WILL JONES

ADVANCED HASKELL

Contents

Higher-order functions

By now you will have seen several examples of higher-order functions: `map`, `zipWith` and `foldr` are but a few of many already provided by the Haskell Prelude. In this chapter you'll see just how flexible some of these functions are and gain an appreciation for some functions which up until now may have seemed, well, useless.

1. You may have seen `(.)` (often pronounced "dot"), which implements function composition. For example:

   ```
      ((2 +) . (2 *)) x          (sum . replicate 10) 1
   == (2 +) ((2 *) x)         == sum ((replicate 10) 1)
   == 2 + (2 * x)             == sum (replicate 10 1)
   ```

   We can use `(.)` to write functions in a so-called "point-free" style, in which the specific arguments (or "points") on which a function operates may be omitted. As an example, suppose that the first of the two lines above formed the definition for a function `doublePlusTwo`:

   Humorously referred to by some as "pointless" programming.

   ```
   doublePlusTwo x = ((2 +) . (2 *)) x
   ```

   We know that if `f x = g x` (that is to say, `f x` is defined as being equal to `g x`) then it must be the case that `f = g`. Consequently, we can rewrite `doublePlusTwo` as follows:

   ```
   doublePlusTwo = (2 +) . (2 *)
   ```

   Rewrite the following familiar definitions so that they do not mention the point `xs`:

   ```
   sum xs    = foldr (+) 0 xs
   any p xs  = or (map p xs)
   odds xs   = length (filter odd xs)
   ```

2. Suppose that we define:

   ```
   (.:) = (.) . (.)
   ```

What is `(.:)`'s type? From this, can you explain in English what `(.:)` does? Can you use it to make the earlier definition of:

```
any p xs = or (map p xs)
```

completely point free? Furthermore, can you inline the definition of `(.:)` and demonstrate a general technique for rewriting arbitrary functions in a point-free manner?

3. `flip` takes a function `f`, say, which expects two arguments and "flips" it so that it takes those arguments in the opposite order. It may be defined as:

   ```
   flip :: (a -> b -> c) -> b -> a -> c
   flip f x y = f y x
   ```

   Can you use `flip` to flip a three-argument function of type `a -> b -> c -> d`? Is there more than one way of flipping such a function?

4. If `flip` takes a function that expects two arguments, why does `flip id` have a type? How many arguments does `id` take?

5. As well as the fold operations you've already encountered, the Prelude provides a series of "scans":

   ```
   scanr   :: (a -> b -> b) -> b -> [a] -> [b]
   scanl   :: (a -> b -> a) -> a -> [b] -> [a]
   scanr1  :: (a -> a -> a) -> [a] -> [a]
   scanl1  :: (a -> a -> a) -> [a] -> [a]
   ```

   Scans differ from folds in that they return the list of intermediate values, so that where `foldl (+) 0 [1..5]` evaluates to `15`, for example, the analogous expression `scanl (+) 0 [1..5]` produces the *list* of values `[0,1,3,6,10,15]`. In order to familiarise yourself with the scans, try implementing the following:

   - The value `factsS :: [Integer]`, which evaluates to the infinite list of factorials $1!, 2!, 3!, \ldots$
   - (Tricky) The value `fibsS :: [Integer]`, which evaluates to the infinite list of Fibonacci numbers $0, 1, 1, 2, 3, \ldots$

   Hint: Try tying a knot by defining `fibsS = 0 : s g z fibsS`, where `s` is the scan you plan to use and `g` and `z` are the scan function and zero respectively.

6. There lurks in the Prelude a function named `const`, with the following type:

   ```
   const :: a -> b -> a
   ```

   Without looking at its implementation, can you tell whether or not it's a higher-order function? Better yet, can you *derive* its implementation from its type?

7. You may have heard that "everything is a fold", at least in the world of lists. Can you write a version of the Prelude's `head` function, say:

   ```
   headF :: [a] -> a
   ```

   using only `foldr`?

   Hint: There lurks in the Prelude a function named `const`...

8. Of course, since `head` isn't a recursive function, using `foldr` to implement it is perhaps overkill. Two slightly trickier problems are reversing a list and concatenating two lists, for which the Prelude provides respectively:

   ```
   reverse :: [a] -> [a]
   (++)    :: [a] -> [a] -> [a]
   ```

   Devise functions `reverseF` and `appendF` which implement `reverse` and `(++)` using `foldr` to capture the necessary recursion.

9. Now let's use a higher-order function to build another higher-order function. Implement:

   ```
   mapF :: (a -> b) -> [a] -> [b]
   ```

   Where `mapF f xs = foldr g z xs`, for some `g` and `z`.

10. (Hard) When we say "everything is a fold", which fold do we mean? We know of `foldr`, `foldl` and even `foldr1` and `foldl1`. Is there a definitive "one fold to rule them all" from which all the others may be derived?

