

// filter :: (a -> Bool) -> [a] -> [a]
compose(map(f), filter(compose(p, f))) === compose(filter(p), map(f));
```

You don't need any code to get these theorems, they follow directly from the types. The first one says that if we get the `head` of our array, then run some function `f` on it, that is equivalent to, and incidentally, much faster than, if we first `map(f)` over every element then take the `head` of the result.

You might think, well that's just common sense. But last I checked, computers don't have common sense. Indeed, they must have a formal way to automate these kind of code optimizations. Maths has a way of formalizing the intuitive, which is helpful amidst the rigid terrain of computer logic.

The `filter` theorem is similar. It says that if we compose `f` and `p` to check which should be filtered, then actually apply the `f` via `map` (remember filter, will not transform the elements - its signature enforces that `a` will not be touched), it will always be equivalent to mapping our `f` then filtering the result with the `p` predicate.

These are just two examples, but you can apply this reasoning to any polymorphic type signature and it will always hold. In JavaScript, there are some tools available to declare rewrite rules. One might also do this via the `compose` function itself. The fruit is low hanging and the possibilities are endless.

Constraints

One last thing to note is that we can constrain types to an interface.

```
// sort :: Ord a => [a] -> [a]
```

What we see on the left side of our fat arrow here is the statement of a fact: `a` must be an `Ord`. Or in other words, `a` must implement the `Ord` interface. What is `Ord` and where did it come from? In a typed language it would be a defined interface that says we can order the values. This not only tells us more about the `a` and what our `sort` function is up to, but also restricts the domain. We call these interface declarations *type constraints*.

```
// assertEqual :: (Eq a, Show a) => a -> a -> Assertion
```

Here, we have two constraints: `Eq` and `Show`. Those will ensure that we can check equality of our `a`s and print the difference if they are not equal.

We'll see more examples of constraints and the idea should take more shape in later chapters.

In Summary

Hindley-Milner type signatures are ubiquitous in the functional world. Though they are simple to read and write, it takes time to master the technique of understanding programs through signatures alone. We will add type signatures to each line of code from here on out.

Chapter 08: Tupperware