3 Types, Polymorphism and Classes

Haskell programs are statically typeable, meaning that the type of an expression can be established from the components without evaluating the expression. The language is strongly typed: an expression has no value unless its type can be inferred, or checked against a claimed type. Its value has to be one of the values of its type.

There are some built-in types, such as *Int*, *Float*, *Char*. There are also some built-in type constructors, in particular the function type $a \to b$ for any types a and b. The types of lists, such as $[Int]$, will turn out to be built-in only in as much as there is a special syntax. Similarly, tuples such as pairs $(Int, Char)$, triples $(Int, Char, Bool)$, and so on. There are also empty tuples, () whose sole value is ().

Declarations like

```
type String = [Char]
```

introduce new *names* for existing types.

3.1 New types

You might think that *Bool* has to be built in, but it could be (and is) defined by

```
data Bool = False | True
```

This declaration introduces a new type, *Bool* and new constants *False* and *True*. Similarly there are predefined types

```
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
```

As well as introducing a range of types, *Maybe a* for each type a and *Either a b* for each type a and type b, these declarations introduce

```
Nothing :: Maybe a
Just :: a -> Maybe a
Left :: a -> Either a b
Right :: b -> Either a b
```

Values of *Either Int Char* include *Left* 42 and *Right* 'x'.

A **data** declaration something akin to a record in other programming languages

```
> data PairType a b = Pair a b
```

introduces a family of types *PairType a b* for each pair of types a and b, together with a function

```
Pair :: a -> b -> PairType a b
```

This new type is equivalent to (a, b). (I might usually have chosen the same name for both *Pair* and *PairType*.)

The function *Pair*, exceptionally for the names of values in Haskell, has a name spelled with an upper case initial, which marks it out (like *True* and *False*, *Left* and *Right* and so on) as a *constructor*: a function which by the form of its definition must be invertible. Constructors can appear on the left hand side of a function definition, such as

```
> sumPair :: PairType Int Int -> Int
> sumPair (Pair x y) = x + y
```

3.2 Polymorphic types

Many standard functions have types that mark them out as being applicable to arguments of many types, for example

```
map :: (a -> b) -> [a] -> [b]
reverse :: [a] -> [a]
```

Types with lower case names are *type variables*, and these types should be read as being "for all types a and b, *map* can have type $(a \to b) \to [a] \to [b]$", and "for all types a, *reverse* can have type $[a] \to [a]$".

When a polymorphic function is applied to an argument of a more constrained type (or vice versa) the polymorphic type is constrained to match.

If you want an instance of *reverse* that applies only to lists of *Char*, and cannot accidentally be applied to a $[Int]$, the expression $reverse :: [Char] \to [Char]$ will do that. (This is deliberately similar to the type signature declaration.)

This is *parametric polymorphism* and the value of a parametrically polymorphic expression is very constrained by its type. For example, there is a standard function

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
```

Such a function has to satisfy

```
either f g (Left x)  = ...
either f g (Right y) = ...
```

where the right-hand sides are both of type c, whatever that type happens to be. There is in each case essentially only one possible expression known to be of that type: $f\ x$ and $g\ y$ respectively.

A parametrically polymorphic function has to operate uniformly on all possible instances of its argument type. The type of *reverse* requires that it treats all lists the same: it cannot inspect the elements of the list because it cannot know what they are. It turns out that a consequence of this is that for any $fun :: [a] \to [a]$

$$map\ f \cdot fun = fun \cdot map\ f$$

You can easily see that this is true for *reverse*; it is also true for any other function (and there are many) of the same type; but the force of the result is that a function that does not satisfy this equation cannot have that type.

Such results, theorems that follow from the types, were called *theorems for free* in a paper by Phil Wadler; clearly a mathematical salesman.

3.3 Selectors, Discriminators, Deconstructors

A constructor like *Pair* makes a value (the jargon is a *product*) which behaves like a record with two fields. These can be recovered by *selector* functions

```
> first, second :: Pair a -> a
> first (Pair x y) = x
> second (Pair x y) = y
```

The constructors of a type like an *Either* make a value (a *sum*) which behaves like a union of two possible fields. The selectors for this type are partial functions

```
> left :: Either a b -> a
> left (Left x) = x
> right :: Either a b -> b
> right (Right y) = y
```

and they are little use without a *discriminator* that tells you which kind of value you have, for example

```
> isleft :: Either a b -> Bool
> isLeft (Left x) = True
> isLeft (Right x) = False
```

I will try to use the term *deconstructors* for the discriminator and selectors of a type. Informally, the community often refers to the discriminator and selectors for a type as the *destructors*, although this is not a good usage: it confuses them with the operations in languages that requires the programmer to manage the memory occupied by data structures in a program.

Deconstructors for a type are enough to let you write functions on that type without pattern matching, for example

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f g e | isLeft e  = f (left e)
             | otherwise = g (right e)
```

3.4 Type inference

How does Haskell check the types of expressions? The essential rule is that if f is applied to x, then f needs to have a function type $a \to b$, and x needs to have type a, and the resulting application $f\ x$ has type b.

