Patterns, Tuples, and Lists

Torben Mogensen

September 21, 2016

These notes are supplementary to chapters 3 and 4 in Hansen & Rischel. We will focus on the *functional* style of programming.

1 Pattern Matching

We have already seen a recursive version of a function for computing the n'th Fibonacci version. Here is a variant of this that is defined only on non-negative numbers:

```
let rec fib n =
  if n = 0 then
    0
  elif n = 1 then
    1
  else
    fib (n - 1) + fib (n - 2)
```

We can test this for the first few integers by writing

```
for i = 0 to 10 do
  printfn "fib(%d) = %d" i (fib i)
```

which will yield

```
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
fib(9) = 34
fib(10) = 55
```

We used an if-then-else expression for special-casing on `n=0`, `n=1`, and everything else. An alternative to using if-then-else is to use *pattern matching*. The `fib` function can be written using pattern matching as

```
let rec fib n =
  match n with
  | 0 -> 0
  | 1 -> 1
  | m -> fib(m-1) + fib(m-2)
```

The match-with expression matches a value with any number of rules, where each rule is composed of a vertical bar (`|`), a pattern, an arrow (`->`) and an expression. A match-with expression first calculates the value of the expression between the `match` and `with` keywords to a value v and then applies the *first* rule that matches v. A rule `|` p `->` e matches v if the pattern p matches v. If p matches v, any variables in p are *locally* in e bound to the corresponding parts of v and the expression e is evaluated.

In the function above, `n` is evaluated to a value v. If $v = 0$, the first rule matches, and 0 is returned. If $v = 1$, the first rule does not match, but the second rule does, so 1 is returned. If v is neither 0 nor 1, neither the first nor the second rule matches. The pattern in the third rule is a variable. A variable pattern always matches, so the third rule will match. The variable `m` is bound to the the value v, and the expression after the arrow is evaluated. For example, if $v = 2$, the expression `fib(m-1) + fib(m-2)` is evaluated with the binding `m` $= 2$, so the results of calling `fib(1)` and `fib(0)` are added, giving the result 2, as expected.

Note that the order of rules is important. If the third rule is moved up before the two other rules, it will match all values, so the other two rules are never used.

So far, we have seen two kinds of pattern:

Constant patterns are identical to constant expressions. We have seen only integer constant expressions, but floating-point constants, string constants, and so on are all also valid patterns. A constant pattern will match exactly the value that the constant represents.

Variable patterns are variable names. A variable pattern will match any value and in the process bind the variable to the value locally in the rule. A special case is a *wildcard pattern*, which is written as an underscore (`_`). It also matches any value, but it does not bind any variables. In the third rule of the example above, we do not really need `m` to hold the value, as it is already contained in `n`, so we could equivalently write the third rule as

```
| _ -> fib(n-1) + fib(n-2)
```

Patterns can also be used when defining functions. The function definition

```
fun f 0 = 0
```

defines a function that given 0 returns 0 and is undefined on all other values. The compiler will give a warning saying that `f` is not defined on all values, but it will generate runnable code. Applying `f` to 0 will, as expected, return 0, but applying `f` to any other integer will result in an error message.

Using pattern matching instead of if-then-else can often lead to more readable programs, and the ability of patterns to bind variables to components of a value, such as the elements of tuples or lists, which we will see below.

2 Tuples

"Tuple" is a generic name for pairs, triples, quadruples, etc.

A *pair* is a combination of two values v and w. The pair is written as `(`v`,`w`)`, using a notation that should be familiar from coordinate pairs in math. v and w do not need to have the same type, so `(2,true)` and `("pi",3.14159)` are both valid tuples.

The type of a pair is written as the types of the components separated by `*`. So the two tuples above have types `int * bool` and `string * float`, respectively. The notation is similar to the math notation for cartesian products, where, for example, the set of pairs of two integers is written as $\mathbf{N} \times \mathbf{N}$. But since $\times$ is not a character in the ASCII character set, `*` is used instead.

Building a pair is easy: You just enclose the components in parentheses separated by a comma. So the function

```
let makePair x y = (x,y)
```

has type `makePair : x:'a -> y:'b -> 'a * 'b`. This states that `makePair` takes two arguments `x:'a` and `y:'b` and produces a pair of type `'a * 'b`. `makePair` can make pairs of any two types, so it is *polymorphic*. "Polymorphic" is derived from Greek and means "of multiple forms". In programming, we use the term for functions that have multiple types. You can read the type of `makePair` as "For any to types a and b, take a value x of type a and a value y of type b and return a pair of type a`*`b".

