Functions as Values and Collections

Torben Mogensen

October 3, 2016

These notes are supplementary to chapters 2 and 5 in Hansen & Rischel.

1 Functional Values and Types

In F#, a function is a value that can be used in the same way as numbers, strings, booleans and so on: You can pass functions as arguments to other functions, return functions as results, build lists of functions, and so on. Just about the only things you can not do with a function that you can do with numbers, strings, and other values are:

- You can not compare functions. If you define a function `f` and afterwards write `f=f`, you get an error saying that equality is not defined on function types. Similarly for inequality (`<`, `>`, and so on).
- You can not print a function to see its definition. If you write `printfn "%A" f`, where `f` is a function, F# will write something like `<fun:it@2>`, and it will even be different strings if you print the function several times. For example, `printfn "%A %A" f f` can print `<fun:it@8-6>` `<fun:it@8-7>` or something similar.

To illustrate functions as values, let us define

```
let f x = x + 1
let g y = y * 2
let h z = z * z + 2 * z + 3
let fl = [f; g; h]
```

`fsharpi` will show the types of these definitions:

```
val f : x:int -> int
val g : y:int -> int
val h : z:int -> int
val fl : (int -> int) list = [<fun:fl@5>; <fun:fl@5-1>; <fun:fl@5-2>]
```

and for `fl` it will even print its value, though all you can see of the value is that it is a list of three functions. You can't even see that if they are the same function three times or three different functions.

The types `x:int -> int`, `y:int -> int`, `z:int -> int`, and `int -> int` are the same. The name of the parameter shown in the types for `f`, `g`, and `h` is not actually a part of the type, which is just `int -> int`, as shown in the list type.

We can pick an element from the list and apply it, for example, `fl.[2] 5`, which will return the value 38.

When you define a function of two arguments, for example,

```
let plus x y = x + y
```

it has a type with two arrows, in this case `int -> int -> int` (with some naming or arguments that we ignore). This type is equivalent to `int -> (int -> int)`, which says that when you give `plus` one integer argument, it will return a function of type `int -> int`. This can later be applied (once or several times) to another integer argument to give an integer result, for example,

```
let plus5 = plus 5
(plus5 3) * (plus5 7)
```

which will give the value 96.

A function can also accept functions as arguments, so we can define

```
let applyTwice f x = f (f x)
```

which defines a function `applyTwice` that takes a function `f` and applies it twice to an argument `x`. For example, `applyTwice plus5 3` will return 13.

2 Anonymous Functions

When you make a definition

```
let plus x y = x + y
```

it is actually an abbreviation for

```
let plus = fun x y -> x + y
```

which you can read as "let `plus` be a function that takes two arguments, `x` and `y`, and returns `x+y`". This also works with recursive functions, so

```
let rec gcd a b = if b = 0 then a else gcd b (a % b)
```

is an abbreviation for

```
let rec gcd = fun a b -> if b = 0 then a else gcd b (a % b)
```

You do not need to give names to functions defined using `fun`, so you can, for example, write

```
[(fun x -> x + 1); (fun y -> y * y); (fun z -> 2 * z + 3)]
```

to define a list of three functions. Note that the parentheses are needed to avoid F# treating the semicolons as part of the function instead of separating these. This is because seimicolon has higher operator precedence than `->`. If in doubt, always enclose anonymous function definitions in parentheses.

Anonymous function definitions are useful if you want to pass a small function as argument to another function. For example, with `applyTwice` as defined above, we can make the call `applyTwice (fun x -> x*x) 7` to get 2401.

You can also make anonymous pattern-matching functions. When you write a definition like

```
let isEmpty = function
  | [] -> true
  | n :: ns -> false
```

this first defines an anonymous function (using the keyword `function` followed by a list of rules) and then binds the name `isEmpty` to that function. Note that a function defined with `function` can use pattern matching with multiple rules, but it can take only one argument. In contrast, a function defined with `fun` can take multiple arguments, but only use one rule.

An infix operator written in parentheses is a function of two arguments. For example: `(+) : int -> int -> int`, so you can make a function like `plus5` above by simply writing `((+) 5)`, which has type `int -> int`. Note that `::` can not be made into a function this way. You have to write `(fun x y -> x :: y)` or similar.

