More About Higher-Order Functions

Björn Lisper
School of Innovation, Design, and Engineering
Mälardalen University

bjorn.lisper@mdh.se
http://www.idt.mdh.se/˜blr/

Currying (what Functions of Several Arguments Really are)

Remember `simple`?

A function of three variables, we said:

```
simple : int -> int -> int -> int
let simple x y z = x*(y + z)
```

But in F#, a function *only takes one argument*!

What's up?

`simple x y z` means `((simple x) y) z` (function application is left associative)

`int -> int -> int -> int` means
`int -> (int -> (int -> int))`

Thus, `simple` is a function in one argument, returning a function of type

`int -> (int -> int)`

which returns a function of type

`int -> int`

which returns an `int`!

Encoding functions with several arguments like this is called `currying` (after Haskell B. Curry, early logician)

We *could* have defined:

```
simple : (int * int * int) -> int
let simple (x,y,z) = x*(y + z)
```

Another way to represent a function of three arguments, as a function taking a 3-tuple

But it is not the same function – it has different type!

This version may seem more natural, but the curried form has some advantages

Currying and Syntactical Brevity

What is `simple 5`?

A function in two variables (say `x`, `y`), that returns `5*(x + y)`

We can use `simple 5` in *every place where a function of type* `int -> (int -> int)` *can be used*

Direct Function Declarations

A declaration

```
let f x = g x
```

where `g` is an expression (of function type) that does not contain `x`, can be written

```
let f = g
```

“The function `f` equals the expression `g`”, not stranger than “scalar” declarations like `let pi = 3.154159`

A First Example

Recall `sum` (and all the other functions defined by folds):

```
let sum xs = List.fold (+) 0 xs
```

Same as

```
let sum xs = (List.fold (+) 0) xs
```

Both `sum` and `List.fold` have `xs` as last argument (and nowhere else)

It can then be “cancelled”:

```
let sum = List.fold (+) 0
```


A Second Example

A function that reverses a list

We first make a “naïve” recursive definition, which is inefficient; then a better recursive definition; then we redo the second definition using higher order functions, and finally we make the declaration as terse as possible

(Solutions on next slide and onwards)

Reverse: First Attempt

Idea: put the first element in the list last, then recursively reverse the rest of the list and put in front. Reverse of the empty list is empty list.

```
let rec reverse l =
  match l with
  | []    -> []
  | x::xs -> reverse xs @ [x]
```

This definition of `reverse` is correct, but has a performance problem. What problem? (Answer on next slide)

Reverse: Problem with First Attempt

This definition uses `List.append` (`@`) with long first arguments

If the list to reverse has length n, then `List.append` will be called with first argument of length $n-1, n-2, \ldots, 1$

Time to run `List.append` is proportional to length of first argument

Thus, the time to run `reverse` is $O((n-1)+(n-2)+\cdots+1) = O(n^2)$

Grows quadratically with the length of the list!!

Can we do better?

(Yes. . . solution on next slide)

A More Efficient Reverse

We use the “stack the books” principle, with an accumulating argument:

```
let reverse xs =
  let rec rev1 acc xs =
    match xs with
    | []    -> acc
    | x::xs -> rev1 (x::acc) xs
  in rev1 [] xs
```

This definition uses n recursive steps

In each step, the amount of work is constant

Thus, the time to reverse the list is $O(n)$ – big difference to $O(n^2)$ when n grows large!

Higher-Order reverse

The main operation of the efficient `reverse` is to put an element in a list, which is accumulated in an argument

Can we define a binary operation and use, say, `List.fold` to define `reverse` (or `rev1`)?

Let's line up their definitions:

```
let rev1 acc xs =
  match xs with
  | []    -> acc
  | x::xs -> rev1 (x::acc) xs

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

Hmmm, an operation `revOp` such that `revOp acc x = x::acc`?

Here's the result:

```
let reverse xs =
  let revOp acc x = x :: acc
  in List.fold revOp [] xs
```


Can we proceed to break down the definition into smaller, more general building blocks?

Consider `revOp`. It is really just a "cons" (`::`), but with switched arguments

A general function that switches (or flips) arguments:

```
flip : (a -> b -> c) -> (b -> a -> c)
let flip f x y = f y x
```

(So `flip f` is a function that performs `f` but with flipped arguments)

Then

```
let cons x xs = x :: xs
revOp acc x = flip cons acc x
```

The declaration of `revOp` can be simplified to

```
revOp = flip cons
```

Finally, we obtain

```
let reverse = List.fold (flip cons) []
```

Simple? Obfuscated? It's much a matter of training to appreciate this style

Nameless Functions

Functions don't have to be given names

We can write *nameless functions* through λ-*abstraction*:

`fun x ->` e stands for function with formal argument x and function body e

(Comes from λ-*calculus*, where we write $\lambda x.e$)

Example: `fun x -> x + 1`, an increment-by-one function

`List.map (fun x -> x + 1) xs` returns list with all elements incremented by one

Nameless functions are often convenient to use with higher-order functions, no need to declare functions that are used only once

Some Syntactical Conveniences

`fun x y -> e` shorthand for `fun x -> ( fun y -> e)`

Pattern matching as in ordinary definitions, like `fun (x,y) -> x + y`

Currying can be defined through λ-abstraction:

`simple 5` $=$ `fun x y -> simple 5 x y`

Also note:

`let (rec) f x = ....`

is precisely the same as

`let (rec) f = fun x -> (....)`

Another Syntactical Convenience

```
function
| pattern_1 -> expr_1
  ...
| pattern_n -> expr_n
```

is shorthand for

```
fun x -> match x with
         | pattern_1 -> expr_1
           ...
         | pattern_n -> expr_n
```

Convenient when matching directly on function arguments. Used a lot in the book

An Example

```
posInts : [int] -> [bool]
posInts xs = let test x = x > 0 in List.map test xs
```

can be written

```
posInts xs = List.map (fun x -> x > 0) xs
```

or even, through "curry-cancelling"

```
posInts = List.map (fun x -> x > 0)
```

Concise! Easy to understand? You judge.

A Second Example

Remember our file i/o example, turning whitespaces between words to single spaces?

```
let string_2_words s = string2words 0 s
```

```
let s = File.ReadAllText("in.txt")
        |> string_2_words
        |> words2string
in File.WriteAllText("out.txt",s)
```

With nameless functions we can avoid some declarations:

```
File.ReadAllText("in.txt")
|> string2words 0
|> words2string
|> (fun s -> File.WriteAllText("out.txt",s))
```


Function Composition

A well-known operation in mathematics, there defined thus:

$$(f \circ g)(x) = g(f(x)), \quad \text{for all } x$$

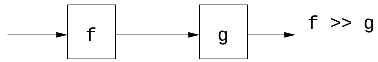
F# definition:

```
(>>) : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c
let (>>) f g x = g (f x)
```

Similar to the “forward pipe” operator |>: we have

```
x |> f |> g = (f >> g) x
```

Which one to use is often a matter of taste



An Example

With function composition,

```
File.ReadAllText("in.txt")
|> string2words 0
|> words2string
|> (fun s -> File.WriteAllText("out.txt",s))
```

can be written as

```
(string2words 0 >> words2string
                >> (fun s -> File.WriteAllText("out.txt",s)))
                   File.ReadAllText("in.txt")
```

(A composed function applied to the string read by `File.ReadAllText`)