Getting the components of a pair is best done using pattern matching. For example, the function

```
let flip (x,y) = (y,x)
```

has the type `flip : x:'a * y:'b -> 'b * 'a`. It takes a pair (v, w) and returns the pair (w, v). Note that the pattern `(x,y)` matches any value of type `'a * 'b`, so there is no warning about values not being matched. The definition is equivalent to

```
let flip pair =
  match pair with
  | (x,y) -> (y,x)
```

or, using pattern matching in a let-binding,

```
let flip pair =
  let (x,y) = pair
  (y,x)
```

but when written as either of the above, the type is shown as `flip : 'a * 'b -> 'b * 'a`. The two types, though they look different, are the same, as naming components of types (such as `x:a`) do not change the type.

Note that a pattern can not contain the same variable more than once, so all of the following definitions are illegal:

```
let f x x = 1
let g (x, x) = 2
let h (x, (y, x)) = 3
```

Pairs can be components of other pairs, so `(3,(4,5))` is a pair of type `int * (int * int)`. The parentheses in the type are significant: The types `int * (int * int)`, `(int * int) * int`, and `int * int * int` are all different.

This brings us to *triples*. Where pairs are composed of two values, triples are composed of three values. Examples of triples (with types) are

```
               (3,4,5) :  int * int * int
      (true,"abc",3.14) :  bool * string * float
    ((1,2),(3,4),(5,6)) :  (int * int) * (int * int) * (int * int)
```

Building triples and using patterns to decompose triples is done exactly the same way as for pairs, except that there are three components. For example, a function that checks if a triple is a Pythagorean triple can be written as

```
let pythagorean (a,b,c) = a*a + b*b = c*c
```

which has type `pythagorean : a:int * b:int * c:int -> bool`.

We can also write a function for testing pythagorean numbers by using three separate arguments

```
let pythagorean' a b c = a*a + b*b = c*c
```

which has type `pythagorean' : a:int -> b:int -> c:int -> bool`. If you use separate arguments, you can partially apply the function, so

```
let p34 = pythagorean' 3 4
```

is a value `p34 : (int -> bool)`, which takes a single argument c and tests if $3*3+4*4=c*c$. On the other hand, you can apply the `pythagorean` function to a value that is already a triple:

```
let triple = (3,4,5)
pythagorean triple
```

and if you want to return three values from a function, a triple is the obvious way to do so.

Pairs and triples extend to tuples with four, five or more components in the obvious way.

3 Lists

A list is a sequence of values that have the same type, but where the length of the sequence is not specified in the type. This contrasts with tuple types, where the components can have different types, but the number of components is specified in the type. A list is written as a sequence of elements separated by semicolons and enclosed in square brackets. For example, a list of four numbers can be written as

`[3; 1; 4; 1]` and has type `int list`, indicating that it is a list of integers. The elements can be of any type, as long as the types are consistent, so the following are all legal lists:

```
['a'; 'e'; 'i'; 'o'; 'u'; 'y'] :  char list
                        [true] :  bool list
[(3, 4, 5); (5, 12, 13); (8, 15, 17)] :  (int * int * int) list
  [[]; [1]; [1; 2]; [1; 2; 3]] :  int list list
                            [] :  'a list
```

Note that the last example – the empty list – has type `'a list` because it is an empty list of any type of element. Note, also, the parentheses in the third example. These are required because the `list` type constructor binds more tightly than the `*` type constructor. The type `int * int * int list` is equivalent to `int * int * (int list)` and would have elements like `(1, 2, [5; 6])`. Similarly, `int list list` is equivalent to `(int list) list`.

Just as with strings, you can use dot-notation to get to elements or sublists of lists:

```
['a'; 'e'; 'i'; 'o'; 'u'; 'y'].[2] = 'i'
['a'; 'e'; 'i'; 'o'; 'u'; 'y'].[2..4] = ['i'; 'o'; 'u']
['a'; 'e'; 'i'; 'o'; 'u'; 'y'].[2..14] = error message
```

In fact, a list of characters is very similar to a string, but they are separate types.

You can concatenate two lists using the `@` operator. Compare

```
['a'; 'e'; 'i'] @ ['o'; 'u'; 'y'] = ['a'; 'e'; 'i'; 'o'; 'u'; 'y'] :  char list
                  "aei" + "ouy" = "aeiouy" :  string
```

The function `List.length` finds the length of a list. Compare

```
List.length ['a'; 'e'; 'i'; 'o'; 'u'; 'y'] = 6
            String.length "aeiouy" = 6
```

You can also pattern match on lists. For example, we can write this function that finds the sum of elements of integer lists of length up to three:

```
let sum ns =
  match ns with
  | [] -> 0
  | [n1] -> n1
  | [n1; n2] -> n1 + n2
  | [n1; n2; n3] -> n1 + n2 + n3
```

This is easily (albeit verbosely) extended to list up to any fixed length, but if we want to handle lists of arbitrary length, we use the `::` constructor, also called the *cons* operator.