3 Selected Functions from the `List` Library

In addition to simple functions such as

```
List.length : 'a list -> int
List.head : 'a list -> 'a
List.isEmpty : 'a list -> bool
```

the `List` library also defines a number of useful functions that take functional arguments in addition to list arguments. By using these library functions, you can avoid having to define recursive functions that use pattern matching. Here are some examples:

- `List.map : ('a -> 'b) -> 'a list -> 'b list`

  takes a function f and a list xs and constructs a new list by applying f to all elements of xs. For example,

  ```
  List.map (fun x -> x*x) [1..5]
  ```

  will produce the result `[1; 4; 9; 16; 25]`.

- `List.map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list- > 'c list`

  takes a function f and two lists xs and ys and constructs a new list by applying f to all pairs constructed by taking corresponding elements of xs and ys. For example,

  ```
  List.map2 (fun x y -> x * x + y) [1..5] [3..7]
  ```

  will produce the result `[1*1+3; 2*2+4; 3*3+5; 4*4+6; 5*5+7]` $\rightsquigarrow$ `[4; 8; 14; 24; 32]`.

  If the two lists are not of the same length, an error message will be given.

- `List.exists : ('a -> bool) -> 'a list -> bool`

  takes a predicate p and a list xs and checks if there is an element in xs for which p returns `true`. For example, if you want to know if the value 7 is found in a list `xs`, you can write

  ```
  List.exists (fun x -> x = 7) xs
  ```

- `List.forall : ('a -> bool) -> 'a list -> bool`

  is similar to `List.exists`, but returns `true` only if the predicate returns `true` for *all* elements of the list.

  For example,

  ```
  List.forall (fun x -> x = 7) xs
  ```

  returns `true` if all elements in `xs` are equal to 7. If `xs` is empty, this is trivially true.

- `List.find : ('a -> bool) -> 'a list -> 'a`

  takes a predicate p and a list xs and returns the first element in xs for which p returns `true`. If there are none, an error is reported.

  For example,

  ```
  List.find (fun (x, y) -> y = "Emil")
            [(1, "Joachim"); (2, "Sune"); (3, "Emil"); (4, "Matthias")]
  ```

  will return `(3, "Emil")]`.

- `List.tryFind : ('a -> bool) -> 'a list -> 'a option`

  takes a predicate p and a list xs and, if there is an element in xs for wh which p returns `true` returns `Some` x, where x is the first element in xs for which p returns `true`. If there are no such elements, it returns `None`. Option-typen er beskrevet i Hansen & Rischel afsnit 3.11.

  For example,

  ```
  List.tryFind (fun (x, y) -> y = "Emil")
            [(1, "Joachim"); (2, "Sune"); (3, "Emil"); (4, "Matthias")]
  ```

  will return `Some (3, "Emil")]`, and

  ```
  List.tryFind (fun (x, y) -> y = "Hans")
            [(1, "Joachim"); (2, "Sune"); (3, "Emil"); (4, "Matthias")]
  ```

will return `None`.

- `List.filter :  ('a -> bool) -> 'a list -> 'a list`

  takes a predicate p and a list xs and returns all elements in xs for which p returns `true`.

  For example,

  ```
  List.filter (fun (x, y) -> y < "Klaus")
            [(1, "Joachim"); (2, "Sune"); (3, "Emil"); (4, "Matthias")]
  ```

  will return `[(1, "Joachim"); (3, "Emil")]`.

- `List.init :  int -> (int -> 'a) -> 'a list`

  is used to construct new lists. The call `List.init` $n\, f$ will construct the list `[`$f\,0$`;` $f\,1$`;`$\ldots$`;`$f\,(n{-}1)$`]`.

  Generally, `(List.init` $n\, f$`).[`i`]` will be $f\,i$ for $0 \le i < n$.

  For example,

  ```
  List.Init 5 (fun x -> x * x)
  ```

  will return `[0; 1; 4; 9; 16]`.

- `List.foldBack :  ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`

  A call `List.foldBack` f `[`a_1`;...;` a_n`]` b does the following:

  1. Views the list `[`a_1`;` a_2`;...;` a_n`]` as the equivalent
     a_1 `::` $(a_2$ `:: ...(`a_n `:: []) ...)`.
  2. Replaces a_i `::` with $f\ a_i$ and `[]` with b, yielding
     $f\ a_1\ (f\ a_2\ \ldots(f\ a_n\ b)\ \ldots)$.
  3. Calculates the result.

