Polymorphic Types

One of the most powerful ideas in functional programming, and one that has really found its way in mainstream languages in the form of *generics*, is that of *polymorphic types*.

Parametric Polymorphism

In general "polymorphic" means "takes many shapes". In this context we are talking about functions. For example think of the function `hd` we wrote for picking out the first element of the list. That function should work just as well if the list contains integers, or floats, or strings, or even other lists. So any of the following could be the type of `hd`, depending on what items the list contains:

```
val hd: int list -> int
val hd: string list -> string
val hd: float list -> float
val hd: int list list -> int list
val hd: (int * int) list -> int * int
```

and so on. The function, and our code, does not really care what types of values there are, it works regardless. We can in essence consider the contained type as a *parameter* of our function. This is called **parametric polymorphism**.

In OCAML, and most functional programming languages, you can specify contained types like this with a "parameter type". Depending on the function's usage later on, that parameter type is *instantiated* to a more specific type.

In OCAML parameter types are specified with a single quote followed by lowercase letters, like `'a`, `'b` etc. So we could define our `hd` function via:

```
val hd: 'a list -> 'a
```

Similarly our `zip2` method from a few days ago could be written with type:

```
val zip2: 'a list * 'b list -> ('a * 'b) list
```

If we then write:

```
hd (zip2 ([1; 3; 4], ["hi", "there"]))
```

then in the evaluation `zip2` will be instantiated to have type `int list * string list -> (int * string) list`, so with `'a = int` and `'b = string`, while `hd` will be instantiated with type `(int * string) list`, so with `'a = int * string`. This instantiation happens automatically for us; OCAML guesses what the correct values are via the process of type inference that we will describe in the future. Java's generics this has to typically be specifically indicated in the code.

Ad-hoc Polymorphism

There is another kind of polymorphism, known as **ad-hoc polymorphism** is some circles, a special case of which is **operator overloading**. While in parametric polymorphism the actual function code that will be executed is essentially the same, in

ad-hoc polymorphism the system has to choose which function to associate to an identifier/symbol based on information like the parameter types.

This behavior is best exemplified in C++ streams. Think of the line:

```
cout << s;
```

What code is executed by the above line? That completely depends on the type/class of `s`. C++ will look at that class and search in the definition of that class for an implementation of <<. It is actually a bit more complicated, but that is the basic idea.

Haskell implements this feature in a restricted form via what are called *type classes*, that we will not get into at the moment. OCAML doesn't really have something analogous.

Parametric Types in Type Variants

Parametric types can be used in type variant definitions. For example we can literally define the option type as follows:

```
type 'a option = None | Some of 'a
```

Note how the contained type is specified as a "parameter" on the left side; it is part of the defined type's "name". With this one line we are effectively defining a whole lot of "concrete" types, like `int option`, `int option option`, `int list option`, `(int * int) option` and so on.

Except for the fact that we don't really know yet how to define operators, we could define "lists" via:

```
type 'a list = [] | :: of 'a * 'a list
```

This is not actually valid OCAML code, but it portrays the idea. We could definitely do:

```
type 'a list = Nil | Cons of 'a * 'a list
```

Then to form the list `[1; 3; 4]` we could write `Cons (1, Cons (3, Cons (4, Nil)))`. It's a bit ugly, but functional.

Language Design Considerations

All the above offers an interesting topic of conversation about language design. When you decide what features to put in a language, you have to decide how they will interact with each other, and what possibilities that creates. For example, if we allow type variants and parametric types into our language, then we can get for free both option types and list types, as well as a host of other useful examples. We do not need to offer each one of those as "special cases"; we just provide the building blocks for creating them and then add those base examples in a standard library that gets loaded in. A general principle in language design is to keep the core language *small* but with *orthogonal features* that can be combined in useful ways.