    Hint: Consider what happens when you instantiate the value `z` in the expression `foldr f z` to `id`.

11. We've already seen that `map` can be expressed in terms of `foldr`, but do we really need all the power that `foldr` offers to implement `map`? Try writing a function, `mapZ`, say, which instead uses `zipWith` to enact recursion. That is to say, write a function `mapZ`, say, which has the type:

    ```
    mapZ :: (a -> b) -> [a] -> [b]
    ```

    and behaves identically to the Prelude's `map` function.

12. Can you play a similar trick and implement `zipWith` using only `zipWith3`? If so, can you identify a pattern and implement a function `mkZ` such that:

    ```
    mkZ zipWith4  == zipWith3
    mkZ zipWith3  == zipWith
    mkZ zipWith   == map
    ```

    What is `mkZ`'s type?

13. Implement `scanr` in terms of `foldr` and `scanl` in terms of `foldl`.

14. (Very hard) Write `zipWith` using `foldr`. You may wish to start with a definition of the form:

```
zipWithF f xs ys = foldr g z xs ys
```

and work backwards from there.

Hint: what are the types of `g` and `z` in the definition given? Does that guide your answer?

15. Write a function `swing`, such that:

```
swing map :: [a -> b] -> a -> [b]
swing any :: [a -> Bool] -> a -> Bool
```

What is `swing`'s type? What sorts of functions does your implementation work with?

16. Write a function which computes "two-dimensional maps":

```
map2D :: (a -> b) -> [[a]] -> [[b]]
```

such that `map2D (2 *) [[1,2],[3,4]]` gives `[[2,4],[6,8]]`. Can you make your resulting implementation entirely point free? If so, can you derive a method for making `map3D`, `map4D` and so on?

Type classes and type constructors

At this point you should at least have a passing familiarity with several of the type classes defined in Haskell's Prelude – `Num`, `Fractional`, `Show` and so on. In this chapter we'll examine some other type classes provided by Haskell's standard libraries and the patterns they capture.

Monoids

The `Monoid` class encompasses types for which there is a meaningful "empty" element, `mempty`, and function which appends elements, `mappend`:

The class is defined by the `Data.Monoid` module, though you may wish to define it yourself as you will be rewriting parts of that module in answering the questions that follow.

```
class Monoid a where
mempty  :: a
mappend :: a -> a -> a
```

Instances of `Monoid` must respect the following *laws*:

Or rather, *should* respect the following laws – Haskell's type system is not powerful enough to enforce them.

```
mempty ‘mappend‘ x          == x
x ‘mappend‘ mempty          == x
x ‘mappend‘ (y ‘mappend‘ z) == (x ‘mappend‘ y) ‘mappend‘ z
```

That is to say, `mempty` is an *identity* for `mappend` and `mappend` is an *associative* function.

1. A monoid you are already familiar with is that concerning lists, where `mempty` is the empty list and `mappend` is the `(++)` operator:

   ```
   instance Monoid [a] where
     mempty          = []
     xs ‘mappend‘ ys = xs ++ ys
   ```

   Can you write an `instance Monoid Int` that obeys the monoid laws?

2. Can you write another, different `Monoid` instance for `Int` that also obeys the monoid laws?

3. It is often desirable to write two different, co-existing, type class instances for the same type (as in your `Monoid Int` instances above). However, Haskell's type system won't permit this. To get around the problem, we can use a `newtype`:

   ```
   newtype Sum = Sum Int
   ```

   Much like `data`, `newtype` defines a new data type. However, `newtypes` are only allowed to introduce a *new name for an existing type*: they may thus only possess a single constructor. Given the definition of `Sum` above (along with the hint it may give you!) and the additional type:

   ```
   newtype Product = Product Int
   ```

   rewrite your two `Monoid Int` definitions so that they may co-exist.

4. Write a function `mconcat`:

   ```
   mconcat :: Monoid a => [a] -> a
   ```

   which concatenates a list of monoidal values into a single value.

   Hint: for the list monoid defined earlier, `mconcat xss` should equal `concat xss`.

5. Given functions:

   ```
   getSum :: Sum -> Int            getProduct :: Product -> Int
   getSum (Sum x) = x              getProduct (Product x) = x
   ```

   define the functions:

   ```
   sumM      :: [Int] -> Int
   productM  :: [Int] -> Int
   ```

   using `mconcat`. Your definitions should behave as the Prelude's `sum` and `product` functions respectively (ignoring the fact that they only work on integers, of course).

6. Simple destructor functions such as `getSum` and `getProduct` may be defined using "record syntax":

   This works for types introduced with `data` too.

