Contents

Turning common "shapes" into functions

Remember these?

```
sum [] = 0
sum (n:ns) = n + sum ns

length [] = 0
length (_:xs) = 1 + length xs

prod [] = 1
prod (n:ns) = n * prod ns
```

They can a common pattern, which is typically referred to as `"folding"`.

Can we abstract this?

Can we produce something (`<abs-fold>`) that captures folding?

Common Aspects

Whey all have the empty list as a base case

```
sum [] = 0
length [] = 0
prod [] = 0
<abs-fold> [] = ...
```

They all have a non empty list as the recursive case

```
sum (n:ns) = n + sum ns
length (_:xs) = 1 + length xs
prod (n:ns) = n * prod ns
<abs-fold> (a:as) = ... <abs-fold> as
```

The base case returns a fixed "unit" value, which we will call `u`

```
<abs-fold> [] = u
```

The recursive case combines the head of the list with the result of the recursive call, using a binary operator we shall call `op`

```
<abs-fold> (a:as) = a `op` <abs-fold> as
```

So we now have the following abstract form

```
<abs-fold> [] = u
<abs-fold> (a:as) = a `op` <abs-fold> as
```

But how do we instantiate <abs-fold>?

Our concrete `fold` needs to be a function that is supplied with `u` and `op` as arguments, and then builds a function on lists as above.

So <abs-fold> becomes `fold u op`

```
fold u op [] = u
fold u op (a:as) = a `op` fold u op as
```

This is a Higher Order Function that captures a basic recursive pattern on lists.

So we now have <abs-fold>`fold u op`. So how do we use fold to save boilerplate code?

```
sum = fold 0 (+)
length = fold 0 incr where _ `incr` y = 1 + y
prod = fold 1 (*)
```

The type of fold

```
fold u op [] = u
fold u op (a:as) = a `op` fold u op as

-- a :: t, as :: [t]
-- u :: r  -- result type may differ, e.g. length
-- op :: t -> r -> r  -- 1st from list, 2nd a "result"

fold :: r -> (t -> r -> r) -> [t] -> r
```

Fold in Haskell

- Haskell has a number of variants of `fold`
- "Fold-right" (`foldr`) is like our `fold` in that the uses of `op` are nested on the *right*

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr (+) 0 [10, 11, 12] = 10 + (11 + (12 + 0))
```

Note: The order of `u` and `op` are also different

- "Fold-left" (`foldl`) is different that the uses of `op` are nested on the *left*

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl (+) 0 [10, 11, 12] = ((0 + 10) + 11) + 12
```

We shall see the results for the distinction later

- There are also variants that don't require the unit `u` to be specified, but which are only defined for non-empty list

Type Constaints

Take this function

```
addpair (x, y) = x+y
```

Which of the following would be the correct type?

```
addpair :: (Integer, Integer) -> Integer
addpair :: (Float, Float) -> Float
addpair :: ([Bool], [Bool]) -> [Bool]
```

Is there a type that covers the first two but not the third? This is too generous:

```
addpair :: (a, a) -> a
```

In order to narrow the acceptable values to things that can be added, we can write this:

```
addpair :: Num a => (a, a) -> a
```

- The delaration `Num a => (a, a) -> a` constains what is known as a *type constraint* (here, `Num a =>`)
- The constaint says that the type `a` must be part of the *class of types* `Num`
- A number of predefined type classes:
  - `Num`: Defines `+` and `-`, among others
  - `Eq`: Defines `==`
  - `Ord`: Defines comparisons, `<=`
  - `Show`: Can convert to `String` (like `toString()` in Java)
  - many more…
- The mention of the class name is a promise that some set of functions will work on the values of that class.
- A type class is an *interface* that the compiler will check for you, allowing you to say things like "this function accepts anything that `(+)` works on"

Being classy

A type class definition has the form

`class` $ClassName\ t_1\ t_2\ \ldots\ t_i$ `where{` $f_1 :: sig_1; \ldots\ f_i :: sig_i;$ `}`

f_i are the names of the functions defined by the type class, and sig_i are their type signatures. t_i are the *type parameters*.

Let's consider the `Show` class:

```
class Show a where
  show :: a -> String
```

The type class `Show` defines one function:

```
show :: a -> String
```

which converts the argument into a String.

```
data Day = Monday | Tuesday | ... | Sunday
```

If we type the name of one of these data constructors in the GHCi prompt, we see

```
> Monday
No instance for (Show Day) arising from a use of print
```

This means that the function 'print' has a type constraint that requires its argument to be a member of the Show class.

