Enumerations and Sums

Michael P. Fourman

February 2, 2010

The ML type system controls access to data, ensuring that functions are only applied to arguments of the appropriate type. The basic types are `unit`, `bool`, `int`, `real`, and `string`. Product types, whose values are tuples or records allow us to group several data values as a single object. For example, we can make the type abbreviation

```
type Point = real * real;
```

However, the `type` declaration only introduces an abbreviation; it does not prevent us from applying a function, such as `Argand.++`, intended to add two complex numbers represented by polar coordinates to data-values using the cartesian representation, or even to a pair of reals representing the weight and volume of some solid object! So, type abbreviations allow us to confuse values that shouldn't be confused.

Sometimes, we *want* to apply the same function to objects with different representations. For example, in a drawing package, we might want to write a single function to move a variety of geometric shapes, with a variety of representations. The ML type system doesn't allow us to apply the same function to different types of object. So we seem to have the worst of both worlds: the type system allows us to make mistakes, but doesn't allow us to do what we want.

In this note, we introduce the ML `datatype` declaration. It provides a solution to both these difficulties. Consider the problem of implementing a type of geometric objects: circles, squares, lines, and rectangles. Here are some type abbreviations we might use to model these different types of object individually:

```
type Line   = Point * Point (* end-points                *)
type Circle = Point * real  (* centre and radius         *)
type Square = Point * real  (* top-left and side         *)
type Rect   = Point * Point (* top-left and bottom-right *)
```

This example exhibits both problems. First, we can all too easily pass lines off as rectangles, and squares as circles. For example, a function for drawing Circles could be applied to a data value representing a square. Second, because, for example, squares and rectangles are represented by different types, we cannot define a single drawing function that will take a geometric object, and draw it, or move it.

The ML `datatype` declaration allows us to define a single type of geometric objects

```
datatype Object =
         Line   of Point * Point (* end-points                *)
       | Circle of Point * real  (* centre and radius         *)
       | Square of Point * real  (* top-left and side         *)
       | Rect   of Point * Point;(* top-left and bottom-right *)
```

This declaration introduces a *new* type, `Object`, that can be used in the same ways as any existing type. It also introduces four *constructor* functions:

```
Line   : Point * Point -> Object
Circle : Point * real  -> Object
Square : Point * real  -> Object
Rect   : Point * Point -> Object
```

These can be used to construct values of the new type:

```
val origin     = (0.0,0.0)
val aPoint     = (1.5,2.7)
val unitCircle = Circle(origin, 1.0);
val aRectangle = Rect((1.0,3.1), aPoint)
val aLine      = Line(origin,aPoint)
```

We can use constructor functions in expressions, just like any other functions. However, we can also use constructors to form patterns that allow us to define functions on the new type:

```
fun area (Line _)       = 0.0
  | area (Circle (_,r)) = pi * r * r
  | area (Square (_,s)) = s * s
  | area (Rect ((top,left),(btm,right)) =
         (btm - top) * (right - left);
```

ML `datatype` declarations allow us to introduce a new type by saying how to construct values of that type. Any value of the new type must have been constructed by applying one of the constructor functions, so we can use pattern-matching to decide which constructor function was used, and to

decompose the value into its components.

We can picture the new type introduced by a datatype declaration like the one above as the disjoint union of the old types. This is called a *sum,* partly because the number of elements in the new type is the sum of the numbers in each of the constituent types.

Enumerations

A special case of the datatype declaration allows us to introduce constants of the new type, as in the following example

```
datatype Colour = Red | Brown | Yellow;
```

Here we introduce a new type `Colour`, and constants

```
Red    : Colour
Brown  : Colour
Yellow : Colour
```

Since these constants are constructors, we can use them in patterns

```
fun clash (Yellow, Brown) = true
  | clash (Brown,  Red)   = true
  | clash _               = false;
```

This example highlights the difference between constructors and other identifiers: they are treated quite differently in patterns. If we had made a slight mistake (using a lower-case `y` for `Yellow`) in typing the previous example,

```
fun clash (yellow, Brown) = true
  | clash (Brown,  Red)   = true
  | clash _               = false;
```

the effect of the function would be quite different. Since `yellow` has not been declared as a constructor, it is treated in the pattern as a variable, and can match any value of type `Colour`.

To minimise the risk of this kind of mistake, we normally give constructors an initial capital, and begin other identifiers in lower-case. This convention also helps the human reader to understand the programmer's intentions.

Add something here to explain the difference between constructor and type constraint. (This may not seem necessary, but there is much confusion on this point!

Type Safety

We can also use the datatype construction to introduce a new type, even when we don't want to form a sum, or enumeration type. For example, the declaration

```
datatype Point = Point of real * real;
```

creates a new type `Point` that cannot be confused with `real * real`. Formally, the constructor `Point` is a function that converts pairs of reals to values of type `Point`; although, in fact, an implementation will normally use the same internal representation for both types, and optimise away this function. However, to declare a function on such a type we must use the constructor in patterns:

```
fun distance (Point(x1,y1)) (Point(x2,y2)) =
    sqrt( square(x2 - x1) + square(y2 - y1) )
```

Summary

We have introduced three ideas

1. A datatype declaration *generates* a new type which can be used in the same way as any other ML type — for example in type constraints and in forming compound types. It is important to note that each time we repeat the "same" datatype declaration we produce a distinct new type. The point can be illustrated using the simplest possible datatype declaration.

   ```
   > datatype Example = Value ;
   datatype Example = Value
   > val a = Value ;
   val a = Value : Example
   > datatype Example = Value ;
   datatype Example = Value
   > val b = Value ;
   val b = Value : Example
   > a = b;
   Error Can't unify Example with Example
   Exception static_errors raised
   ```

   Here, the values bound to `a` and `b` have different types (which, confusingly, have the same name).

2. A datatype declaration introduces *constructors* — constants and functions that can be used to construct both, values of the new type, and functions whose result is a value of the new type.

3. Constructors allow us to write new patterns — the constant constructors, or formal application of constructors to patterns of the appropriate type. These allow us to declare functions that apply to arguments of the new type.

Later we will see that the datatype declaration can also be used recursively.