A definition such as

```
flip f x y = f y x
```

need not have an explicit type signature because Haskell can deduce a type for *flip* from this rule. The algorithm allocates a type variable to each name

$$\mathit{flip} :: \beta \qquad f :: \gamma \qquad x :: \delta \qquad y :: \epsilon$$

and to the result of each application, then assembles the constraints:

$$\begin{array}{lcl} \mathit{flip}\ f & \Rightarrow & \beta = \gamma \to \zeta \\ (\mathit{flip}\ f)\ x & \Rightarrow & \zeta = \delta \to \eta \\ ((\mathit{flip}\ f)\ x)\ y & \Rightarrow & \eta = \epsilon \to \theta \\ f\ y & \Rightarrow & \gamma = \epsilon \to \iota \\ (f\ y)\ x & \Rightarrow & \iota = \delta \to \kappa \end{array}$$

and then finally because both sides are equal, $\kappa = \theta$. These constraints can be met by substitution:

$$\begin{array}{rcl} \beta & = & \gamma \to \zeta \\ & = & (\epsilon \to \iota) \to \delta \to \eta \\ & = & (\epsilon \to \delta \to \kappa) \to \delta \to \epsilon \to \theta \\ & = & (\epsilon \to \delta \to \kappa) \to \delta \to \epsilon \to \kappa \end{array}$$

which (up to changing the names of the variables) is the type with which *flip* is declared.

If you give a type declaration, it will be checked against the inferred type. Type declarations, as well as being good documentation, have the advantage of improving the explanation of any type errors found.

3.5 Type classes

Some functions (such as equality) are polymorphic, but not in the same regular way as parametric polymorphism. For example

```
> data UPair a = UPair a a
```

might represent unordered pairs, for which you would want $UPair\ x\ y$ to equal $UPair\ y\ x$. The mechanism for this *ad-hoc polymorphism* in Haskell is the type class. The type of equality is

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

which you can read as "provided a is an Eq-type, $a \to a \to Bool$". What does it take to be an Eq-type? A definition of `(==)`!

The concept of an Eq-type is introduced by a class declaration:

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

and a type such as $UPair\ a$ is put in that class by a matching instance declaration:

```
> instance Eq a => Eq (UPair a) where
>    (UPair p q) == (UPair r s)
>                     | p==r && q==s = True
>                     | p==s && q==r = True
>                     | otherwise    = False
```

or more succinctly

```
> instance Eq a => Eq (UPair a) where
>    (UPair p q) == (UPair r s) = (p==r && q==s) || (p==s && q==r)
```

The instance requires $Eq\ a$ in order to implement the equality tests on the components.

In fact the class declaration for Eq is more like

```
class Eq a where
  (==) :: a -> a -> Bool
  x == y = not (x /= y)
  (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

so that it also provides an inequality test. However an *instance* need only provide either an implementation of `(==)` or one of `(/=)` and the equation in the *class* declaration will define the other.

3.6 Recursive types

If the type being defined appears on the right hand side of a data definition, such as

```
> data List a = Nil | Cons a (List a)
```

it allows for recursive values: *Nil*, *Cons* 1 *Nil*, *Cons* 1 (*Cons* 2 *Nil*), ... are all of type *List Int*. In fact the built-in list type $[a]$ is like this, but with constructors [] and (:). The notation $[1, 2, 3]$ is shorthand for $1 : (2 : (3 : []))$ (and the parentheses can be left out since (:) is right-associative).

Functions over recursive types are also naturally defined by recursion. Functions over lists will often be naturally expressed by two equations: one for the empty case and one for the non-empty case, and the non-empty case will often include a recursive call of the same function.

Take for example

```
map f xs = [ f x | x <- xs ]
```

then you can calculate that

```
map f    []  = []
map f (x:xs) = f x : map f xs
```

and these two equations will do as a definition. Similarly the instance of the equality class for lists

```
instance Eq a => Eq [a] where
  []   ==   [] = True
  x:xs == y:ys = x == y && xs == ys
  _    ==    _ = False
```

includes equality on values of a (justified by the *Eq a* constraint in the header) and a recursive call of the equality test on lists.

The underlines are dummy arguments, which need not be named because their value are not used, names could be used, but the convention of the un-named underline is good documentation because the reader does not have to check that a named argument is in fact not used.