New types do not automatically belond to any classes. However, we can get the compiler to generate a default class instance for many built-in classes automatically. To do this, we use the `deriving` keyword.

```
data Day = Monday | Tuesday | ... | Sunday
           deriving (Show, Eq)
```

Writing our own Type Class Instances

We can provide our own Showinstance by writing an instance declaration

```
data Day = Monday | Tuesday | ... | Sunday

instance Show Day where
  show Monday = "Maandag"
  show Tuesday = "Dinsdag"
  ...
  show Sunday = "Zondag"
```

`Show` is easy because there is only one function. We will return to other type classes later.

The Type of Equality

- We test for equality, using infix operator `==`

```
> 1 == 2
False
> [1, 2, 3] == (reverse [3, 2, 1] )
True
```

- What is the type of `==`?
  - It compares things of the same type to give a boolean result: `(==) :: a -> a -> Bool` (so it is polymorphic, then)
  - What does Haskell think?

```
> :t (==)
(==) :: (Eq a) => a -> a -> Bool
```

- Equality is "polymorphic"

```
(==) :: a -> a -> Bool
```

- However, it is *ad-hoc*
  - There has to be a dpecific (different) implementation of it for each type

```
primIntEq :: Int -> Int -> Bool
primFloatEq :: Float -> Float -> Bool
```

- Constrast with the (parametric) polymorphism of `length`
  - The same program code works for all lists, regardless of the underlying element type.

```
length [] = 0
length (x:xs) = 1 + length xs
```

Ad-hoc polymorphism is Ubiquitous

- Ad-hoc polymorphism is very common in programming languages

Operators	Types
$= \neq < \leq > \geq$	$T \times T \rightarrow \mathbb{B}$, for (almost) all types ot T
$+ - * /$	$N \times N \rightarrow N$, for numeric types N

- The use of single symbol (`+`, say) to denote lots of (different but related) operators, is also often called "overloading"
- In many programming languages this overloading is built-in
- In Haskell, it is a language feature called "type-classes", so we can "roll our own"

Defining (Type-)Classes in Haskell (Overloading)

- In order to define our own name/operator overloading we:
  - Need to specify the name/operator involved (e.g. `==`)
  - Need to describe its pattern of use(e.g. `a -> a -> Bool`)
  - Need an overarching "class" name for the concept (e.g. `Eq`)
- In order to use our operator with a given type (e.g. `Bool`), we:
  - Need to give the implementation of `==` for that type (`Bool -> Bool -> Bool`)
  - In other words, we give an instance of the type for the class

Defining The Equality Class

- We define the class `Eq` as follows:

```
class Eq a where
  (==) :: a -> a -> Bool
```

- The first line introduced `Eq` as a class characterising a type (here called `a`)
- The second line declares that a type belonging to this class must have an implementation of `==` of the type shown
- `class` and `where` are Haskell keywords

Giving an instance of the Equality Class

- We define an instance of `Eq` for booleans as follows

```
instance Eq Bool where
  True == True   = True
  False == False = True
  _ == _         = False
```

- How all we do is define instances for the other types for which equality is desired

- (In fact, in many cases, for equality, we simply refer to a primitive builtin function to do the comparison)
- Most of this is already done for us as part of the Haskell *Prelude*

- `instance` is a Haskell keyword

The "real" Equality Class

- In fact, `Eq` has a slightly more complicated definition:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
    -- Minimal complete definition: (==) or (/=)
  x /= y = not (x==y)
  x == y = not (x/=y)
```

- First, an instance must also provide `/=`
- Second, we give (circular) definitions of `==` and `/=` in terms of each other
  - The idea is that an instance need only define one of these
  - The other is then automatically derived
  - However, we may want to explicitly define both (for efficiency)

How Haskell handles a class name/operator

- Consider the following (well-typed) expression

```
x == 3 && y == False
```

- The compiler sees the symbol `==`, notes it belondss to the `Eq` class and then…
  - Seeing `x :: Int` deduces (via type intference) that the first `==` has the type `Int -> Int -> Bool`
  - Generators code using that instance for that use of equality
  - Does a similar analysis of the second `==` symbol, and generates boolean-equality code there

- Now consider the following (well-typed) expression

```
x === 3 && y == False || z == MyCons
```

- The compiler, seeing the 3rd `==`, looks for an instance for `MyType` of `Eq`, and fails to find one

- It generators an error message of the form

```
No instance for (Eq MyType)
  arising from a use of `==` at...
Possible fix: add an instance declaration for (Eq MyType)
```