The `::` constructor can be used as an infix operator both in expressions (to build lists) and in patterns (to match against lists), but unlike other infix operators (such as `+` or `@`) it is not a function, so writing `(::)` will give an error message. Used in an expression, `::` takes on its left-hand side a value x of any type a and on its right-hand side a list xs of type a `list` and will produce a list of type a `list` by adding x as an element in front of xs. Examples:

```
'a' ::  ['e'; 'i'; 'o'; 'u'; 'y'] = ['a'; 'e'; 'i'; 'o'; 'u'; 'y']
                        1 ::  [] = [1]
 1 ::  2 ::  3 ::  [] = [1; 2; 3] =
```

Note that `a ::  b ::  c` is equivalent to `a ::  (b ::  c)`, so `::` is *right associative*: multiple applications of `::` are implicitly grouped to the right.

The expression `x :: xs` is equivalent to `[x] @ xs`. But where `@` can not be used in patterns, `::` can. Typically, a function that operates on a list will be recursive and use a match-with expressions with one rule for the empty list and one rule for the non-empty list:

```
let rec listSum ns =
  match ns with
  | [] -> 0
  | n :: ns -> n + listSum ns
```

The first pattern `[]` matches the empty list, which has sum 0. The second pattern matches a list with at least one element, and binds `n` to the first element of the list (also called the *head* of the list) and `ns`

to the rest of the list (also called the *tail* of the list). Note that the variable name `ns` is rebound locally within the second rule. Application of the `listSum` function to an argument can be illustrated by the following reduction sequence:

```
listSum [1;4;9;16]
                ⇝  1 + listSum [4;9;16]
                ⇝  1 + 4 + listSum [9;16]
                ⇝  1 + 4 + 9 + listSum [16]
                ⇝  1 + 4 + 9 + 16 + listSum []
                ⇝  1 + 4 + 9 + 16 + 0
                ⇝  30
```

Note that a function equivalent to `listSum` is predefined in F# as `List.sum`, but where `listSum` only works on lists of integers, `List.sum` works on lists of floats or other number types as well.

Let us consider writing a function that tests if a list is sorted. An empty list is always sorted, and so is a list consisting of exactly one element. If a list has two or more elements, it is sorted if the first element is less than or equal to the second element and the rest of the list (starting from the second element) is also sorted. We can write this as

```
let rec isSorted ns =
  match ns with
  | [] -> true
  | [n] -> true
  | n1 :: n2 :: ns -> n1 <= n2 && isSorted (n2 :: ns)
```

This has the type `isSorted :  ns:'a list -> bool when 'a :  comparison`. This is because the `<=` operator is *overloaded* and works on several (but not all) types. The set of types where `<=` is defined are categorised by the pseudo-type `comparison`, so `'a :  comparison` means "any type `'a` where `<=` is defined". So we can use `isSorted` on many, but not all, list types, for example:

```
     isSorted [2; 1; 3; -1] = false
isSorted ["Cc"; "ab"; "c"] = true
 isSorted [1e-2; 1.0; 1e0] = true
```

We note that we rebuild the list from the second element onwards by writing `n2 :: ns` in the recursive call. This is rather inefficient, so an alternative is to use *as-pattterns*. An as-pattern is a pattern p followed by the keyword `as` and a variable x. The as-pattern matches if p matches but in addition to binding the variables in p to the corresponding components, it binds x to the entire value. We can write `sorted` using an as-pattern by

```
let rec isSorted ns =
  match ns with
  | [] -> true
  | [n] -> true
  | n1 :: (n2 :: _ as ns) -> n1 <= n2 && isSorted ns
```

The pattern `n2 :: _ as ns` matches a non-empty list v and binds `ns` to v, `n2` to the head of v. We don't need the tail of v, so we use a wildcard pattern.

In a similar way, we can write a function that tests if all the lists in a list of lists have the same length:

```
let rec sameLength xs =
  match xs with
  | [] -> true
  | [x] -> true
  | x1 :: (x2 :: _ as xs)-> List.length x1 = List.length x2 && sameLength xs
```

which has type `sameLength :  xs:'a list list -> bool`.

While we by using the as-pattern have avoided rebuilding the list, we compute the length of most lists twice. To avoid this, we can use a helper-function that compares the length of all lists in a list of lists to the length of the first list:

```
let sameLength xs =
  match xs with
```

```
  | [] -> true
  | x1 :: xs ->
      let lx1 = List.length x1
      let rec sameLengthAs xs =
        match xs with
        | [] -> true
        | x1 :: xs -> List.length x1 = lx1 && sameLengthAs xs
      sameLengthAs xs
```

Note that both versions of `sameLength` compute the same function, the only difference is efficiency, and that difference is less than a factor of two.

