Higher order functions

Ziyan Maraikar

August 27, 2014

Lecture Outline

1. Functions as values
2. Closures
3. Declarative programming

Common patterns

We often write functions that are nearly identical. For example, computing

- Sums of squares, cubes, ...
- Arithmetic series
- Geometric series
- Taylor series of various functions

all have the same basic pattern.

Generalising functions

$$\sum_{n=1}^{N} a + (n-1)d$$

$$\sum_{n=1}^{N} ar^n - 1$$

```
let rec arith_sum a d n =
  if n=1 then a
  else
  let t = a +. float (n-1)
   *.d in
  t +. arith_sum a d (n-1)
```

```
let rec geom_sum a r n =
  if n=1 then a
  else
  let t = a *. r ** float (n
   -1) in
  t +. geom_sum a r (n-1)
```

Is there a way to avoid such repetition?

Higher order functions

$$\sum_{n=1}^{N} t_{n-1}$$

We need a way to *abstract out* the calculation of the terms

```
let rec series_sum term n =
  if n=1 then term 1
  else term n +. series_sum term (n-1)
```

Note that the parameter `term` is a *function*.

```
let aterm i = 1. +. float (i-1) *. 2. in
series_sum aterm 10
```

`series_sum` is an example of a *higher order function*.

Exercise

```
let rec series_sum term n =
  if n=1 then term 1
  else term n +. series_sum term (n-1)
;;
let aterm i = 1. +. float (i-1) *. 2.in
series_sum aterm 2
```

1. Show the evaluation of the expression above.
2. Write an expression to calculate the sum of the first ten terms of the geometric series with $a = 5, r = 2$.

Anonymous functions (lambdas)

Ocaml provides a shorthand notation to define functions without a name.

```
(* square function *)
fun x -> x *x

let rec series_sum term n =
  if n=1 then term 1
  else term n +. series_sum term (n-1)
  ;;
series_sum (fun i -> 1. +. float (i-1) *. 2.) 2
```

Can you define a recursive function using a lambda?

Lecture Outline

Specialising an HOF

Our "general solution" is a lot less convenient to use than a specialised function. e.g: compare the following equivalant expressions:

```
arith_sum 1. 2. 2
series_sum (fun i ->  1. +. float (i-1) *. 2.) 2
```

We can rewirte `arith_sum` in terms of `series sum` so we have to provide the lambda ourselves

```
let arith_sum a d n =
  series_sum (fun i ->  a +. float (i-1) *. d) n
```

Exercise

1. ```
   let arith_sum a d n =
     series_sum (fun i ->  a +. float (i-1) *. d) n
   let rec series_sum term n =
     if n=1 then term 1
     else term n +. series_sum term (n-1)
   ```

   Show the evaluation of `arith_sum 1 1 1`.

2. Write an HOF `compose f g` which returns $f(g(x))$.
   ```

Lexical closures

```
arith_sum 1 1 1
series sum (fun i -> 1. +. float(i-1) *. 1.) 1
term 1
1. +. float(1-1) *. 1.
```

Note how the values for `a` and `d` of `arith_sum` are *bound* within the lambda.

They are available at the point of use in `series_sum`.

Lecture Outline

Abstracting control flow

- Recursion is difficult to reason about[1].
- Writing an HOF for a particular task lets us avoid direct use of recursion.

  ```
  let arith_sum a d n =
    series_sum (fun i ->  a +. float (i-1) *. d) n
  ```

  Once we have written `series_sum` we can ignore the "low level details" of how the summation happens.
- Can we do the same for other kinds of operations as well?

[1] Loops are no easier!

Common list operations

- **map** Perform an operation on individual elements of a list, e.g. scale each element by a constant.
- **fold** [2] Combine the elements of a list using an operation, e.g. sum of elements.
- **filter** Remove elements that do not meet a particular condition, e.g. remove all negative elements.
- **zip** Combine pairs of elements in two lists together using an operation.

Have you seen these functions elsewhere?

[2] or reduce

Map

```
(* Apply the function f to each element of list l *)
let map f l =
  match l with
  hd::tl -> f hd :: map f tl
  [] -> []
```

What is the type of this function?

Fold (left)

```
(* Compose element of list l using
    function f and identity e *)
let fold f e l =
  match l with
  hd::tl -> f hd (fold f e tl)
  [] -> ______
```

Example: `fold (+) 0 [1; 2; 3]`

Exercises

- Multiply the numbers in the list `[1; 2; 3]` together.
- Show the evaluation of the expression
  `map (fun x -> x * x) [2]`.
- Define the function `filter p l` that removes the elements that fail the predicate[3] `p` from list `l`.

[3] boolean function

Why HOFs?

- ★ Abstracts common patterns leading to less code.
- ★ Makes it easier to reason about programs by abstracting control flow.
- ★ Easy to parallelise. e.g. Map-reduce computing on clusters and CUDA kernels.

Algebraic data types

Ocaml provides three basic ways to structure data

Tuples and records represent data combinations that are a *product sets* $\alpha \times \beta$

Variants represent data combinations that are a *disjoint union* of sets $\alpha \sqcup \beta$

Functions represent mappings from one data type to another $\alpha \rightarrow \beta$. This corresponds to the *power set*[4] b^a.

[4] Functions can be represented as a (possibly infinite) set of pairs.