CS 320: Concepts of Programming Languages

Lecture 8: Higher-Order Programming

Ankush Das

Nathan Mull

Sep 26, 2024

Abstraction of the Day

- Today, we will talk about another cool abstraction!
- *Higher-Order Functions*
- These are functions that take other functions as arguments
- In fact, functions in OCaml have *first-class status* (they're also called *first-class citizens*)
- Today, we will understand what this means and why it's cool

Why First-Class Functions?

- Meaning: functions in OCaml are like any other value of type bool, int, float, string
  - They can be assigned to local/global variables
  - They can be returned by a function
  - They can be given as arguments to functions
  - They are treated like any other expression/value
- *Contrast: Types are not first-class, they cannot be given as arguments or returned by functions*

An Example of First-Class

- Suppose we define: `let f x y = x * x + y * y`
- Functions can be given another name: `let h = f`
- We can also apply functions partially:
  `let g = f 3`   `let h = f 4`
- Definition of g: `fun y -> 3 * 3 + y * y`
  Definition of h: `fun y -> 4 * 4 + y * y`
- Functions can be returned: `let foo x = if x then g else h`
- We can call `foo true 10 = 3 * 3 + 10 * 10 = 109`

Functions can also be Arguments

- We can define:

```
('a -> 'a) -> 'a -> 'a
let twice f x = f (f x)
```

- We can also apply the function f recursively:

```
('a -> 'a) -> 'a -> int -> 'a
let rec ntimes f x n =
  if n = 0 then x else f (ntimes f x (n-1))
```

- Application: Computing Fixed Point of a Function

```
('a -> 'a) -> 'a -> 'a
let rec fixed_point f x =
  if f x = x then x else fixed_point f (f x)
```

Even Simple Functions can benefit!

- Recall factorial, sum, and generate functions:
- What operations are common in the three?
- All stop at `n = 0`; make a recursive call at `(n-1)` and combine the result with `n`

```
int -> int
let rec sum n =
  if n = 0 then 0 else n + sum (n-1)
```

```
int -> int
let rec factorial n =
  if n = 0 then 1 else n * factorial (n-1)
```

```
int -> int list
let rec generate n =
  if n = 0 then [] else n::(generate (n-1))
```

This Pattern can be Abstracted!!


```
'a -> (int -> 'a -> 'a) -> int -> 'a
let rec f base op n =
  if n = 0 then base else op n (f base op (n-1))
```

- If `n = 0`; return `base`; else call function `op` on `n` and `f (n-1)`
- `base` and `op` become arguments to the function

```
int -> int
let sum = f 0 ( + )
int -> int
let factorial = f 1 ( * )
int -> int list
let generate = f [] (fun h t -> h :: t)
```

How to make Functions Higher-Order?

- Observe the main computation happening in a function
- See what operations does the computation need? What function calls, what arguments, etc.
- Try and make them arguments to an appropriate higher-order function
- This principle is informally called "*Abstraction Principle*"
- We will see two main examples of this: `map` and `filter`

List Operations

- Suppose we want to update elements of a list, e.g. increment, decrement, double, etc.
- In each function, the pattern is exactly the same
- Empty list returns an empty list
- Otherwise, we perform an operation on the head and call the function recursively on the tail

```
int list -> int list
let rec inc l =
  match l with
  | [] -> []
  | h::t -> (h+1)::(inc t)

int list -> int list
let rec dec l =
  match l with
  | [] -> []
  | h::t -> (h-1)::(dec t)

int list -> int list
let rec double l =
  match l with
  | [] -> []
  | h::t -> (2*h)::(double t)
```

Abstracting this Pattern: Map Function


```
('a -> 'b) -> 'a list -> 'b list
let rec map f l =
  match l with
  | [] -> []
  | h::t -> (f h)::(map f t)
```

- Empty list returns an empty list
- Otherwise, call `f` on the head and call map recursively on the tail
- Effect: `f` is called on every element of the list
- Observe the higher-order type!

