Products and Records

Michael P. Fourman

February 2, 2010

1 Simple structured types

Tuples

Given a value v_1 of type t_1 and a value v_2 of type t_2, we can form a pair, `(`v_1`, `v_2`)`, containing these values. The type of the pair is t_1 `*` t_2. Triples, quadruples etc are built in an analogous way. To decompose such values we extend the class of patterns to include tuple patterns, so that `(`*pat*$_1$`, ...,` *pat*$_n$`)` is now a pattern. Note the use of the pattern `(i,s)` in the following example:

```
fun f (i,s) = i + (size s);
```

This declares a function

$$\texttt{int} \times \texttt{string} \xrightarrow{\texttt{f}} \texttt{int}$$

Evaluating `f (4, "abc");` would produce the value `7`. All functions in SML take a single argument and return a single result. However, as the argument can be a tuple, and pattern matching allows us to break down a tuple into its components, syntactically it looks as though a function can take more than one argument.

Although pattern-matching is a convenient way to access the components of a tuple, sometimes it is useful to use functions that extract the components. The components of a pair may be extracted using the "selector functions", `#1`, and `#2`, shown in the following diagram.

$$\begin{tikzcd} A & A \times B \arrow[l, "\texttt{\#1}"'] \arrow[r, "\texttt{\#2}"] & B \end{tikzcd}$$

Our function, `f` could, equivalently, have been declared as

```
fun f (p: int * string) = #1 p + size (#2 p)
```

The type constraint is needed; without it, the compiler could not tell that the argument to `f` must be a pair. In this example, pattern-matching is clearer.

A function can also return more than one result, as in

```
fun g s = (size s, s);
```

If a function takes an argument of type t_1 and returns a result of type t_2, then the type of the function is displayed as t_1 `->` t_2. In the examples above, `f: int * string -> int`, and `g: string -> (int * string)`.

All values in ML are *first-class;* they can be passed as arguments, returned as results, and used as components of other, structured values. Pairs[1] are values and so can be manipulated just like the values of the basic types. As an example, consider the definition of an infix function `++`, for adding two vectors, component-wise.

```
infix ++;
fun ((vx, vy) ++ (wx, wy)) =
    (vx + wx : real, vy + wy : real);
```

This function takes a pair of pairs of reals as argument, and returns a pair of reals as result. If we use it to define a further function, summing three vectors, we don't need to decompose each vector into its components.

```
fun sumvecs (u, v, w) = u ++ v ++ w;
```

Records

The components of a tuple are unnamed — we retrieve a component by its position. SML also has a type, called a record, analogous to a C structure, or Pascal record. To build a record you must specify the field names and the corresponding values, as in

```
val point = {x = 3, y = 4, colour = "red"};
```

Such a value would have type `{x: int, y: int, colour: string}`. To decompose such values we extend the definition of a pattern to include record patterns. The order of the fields is unimportant, and so we could define

```
fun pointColour {colour = c, x = xp, y = yp} =
    c;
```

[1]This also applies to values of other types, that we will introduce later. It even applies to functions, even the built-in ones; they are also first-class values.

Applying this function to `point` would return the value `"red"`. We can also use a shorthand for record patterns, to introduce variables with the same names as the labels of the record.

```
fun pointInfo {colour, x, y} = (colour, x*x + y*y);
```

Applying `pointInfo` to `point` would give the value `(red, 25)`. Records provide documentation for code, but they make code more verbose and the "documentation" supplied is sometimes redundant. They may need to be handled by special-purpose functions, whereas corresponding functions on tuples could be placed in a library.

2 Functions and pattern matching

So far our elementary patterns have been restricted to variables and larger patterns constructed using just tuple and record patterns. We now look at other kinds of pattern. Consider the definition of a function that returns the first element of a pair:

```
fun fst (x, y) = x;
```

It's clearly important that we give a name to the first component of the pair, `x`, as we refer to it in the body of the function. Having to specify a name for the second component, in this case `y`, is tedious as it is never used. SML allows _ to be used as a *wild-card* pattern in these cases. This avoids forcing the programmer to think up pointless names, and indicates to the reader that this subcomponent is not referred to in the body. For example,

```
fun fst (x, _) = x;
fun snd (_, y) = y;
```

When we use pattern-matching against a pattern composed from variables and wild-cards, to decompose a value into subcomponents, the type system ensures that such a decomposition always succeeds. There are other cases where the pattern-matching can fail. The simplest situation where this can happen is if we allow constants such as integers, booleans and strings as patterns. For example, consider the following function declaration:

```
fun f ("abc", true, x) = x + 1;
```