   ```
   newtype Sum     = Sum { getSum :: Int }
   newtype Product = Product { getProduct :: Int }
   ```

   For example, the definitions:

   ```
   newtype All = All { getAll :: Bool }
   newtype Any = Any { getAny :: Bool }
   ```

introduce the destructor functions `getAll :: All -> Bool` and `getAny :: Any -> Bool`, with the obvious implementations. Using these building blocks, write `Monoid` instances for `All` and `Any` and subsequently derive the two functions from the Prelude that you can now rewrite using `mconcat`.

Hint: it's not a trick question!

7. Of course, there are natural ways of gluing *functions* together, too. Write an instance of `Monoid` for the following type:

```
newtype Endo a
  = Endo { runEndo :: a -> a }
```

Hint: Assuming that `mempty`'s definition is of the form `Endo f`, how many such functions `f :: a -> a` are there?

Foldables

In the same way that `Functor` generalises "mapping" over some "container" of values, `Foldable` generalises the notion of folding, or reducing, such containers into a single value:

```
class Foldable f where
  foldMap :: Monoid m => (a -> m) -> f a -> m
  foldr   :: (a -> b -> b) -> b -> f a -> b
```

As with `Monoid`, `Foldable` is defined in its own module, `Data.Foldable`. In fact, the class definition in the `Data.Foldable` module also includes variants such as `foldr1`, `foldl`, `foldl1` and `foldl'`, with which you may be familiar. The question, then, is this: why have we omitted them here?

Again, you may wish to define it yourself in order to avoid clashes when answering the questions below; consider hiding the Prelude's version of `foldr` through an `import Prelude hiding (foldr)`.

1. Write an instance of `Foldable` for lists. Naturally, `foldr` will be defined as in your course notes, but what about `foldMap`? Recall that the list type constructor is written as `[]`; you should thus write an instance of the form:

```
instance Foldable [] where
  ...
```

2. Write an instance `Foldable Tree` for the `Tree` type you've seen throughout your course:

```
data Tree a
  = Empty
  | Node (Tree a) a (Tree a)
```

3. Implement a function `toList :: Foldable f => f a -> [a]` which turns any `Foldable` structure into a list containing the same elements.

Hint: consider that `toList` should be the identity function when applied to a list.

4. In the previous chapter, we saw that given an implementation of `foldr` over lists, we can implement any of `foldl`, `foldr1` and so on using just that function. We might expect the same to hold true of any `Foldable`

structure, explaining our omission of these functions from the `Foldable` class. However, what about `foldMap`? Can you implement it in terms of `foldr`?

5. If you succeeded in implementing `foldMap` using `foldr`, you may think it just another derivable function and thus a candidate for removal from the `Foldable` class. However, it is possible that `foldr`'s ability to implement `foldMap` stems from its ability to implement *any* fold, not that `foldMap` is a "weaker" fold. Can you show that this is indeed the case by implementing `foldr` *in terms of* `foldMap`?

Hint: What happens when you pick the `Endo` type from the previous section as `foldMap`'s target `Monoid` instance?

Functors and applicative functors

As discussed in the lectures, functors and their more powerful applicative counterparts generalise "mapping" and "zipping" over composite values:

Recall that `(<*>)` is pronounced "app".

```
class Functor f where
  fmap  :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

While `Functor` is provided by the Prelude, `Applicative` was only added later[1] and is thus defined in the `Control.Applicative` module.

[1] Conor McBride and Ross Paterson. "Applicative Programming With Effects". In: *Journal of Functional Programming* 18.1 (Jan. 2008), pp. 1–13. ISSN: 0956-7968

1. At this point you've seen `Functor` and `Applicative` instances for type constructors such as `[]`, `Maybe` and `Tree`. Notice that these constructors are all *unary* – they take a single type and produce a new one. What about (for instance) *binary* constructors, such as that which builds pairs? Ignoring the syntactic sugar to which you are used, the pair constructor is defined as:

   ```
   data (,) a b = (,) a b
   ```

   Such a definition may suggest the possibility of *partial application at the type level* and indeed, this is the case! `(,) a` is the type constructor which takes a type, `b`, say, and produces the type of `a`s paired with `b`s! Armed with this knowledge, can you write a `Functor` instance for the type of "pairs whose first element is of type `a`"?

   ```
   instance Functor ((,) a) where
     ...
   ```

2. Unfortunately, we encounter difficulties when trying to make our partially-applied pair an instance of `Applicative`:

```
instance Applicative ((,) a) where
  pure  :: b -> (a, b)
  (<*>) :: (a, b -> c) -> (a, b) -> (a, c)
```

In the case of `pure`, we need to produce some value of type `a`, seemingly out of thin air. As for `(<*>)`, we're spoilt for choice – do we pick the `a` paired with the function or the `a` paired with the argument? Without any extra information, there's certainly no way we can combine the two.

Can you spot the extra information we need about the type `a`? Furthermore, can you use that information to write a working instance?

Hint: Consider adding a constraint `C a`, as in `instance C a => Applicative ((,) a)`, where `C` is a class we've already discussed.

3. Whereas the type `(a, b)` encodes conjunction—the notion that we have both a value of type `a` and a value of type `b`—the type `Either a b` encodes disjunction, whereby we only have a value of type `a` *or* a value of type `b`:

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

Write `Functor` and `Applicative` instances for an appropriate partial application of `Either`.

4. (From your lectures) The `(->)` type constructor (pronounced "arrow") is not special – we can write instances for it as we would any other type. `Functor` and `Applicative` are no exception, provided we partially apply `(->)` (since it is a binary constructor "functor-like" things must be unary constructors):

We'll return to providing precise definitions for "functor-like" things and the like later on in your course.

```
instance Functor ((->) e) where
  ...

instance Applicative ((->) e) where
  ...
```

Can you complete these instances? You are strongly encouraged *not* to look it up (!) – the results are extremely beautiful and offer a lot of insight into Haskell and programming in general.

5. In your lectures, you were exposed to the first of the four `Applicative` laws:

```
pure id <*> xs                == xs
pure (.) <*> fs <*> gs <*> xs == fs <*> (gs <*> xs)
pure f <*> pure x             == pure (f x)
fs <*> pure x                 == pure ($ x) <*> fs
```

which state that applicative functors should preserve identity, composition and application respectively (the latter requiring two laws to express completely). Furthermore, you encountered the first law (identity) when deriving the following instance of `Applicative` for lists:

```
instance Applicative [] where
  pure x              = repeat x
  (f : fs) <*> (x : xs) = f x : (fs <*> xs)
  _ <*> _             = []
```

That is, where `(<*>)` recovers the $\texttt{zipWith}_n$ family of functions. There is in fact *another* instance `Applicative []` which obeys the `Applicative` laws – can you find it?

Monads

Perhaps Haskell's most (in)famous type class, the `Monad` class introduces the `(»=)` (pronounced "bind") operator, which allows one to "flatten" or "extract" the value from a lifted computation in order to feed it to another lifted computation. Importantly, this allows lifted computations to *depend* on the results of previously evaluated lifted computations:

```
class Monad m where
  return  :: a -> m a
  (>>=)   :: m a -> (a -> m b) -> m b
```

Note that, for historical reasons, `Monad` depends on neither `Functor` nor `Applicative`, even though (as we shall see) all monads can be made instances of both these classes. In this respect, `return` is just an alias for the `pure` function of the `Applicative` class – `(»=)` is the only new addition to the team.

Recall that `(»=)` is the mechanism by which Haskell's "`do`-notation" is desugared. For example:

```
do
  x <- mx                          mx >>= \x ->
  y <- f x                           f x >>= \y ->
  return (x, y)                        return (x, y)
```

allowing us to write lifted (or, more typically, effectful) computations in an imperative style. Indeed, many would claim that Haskell is also the world's most beautiful imperative language!

1. "All monads are functors." Write a function:

```
   liftM :: Monad m => (a -> b) -> m a -> m b
```

using only the members of the `Monad` type class. By its type, such a function would seem to be a suitable implementation of `fmap` – is this the case for the `[]` and `Maybe` monads?

2. "All monads are applicative functors." Write a function:

```
ap :: Monad m => m (a -> b) -> m a -> m b
```

using only the members of the `Monad` type class. Here, the type resembles that of `(<*>)`; is it so? Try checking for the `[]` and `Maybe` monads once more.

Note: the `Monad []` instance defined by the Prelude matches the other `Applicative []` instance you may have discovered at the end of the previous section, and not the "zip"-style instance discussed in your lectures. Indeed, you may use this to derive the answer for the aforementioned question!

3. Recall that your lectures introduced `(»=)` as a shortcut for the act of "joining" the two layers of a lifted computation produced by using `fmap`:

```
join  :: Monad m => m (m a) -> m a

(>>=) :: Monad m => m a -> (a -> m b) -> m b
m >>= f
  = join (fmap f m)
```

This definition of `(»=)` illustrates that `join` is at least as expressive as `(»=)`, but our omission of `join` from the `Monad` class indicates that we believe `(»=)` equally as expressive. Can you confirm the correctness of this belief by implementing `join` using `(»=)`?

4. Just as there are instances of `Functor` and `Applicative` for the partially-applied arrow type `((->) e)`, so too is there a `Monad` instance:

```
instance Monad ((->) e) where
  ...
```

Can you find it? What effect does it let you work with?

Bibliography

[1] Conor McBride and Ross Paterson. "Applicative Programming With Effects". In: *Journal of Functional Programming* 18.1 (Jan. 2008), pp. 1–13. ISSN: 0956-7968.