As a slightly more complex example, let us consider a function that finds the largest element of a list of values. This function is not defined on the empty list, as an empty list does not have a largest element, so we write the function as

```
let rec largest ns =
  match ns with
  | [n] -> n
  | n :: ns -> max n (largest ns)
```

The F# compiler will complain that the pattern matching is incomplete, as it does not match the empty list, but it will compile it to a function with type `largest : ns:'a list -> 'a when 'a : comparison`. Note that the function is not restricted to lists of integers, but can be applied to, for example, lists of floats or lists of strings. This is because the `max` function used in the last line is overloaded and works on all types where comparison using `<=` is defined, just as we saw for the function `isSorted` earlier.

Applying `largest` to an empty list will yield a rather cryptic error message:

```
error FS0030: Value restriction. The value 'it' has been inferred to have generic type
    val it : '_a when '_a : comparison
Either define 'it' as a simple data term, make it a function with explicit arguments or,
if you do not intend for it to be generic, add a type annotation.
```

The reason for this error message is rather complicated, so we will ignore it for now. The interested reader can see Section 4.3 of Hansen & Rischel.

4 Patterns Revisited

We summarise the forms of patterns we have seen so far. A pattern can be:

- A constant in a type that is comparable be equality. A constant pattern matches any value that tests for true with equality with the constant. No variables are bound.
- A variable. This matches any value and binds the variable to that value.
- An underscore (`_`). This *wildcard pattern* matches anything, but does not bind anything.
- A tuple pattern of the form $(p_1, \ldots, p_n)$. This matches a tuple value $(v_1, \ldots, v_m)$ if $m = n$ (which is verified by the type checker) and p_i matches v_i for all $i \in \{1, \ldots, n\}$. Variables can be bound when matching each p_i to v_i, and all bindings are in scope in the rule that uses the pattern. No variable may be repeated in the pattern.
- A list pattern of the form $[p_1; \ldots; p_n]$. This matches a list value $[v_1; \ldots; v_m]$ if $m = n$ and p_i matches v_i for all $i \in \{1, \ldots, n\}$. The same rules for binding and repeated variables that apply to tuple patterns also apply to list patterns.
- A constructor pattern. We have seen the nullary constructor pattern `[]` and the infix constructor pattern p_1 `::` p_2. Generally, a constructor pattern is either a nullary constructor (a constructor that does not take arguments) or a constructor applied to a single argument which may be a tuple or list pattern. An infix constructor pattern is equivalent to a constructor applied to a pair pattern. A constructor pattern matches if the value is constructed by the same constructor and the argument of that constructor (if any) match the argument pattern.

- An as-pattern. This is of the form p `as` x, where p is a pattern and x is a variable. It matches a value v if p matches v, and x is bound to v in addition to any bindings made when matching p to v. x may not occur in p.

- Operator precedence and parentheses can effect grouping of nested patterns as they affect nested expressions or types. Operators in patterns have the same precedence as in expressions.

We have seen patterns used in the following contexts:

- In match-with expressions, where patterns are used in rules of the form `|` p `->` e. A warning is given if the set of rules do not cover all possible values allowed by the type of the matched expression, and if at runtime a value is given that does not match any rule, an error is reported.

  A warning is also given if a rule can not be reached because all values matched by that rule are already matched by earlier rules. This can not cause runtime errors, but if you get this error, don't just delete the rule that can never me reached. Instead, think of *why* it can not be reached, and fix the problem with this understanding in mind.

- In function definitions, where a pattern can be used instead of an argument variable. If the pattern does not cover all values allowed by the argument type, a warning is given, and if at runtime an argument is given that does not match the pattern, an error is reported.

- In let-definitions, where a pattern can be used instead of a variable. Again, warnings and errors are reported if the pattern does not cover all possible values.

Additionally, rules in the style of match-with can be used in an alternative notation for function definition. The `fib` function shown in Section 1 can alternatively be written as

```
let rec fib = function
  | 0 -> 0
  | 1 -> 1
  | m -> fib(m-1) + fib(m-2)
```

You can read the notation as "`fib` is a function that maps 0 to 0, 1 to 1, and any other value m to `fib`$(m-1)+$`fib`$(m-2)$".

Note that the argument to `fib` is not named, it is implicitly given to the rules shown after the `function` keyword.

Similarly, `listSum` from Section 3 can be written as

```
let rec listSum = function
  | [] -> 0
  | n :: ns -> n + listSum ns
```

The book by Hansen & Rischel uses the `function` notation by default.