Programming Languages and Techniques (CIS1200)

Lecture 9

Higher-order functions: transform and fold

Lecture notes: Chapter 9

Announcements (1)

- Please complete the intro survey (link on Ed) available this afternoon
- Homework 3 available
  - Practice with BSTs, generic functions, first-class functions, and abstract types
  - Due Tuesday, February 11th at 11:59pm
  - *Start early!*
    - *Problems 1-4 can be done after class today*
    - *Problems 5-8 can be done after class on Friday*
- Reading: Chapters 8, 9, and 10 of the lecture notes

Announcements (2)

- Midterm 1: Friday, February 14th
  - Coverage: up to Wednesday, Feb 12th (Chapters 1-10)
  - During lecture
    Last names: A – Z Meyerson Hall B1
  - 60 minutes; closed book, closed notes
  - Review Material
    - old exams on the web site (“schedule” tab)
  - Review Session
    - Wednesday, Feb 12th, 7:00-9:00pm, Towne 100 (will be recorded)
    - Review Videos will be posted this weekend

First-Class Functions

First-class Functions

function body

```
let f : t -> u = fun (x:t) -> <body>
```

function type

anonymous function value

- Functions are *first-class values* in OCaml: they can be manipulated like any other value.
- They have a type that specifies the input and output types.
- The "`fun`" keyword introduces an *anonymous function*.
  - Sometimes called *lambdas** or *closures*

*The term "lambda" comes from Church's *lambda calculus*.

2 = 1 + 1

A function that takes *two* arguments…

```
int -> int -> int
```

has the same type as a function that takes *one* argument and returns a function that takes *one* argument

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

This is actually useful!

Multiple Arguments

We can decompose a standard function definition

```
let sum (x : int) (y:int) : int = x + y
```

into parts:

```
let sum = fun (x:int) -> fun (y:int) -> x + y
```

define a variable with that value

create a function value

that returns a function value

The two definitions of sum have the same type and behave the same!

```
let sum : int -> int -> int
```

Partial Application

```
let sum (x : int) (y:int) : int = x + y
```

```
sum 3
⟼ (fun (x:int) -> fun (y:int) -> x + y) 3      definition
⟼ fun (y:int) -> 3 + y                         substitute 3 for x
```

the result of a "partially applied function" is itself a function (that can later be applied)

Functions that return functions

```
let sum (x : int) (y:int) : int = x + y
```

```
let sum = fun (x:int) -> fun (y:int) -> x + y
```

```
sum 3
⟼ (fun (x:int) -> fun (y:int) -> x + y) 3      definition
⟼ fun (y:int) -> 3 + y                         substitute 3 for x
```

the result of a "partially applied function" is itself a function (that can later be applied)

List transformations

A fundamental design pattern
using first-class functions

Phone book example

```
type entry = string * int
let phone_book = [ ("Stephanie", 2155559092), … ]

let rec get_names (p : entry list) : string list =
  begin match p with
  | ((name, num)::rest) -> name :: get_names rest
  | [] -> []
  end

let rec get_numbers (p : entry list) : int list =
  begin match p with
  | ((name, num)::rest) -> num :: get_numbers rest
  | [] -> []
  end
```

Can we use first-class functions to refactor code to share common structure?

Refactoring

```
let rec helper (f:entry -> 'b) (p:entry list) : 'b list =
  begin match p with
  | (entry::rest) -> f entry :: helper f rest
  | [] -> []
  end

let get_names (p : entry list) : string list =
  helper fst p
let get_numbers (p : entry list) : int list =
  helper snd p
```

fst and snd are functions that access the parts of a tuple:

```
let fst (x,y) = x
let snd (x,y) = y
```

The argument `f` determines what happens with the entry at the head of the list

Going even more generic

```
let rec helper (f:entry -> 'b) (p:entry list) : 'b list =
  begin match p with
  | (entry::rest) -> f entry :: helper f rest
  | [] -> []
  end

let get_names (p : entry list) : string list =
  helper fst p
let get_numbers (p : entry list) : int list =
  helper snd p
```

Now let's make it work for *all* lists, not just lists of entries…

Going even more generic

```
let rec helper (f:'a -> 'b) (p:'a list) : 'b list =
  begin match p with
  | (entry::rest) -> f entry :: helper f rest
  | [] -> []
  end

let get_names (p : entry list) : string list =
  helper fst p
let get_numbers (p : entry list) : int list =
  helper snd p
```

'a stands for `(string*int)`
'b stands for `int`

`snd : (string*int) -> int`

Transforming Lists

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

List *transformation*

a.k.a. "*mapping** a function across a list"

- foundational function for programming with lists
- part of OCaml standard library (called List.map)
- used over and over again

(e.g., Google's famous *map*-reduce infrastructure)

*many languages (including OCaml) use the terminology "map" for the function that transforms a list by applying a function to each element. Don't confuse List.map with "finite map".

9: What is the value of this expresssion?