The pattern only matches values of type `string * bool * int` where the first component is the string `"abc"` and the second component is `true`. What happens if you apply `f` to an argument that doesn't match this pattern? The system will raise an *exception* and return control to the top-level prompt. We will see later in the course how to explicitly raise and catch such exceptions. If a pattern doesn't match all values of the type it may be applied to then the system warns you of this with a message such as

```
Warning  Matches are not exhaustive.   Found near ...
```

The system also warns you if some patterns are redundant, in the sense that every possible argument will be matched by some earlier pattern. Both types of warning should be heeded — a well written, robust program will not generate warnings.

Definition by cases

Often it is useful to define a function by cases. For example, we might use strings to represent colours. We can then define a function to help choose our clothes by checking whether two colours clash. If the first colour is `"yellow"` and the second one is `"brown"` then we should (perhaps) return `true`. Similarly, we should return `true` if the colours are `"brown"` and `"red"`. Each one of the clashing choices can be expressed as a pattern on pairs. We extend the syntax for defining functions to allow a number of separate clauses (a clausal functional declaration):

$$\begin{array}{l} \textbf{fun}\ \langle \textit{function name} \rangle\ \langle \textit{pattern}_1 \rangle\ \texttt{=}\ \langle \textit{expression}_1 \rangle \\ \quad|\ \langle \textit{function name} \rangle\ \langle \textit{pattern}_2 \rangle\ \texttt{=}\ \langle \textit{expression}_2 \rangle \\ \quad\vdots \\ \quad|\ \langle \textit{function name} \rangle\ \langle \textit{pattern}_n \rangle\ \texttt{=}\ \langle \textit{expression}_n \rangle\texttt{;} \end{array}$$

A (tasteless) version of the `clash` function may now be defined by

```
fun clash ("yellow", "brown") = true
  | clash ("brown", "red") = true
  | clash _ = false;
```

When the function is applied to an argument the patterns are tried in sequence. If $\langle \textit{pattern}_i \rangle$ is the first one that matches then $\langle \textit{expression}_i \rangle$ is evaluated. Thus in the above example the third clause is only applicable if the first two don't match.

Variables and constants

It might seem inefficient to use strings to model colours, and the unwary might try the following definitions:

```
val red = 1;
val brown = 2;
val yellow = 3;

fun clash (yellow, brown) = true
  | clash (brown, red) = true
  | clash _ = false;
```

However, the occurrences of `yellow` and `brown` in the patterns will be treated as variables, not constants. The first clause will always match, binding the first colour to the *variable* `yellow` and the second element of the pair to the variable `brown`. This is an easy mistake to make, although in this simple case the compiler should warn you that the second and third clauses are redundant. We will see how to introduce *constructors,* new constants that *can* be used in patterns, later in the course when enumerated types are introduced.

Patterns are linear

Patterns, in SML, are said to be *linear.* This means that you can't repeat a variable in a pattern to indicate that two subcomponents are equal. Thus you must write

```
fun testPair (x, y) = if (x = y) then "same" else "different";
```

instead of

```
fun testPair (x, x) = "same"
  | testPair (_, _) = "different";
```

Layered patterns

Sometimes you would like to name a component of an argument *and* name some of its subcomponents. *Layered* patterns, using the keyword `as` to join two patterns that will both be matched against the same value, allow you to do this, as in

```
fun f (x as (y,_), _) = (x, (y,y));
```

Without layered patterns you would have to write

```
fun f ((y,z), _) = ((y,z), (y,y));
```

This definition is rather artificial. However, as the course progresses you will find many places where layered patterns arise naturally.

The `case` expression

A function may be used to discriminate between different forms of argument. SML also provides a **case** expression:

case $\langle expression\rangle$ **of**
    $\langle pattern_1\rangle$ `=>` $\langle expression_1\rangle$
  | $\langle pattern_2\rangle$ `=>` $\langle expression_2\rangle$
  $\vdots$
  | $\langle pattern_n\rangle$ `=>` $\langle expression_n\rangle$

Here is an example illustrating its use.

```
fun check {day, month, year} =
    let val leap = year mod 4 = 0
                   andalso
                   year mod 100 <> 0
        val daysInMonth = case month of
            "january"  => 30
          | "february" => if leap then 29 else 28
          (* ... *)
          | "december" => 31
          | _ => error "month not recognised"
     in
        day <= daysInMonth
     end;
```