  For example,

  ```
  let f x y = x * y
  List.foldBack f [2; 3; 4] 1
  ```

  first sees `[2; 3; 4]` as

  `2 :: (3 :: (4 :: []))`,

  then rewrites this to

  `f 2 (f 3 (f 4 1))`,

  which is evaluates to `2 * 3 * 4 * 1` $\rightsquigarrow$ 24.

  It is easier to understand the behavior if f is an infix operator. For example,

  ```
  List.foldBack (*) [2; 3; 4] 1
  ```

  sees the list as

  `2 :: (3 :: (4 :: []))`,

  then replaces `::` by `*` and `[]` by `1`, giving

  `2 * (3 * (4 * 1))`,

  which evaluates to 24, as above.

  So we can define the factorial function and the length function as

  ```
  let fac n = List.foldBack (*) [2..n] 1
  let length xs = List.foldBack (fun x y -> y + 1) xs 0
  ```

The factorial function multiplies the elements in the list `[2..n]`, using `1` as a starting point. The length function adds 1 for every element in the list, ignoring the actual elements (by not using `x`).

We can define `foldBack` as a recursive function:

```
let rec foldback f xs b =
  match xs with
  | [] -> b
  | x :: xs -> f x (foldBack f xs b)
```

It is easy to see that this precisely replaces `[]` by `b` and every `x ::` by `f x`.

- `List.fold : ('b -> 'a -> 'b) -> 'b -> 'a list -> 'b`

  This function is similar to `List.foldBack`, but groups the elements differently:

  `List.fold` f b `[`a_1`;` a_2`;...;` a_n`]` reduces to $(f \ldots (f\,(f\,b\,a_1)\,a_2) \ldots a_n)$.

  If f is an infix operator, this is easier to see. For example, `List.fold (+) 0 [1; 2; 3]` reduces to `(((0 + 1) + 2) + 3)` $\rightsquigarrow$ 6.

  We can define `fold` recursively using an accumulating parameter `b`:

```
let rec fold f b xs =
  match xs with
  | [] -> b
  | x :: xs -> fold f (f b x) xs
```

  Note that, if f is an associative and commutative infix operator (like `+` or `*`), `List.fold` f b xs $=$ `List.foldBack` f xs b. For example, `(((0 + 1) + 2) + 3)` $=$ `1 + (2 + (3 + 0))`. So we can also define the factorial function function by

```
let fac n = List.fold (*) 1 [2..n]
```

4 Arrays

Arrays are similar to lists: Sequences of elements of the same type, where the length of the sequence is unspecified by the type. An array with elements of type a is written as a `[]`, for example `int []` for an array of integers.

Some similarities between lists and arrays can be seen in the table below:

List expressions	Array expressions
`[1; 2; 3] : int list`	`[|1; 2; 3|] : int []`
`[] : 'a list`	`[||] : 'a []`
`List.head [1; 2; 3]` $\rightsquigarrow$ `1`	`Array.head [|1; 2; 3|]` $\rightsquigarrow$ `1`
`List.tail [1; 2; 3]` $\rightsquigarrow$ `[2; 3]`	`Array.tail [|1; 2; 3|]` $\rightsquigarrow$ `[|2; 3|]`
`[1; 2; 3].[2]` $\rightsquigarrow$ `3`	`[|1; 2; 3|].[2]` $\rightsquigarrow$ `3`
`[1; 2; 3].[1..2]` $\rightsquigarrow$ `[2; 3]`	`[|1; 2; 3|].[1..2]` $\rightsquigarrow$ `[|2; 3|]`
`List.length [1; 2; 3]` $\rightsquigarrow$ `3`	`Array.length [|1; 2; 3|]` $\rightsquigarrow$ `3`
`List.map f [1; 2]` $\rightsquigarrow$ `[f 1; f 2]`	`Array.map f [|1; 2|]` $\rightsquigarrow$ `[|f 1; f 2|]`
`List.filter odd [1; 2; 3]` $\rightsquigarrow$ `[1; 3]`	`Array.filter odd [|1; 2; 3|]` $\rightsquigarrow$ `[|1; 3|]`
`[1; 2] @ [3]` $\rightsquigarrow$ `[1; 2; 3]`	`Array.append [|1; 2|] [|3|]` $\rightsquigarrow$ `[|1; 2; 3|]`

List patterns	Array patterns
`[x; y; z]`	`[|x; y; z|]`
`[]`	`[||]`

Generally, almost all the library functions from the `List` library (`init`, `exists`, `find`, `foldBack`, ...) are also found in the `Array` library with equivalent behaviour. The main differences between lists and arrays are:

- The `::` operator and pattern is not defined for arrays.