Simple Examples


```
('a -> 'b) -> 'a list -> 'b list
let rec map f l =
  match l with
  | [] -> []
  | h::t -> (f h)::(map f t)
```

```
int list -> int list
let inc l = map (fun x -> x + 1) l

int list -> int list
let dec l = map (fun x -> x - 1) l

int list -> int list
let double l = map (fun x -> 2 * x) l
```

Map can Update Types as Well!

('a -> 'b) -> 'a list -> 'b list

```
let rec map f l =
  match l with
  | [] -> []
  | h::t -> (f h)::(map f t)
```

int list -> bool list

```
let is_pos l = map (fun x -> x > 0) l
```

int list -> float list

```
let to_float l = map float_of_int l
```

Tail-Recursive Map Function


```
('a -> 'b) -> 'a list -> 'b list
let map_tr f l =
  let rec helper l acc =
    match l with
    | [] -> List.rev acc
    | h::t -> helper t (f h :: acc)
  in
  helper l []
```

- Usual trick: keep an accumulator to collect the mapped elements in reverse order
- When `l` becomes empty, return the reverse of the accumulator

Map Works on Trees Too!


```
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree

('a -> 'b) -> 'a tree -> 'b tree
let rec map f tr =
  match tr with
  | Leaf -> Leaf
  | Node(x, l, r) -> Node(f x, map f l, map f r)
```

- Leaf returns leaf; otherwise, call `f` on the node element `x` and call map recursively on the left and right children
- Effect: `f` is called on every node of the tree

List Filter Function

- Idea behind filter: only keep elements that satisfy a given property
- Definition:

```
('a -> bool) -> 'a list -> 'a list
let rec filter p l =
  match l with
  | [] -> []
  | h::t -> if p h then h::(filter p t) else filter p t
```

- Takes a function `p : 'a -> bool` (aka predicate)
- If `p h` returns `true`, then `h` goes in the returned list, otherwise not

Examples

- Suppose we want to filter out negative elements of a list
- Definition:

```
int list -> int list
let drop_negs l = filter                          l
```

- Suppose we want to keep only even numbers in a list
- Definition:

```
int list -> int list
let evens l = filter                              l
```

Examples

- Suppose we want to filter out negative elements of a list
- Definition:

```
int list -> int list
let drop_negs l = filter (fun x -> x >= 0) l
```

- Suppose we want to keep only even numbers in a list
- Definition:

```
int list -> int list
let evens l = filter                            l
```

Examples

- Suppose we want to filter out negative elements of a list
- Definition:

```
int list -> int list
let drop_negs l = filter (fun x -> x >= 0) l
```

- Suppose we want to keep only even numbers in a list
- Definition:

```
int list -> int list
let evens l = filter (fun x -> x mod 2 = 0) l
```

Tail Recursive Filter


```
('a -> bool) -> 'a list -> 'a list
let filter_tr p l =
  let rec helper l acc =
    match l with
    | [] -> List.rev acc
    | h::t -> if p h then helper t (h::acc) else helper t acc
  in
  helper l []
```

Tail Recursive Filter


```
('a -> bool) -> 'a list -> 'a list
let filter_tr p l =
  let rec helper l acc =
    match l with
    | [] -> List.rev acc
    | h::t -> if p h then helper t (h::acc) else helper t acc
  in
  helper l []
```

Tail Recursive Filter


```
('a -> bool) -> 'a list -> 'a list
let filter_tr p l =
  let rec helper l acc =
    match l with
    | [] -> List.rev acc
    | h::t -> if p h then helper t (h::acc) else helper t acc
  in
  helper l []
```

Conclusion

- Functions `map` and `filter` are available from the standard list library
- You can use `List.map` and `List.filter`
- Read OCaml book 4.1, 4.2, 4.3
- Next week, we will cover another higher-order function called `fold`