Parametric Polymorphism

Polymorphism is a general term describing how the same piece of code might behave differently depending on the arguments provided. The term usually refers to a function or operator call. There are fundamentally two different kinds of polymorphism:

Parametric Polymorphism refers to the situation where the same function code may act on values of different types but without changing the code. In that case the type of the corresponding value is a "parameter". A good example of this is functions operating on lists: The function `head` does not particularly care what type of values your list contains, only that it contains a list. So it can operate on *any list type*, and the content type of the list type is in effect a parameter.

In C++ parametric polymorphism is achieved via *templates*, and in Java via *generics* (`<T>`).

Ad-hoc Polymorphism refers to the situation where the same function symbol refers to multiple code chunks, and the decision on which code chunk to execute depends on the types of the arguments. A good example of this is the addition operation `+` or the equality operator `==`. Testing if two integers are equal requires a different code than testing if two strings are equal, yet they are both written the same way.

In C++ and Java you have probably encountered this as function/operator overloading, where the meaning of an expression like `a.add(b)` depends on the types of `a` and `b`, and different functions will be executed depending on those types. One can further say that object-oriented programming requires this kind of polymorphism in an essential way, to direct method calls to the correct place.

Almost every functional programming language implement parametric polymorphism in a similar way, via *type variables*. Support for ad-hoc polymorphism varies. Some languages don't have it at all, and those that do implement it in different ways. Haskell uses a clever concept called *type classes* that we will discuss in a moment.

Parametric Polymorphism: Type Variables

Some functions can do their job perfectly fine without needing to know precisely the type of value that they act on. A good example of that is the `tail` function for lists. A possible implementation for it could be simply:

```haskell
tail (x:xs) = xs
```

If Haskell tries to determine the type of this function, it will run into some trouble. It can tell that the input must be a list, and that the output must be a list *of the same type*. But it has no way of knowing what type of elements the list contains, nor does it care; it can do its job regardless.

In order to assign a proper type to this function, we must expand our type system. We don't want a drop function that only works on say [Int] values, nor one that only works on [Char] values. We need a function that only works on any [...] values.

This is where *type variables* come in. A **type variable** is essentially a variable used to represent a type in a function signature. These are lowercase and typically consist of only one letter, like a, b, t. When a function is being called, these types are **instantiated** for a particular type.

As an example, the type of the drop function would therefore be:

```
drop :: Int -> [t] -> [t]
```

This says that drop accepts as input a value of any list type, and returns a value of *the same list type*. When the function is actually used, like in

```
drop 1 [1, 2, 3]     -- t=Int.  Used as  drop :: Int -> [Int] -> [Int]
drop 1 "abc"         -- t=Char. Used as  drop :: Int -> [Char] -> [Char]
```

a specific type is chosen in place of the variable, for each use of the function. But the body of the function that is executed does not change.

If the body of the function does not provide any constraints on the types of some of the parameters, the function typically ends up with a parametric type (unless the programmer specified a more stringent type).

We can also have *multiple type variables*, if there are elements in the function that have arbitrary but different types. An example is the zip function. It takes two lists of elements and returns a new list by forming pairs from elements one from each list. It looks roughly like this:

```
zip (x:xs) (y:ys) = (x, y) : zip xs ys
```

In this case we don't care what the types of the two lists are, and they can be different from each other. But the type of the result list depends on them. We could write the type for zip thus:

```
zip :: [a] -> [b] -> [(a, b)]
```

Practice: Determine the types of the following functions:

1. head, tail.
2. take.
3. length.
4. fst. This function takes as input a tuple of two elements and returns the first element.