- Array elements are *mutable*, so you can write, for example

  ```
  let aa = [|1..5|]
  aa.[2] <- 17
  printfn "%A" aa
  ```

  which will print `[|1; 2; 17; 4; 5|]`.

- Indexing into an array a to get an element using $a.[i]$ is faster than indexing into a list l using $l.[i]$, unless $i < 2$.

- Appending two arrays has a cost proportional to their combined size, where appending two arrays has a cost proportional only to the first argument to `@`.

- Using `::` to add an element to a list or split a list into head and tail, or using `List.head` and `List.tail` to split a list is *very* cheap compared to using `Array.append` to add an element to a list and `Array.head` and `Array.tail` to split an array into head and tail.

- There is a function `Array.create : int -> 'a -> 'a []` such that `Array.create` n v creates an array of n elements all equal to v.

You can say that lists are optimised for recursive functions using `[]` and `::` patterns or `List.head` and `List.tail`, while arrays are optimized for indexing using a`.[`i`]`. Library functions like `map`, `filter`, `fold` and `init` have roughly similar cost for arrays and lists.

It is perfectly possible (and quite efficient) to work with arrays without using assignment (the `<-` operator) by using `Array.init`, `Array.fold`, and so on, but it is also quite common to use loops and assignment when programming with arrays. For example, we can make a function that prints all primes up to a number n this way:

```
let primesUpto n =
  let prime = Array.create (n+1) true
  for p in 2 .. int (sqrt (float n)) do
    if prime.[p] then
      for i in (p*p) .. p .. n do
        prime.[i] <- false
  for i in 2 .. n do
    if prime.[i] then printfn "%d" i
```

5 Two-dimensional Arrays

Two-dimensional arrays are just that: Rows and columns of elements. The type a `[,]` denotes a two-dimensional array with elements of type a. The corresponding library is called `Array2D` and has equivalents to many of the functions found in the `Array` library, but slightly different since we work in two dimensions.

For example, `Array2D.init 3 5 (fun x y -> 10*x+y)` creates the two-dimensional array

```
[[0; 1; 2; 3; 4]
 [10; 11; 12; 13; 14]
 [20; 21; 22; 23; 24]]
```

Note that the notation looks like a list of lists, and typing in the above in `fsharpi` will, indeed, produce something of type `int list list`. There is no notation for entering a constant two-dimensional array, but you can use the function `array2D` to convert a list of lists into a two-dimensional array.

Writing

```
[|[|0; 1; 2; 3; 4|]
  [|10; 11; 12; 13; 14|]
  [|20; 21; 22; 23; 24|]|]
```

produces an array of arrays, i.e., something of type `int [] []`.

The differences between a two-dimensional array and an array of arrays are subtle:

- When indexing into a two-dimensional array $a2$, you use the notation `a2.[i,j]`, where you use `aa.[i][j]` to index into an array of arrays aa.

-

- In a two-dimensional array, all the rows have the same length, where an array of arrays can have rows of different length.

The following functions from `Array2D` are of interest:

- `Array2D.init : int -> int -> (int -> int -> 'a) -> 'a [,]`

  We have already seen this in the example above. The call `Array2D.init` i j f creates a two-dimensional array with i rows and j columns where the element at position (i, j) i sgiven by $f(i, j)$.

- `Array2D.length1 : 'a [,] -> int`

  Returns the number of rows in a two-dimensional array.

- `Array2D.length2 : 'a [,] -> int`

  Returns the number of columns in a two-dimensional array.

6 Further reading

The F# language reference at Mircrosoft has a page about one- and two-dimensional arrays that also explains how to take whole rows or columns out of a two-dimensional arrays:
`https://docs.microsoft.com/en-us/dotnet/articles/fsharp/language-reference/arrays`