✓ 0

[0; -1; 1; -2]

0%

[1]

0%

[1; 1; 0; 1]

0%

[false; false; true; false]

0%

runtime error

0%

What is the value of this expresssion?

```
transform (fun (x:int) -> x > 0)
    [0 ; -1; 1; -2]
```

1. [0; -1; 1; -2]
2. [1]
3. [1; 1; 0; 1]
4. [false; false; true; false]
5. runtime error

ANSWER: 4

The ‘`fold`’ design pattern

a general-purpose recursive function

Refactoring code, again

Is there a pattern in the definition of these two functions?

```
let rec exists (l : bool list) : bool =
   begin match l with
   | [] -> false
   | h :: t -> h || exists t
   end
```

```
let rec acid_length (l : acid list) : int =
   begin match l with
   | [] -> 0
   | h :: t -> 1 + acid_length t
   end
```

Refactoring code, again

Is there a pattern in the definition of these two functions?

```
let rec exists (l : bool list) : bool =
   begin match l with
   | [] -> false
   | h :: t -> h || exists t
   end
```

```
let rec acid_length (l : acid list) : int =
   begin match l with
   | [] -> 0
   | h :: t -> 1 + acid_length t
   end
```

base case:
Simple answer when the list is empty

combine step:
Do something with the head of the list and the result of the recursive call

Can we factor out this pattern using first-class functions?

Preparation

```
let rec exists (l : bool list) : bool =
   begin match l with
   | [] -> false
   | h :: t -> h || exists t
   end
```

```
let rec acid_length (l : acid list) : int =
   begin match l with
   | [] -> 0
   | h :: t -> 1 + acid_length t
   end
```

Preparation: introduce a helper

```
let rec helper (l : bool list) : bool =
   begin match l with
   | [] -> false
   | h :: t -> h || helper t
   end

let exists (l : bool list) = helper l
```

First: introduce a helper function that will (eventually) become the same for both definitions.

```
let rec helper (l : acid list) : int =
   begin match l with
   | [] -> 0
   | h :: t -> 1 + helper t
   end

let acid_length (l : acid list) = helper l
```

Abstracting with respect to Base

```
let rec helper (l : bool list) : bool =
   begin match l with
   | [] -> false
   | h :: t -> h || helper t
   end

let exists (l : bool list) = helper l
```

```
let rec helper (l : acid list) : int =
   begin match l with
   | [] -> 0
   | h :: t -> 1 + helper t
   end

let acid_length (l : acid list) = helper l
```

Abstracting with respect to Base

```
let rec helper (base : bool) (l : bool list) : bool =
   begin match l with
   | [] -> base
   | h :: t -> h || helper base t
   end

let exists (l : bool list) = helper false l
```

```
let rec helper (base : int) (l : acid list) : int =
   begin match l with
   | [] -> base
   | h :: t -> 1 + helper base t
   end

let acid_length (l : acid list) = helper 0 l
```

Abstracting with respect to Combine

```
let rec helper (base : bool) (l : bool list) : bool =
   begin match l with
   | [] -> base
   | h :: t -> h || helper base t
   end

let exists (l : bool list) = helper false l
```

```
let rec helper (base : int) (l : acid list) : int =
   begin match l with
   | [] -> base
   | h :: t -> 1 + helper base t
   end

let acid_length (l : acid list) = helper 0 l
```

Abstracting with respect to Combine

```
let rec helper (base : bool) (l : bool list) : bool =
   begin match l with
   | [] -> base
   | h :: t -> h || helper base t
   end

let exists (l : bool list) = helper false l
```

```
let rec helper (base : int) (l : acid list) : int =
   begin match l with
   | [] -> base
   | h :: t -> 1 + helper base t
   end

let acid_length (l : acid list) = helper 0 l
```

Abstracting with respect to Combine

```
let rec helper (combine : bool -> bool -> bool)
               (base : bool) (l : bool list) : bool =
   begin match l with
   | [] -> base
   | h :: t -> combine h (helper combine base t)
   end

let exists (l : bool list) =
  helper (fun (h:bool) (acc:bool) -> h || acc) false l
```

```
let rec helper (combine : acid -> int -> int)
               (base : int) (l : acid list) : int =
   begin match l with
   | [] -> base
   | h :: t -> combine h (helper combine base t)
   end

let acid_length (l : acid list) =
  helper (fun (h:acid) (acc:int) -> 1 + acc) 0 l
```

What about the types?

```
let rec helper (combine : bool -> bool -> bool)
               (base : bool) (l : bool list) : bool =
   begin match l with
   | [] -> base
   | h :: t -> combine h (helper combine base t)
   end

let exists (l : bool list) =
  helper (fun (h:bool) (acc:bool) -> h || acc) false l
```

```
let rec helper (combine : acid -> int -> int)
               (base : int) (l : acid list) : int =
   begin match l with
   | [] -> base
   | h :: t -> combine h (helper combine base t)
   end

let acid_length (l : acid list) =
  helper (fun (h:acid) (acc:int) -> 1 + acc) 0 l
```

What about the types?

```
let rec helper (combine : bool -> bool -> bool)
               (base : bool) (l : bool list) : bool =
   begin match l with
   | [] -> base
   | h :: t -> combine h (helper combine base t)
   end

let exists (l : bool list) =
  helper (fun (h:bool) (acc:bool) -> h || acc) false l
```

```
let rec helper (combine : acid -> int -> int)
               (base : int) (l : acid list) : int =
   begin match l with
   | [] -> base
   | h :: t -> combine h (helper combine base t)
   end

let acid_length (l : acid list) =
  helper (fun (h:acid) (acc:int) -> 1 + acc) 0 l
```

Making the Helper Generic

```
let rec helper (combine : 'a -> 'b -> 'b)
               (base : 'b) (l : 'a list) : 'b =
   begin match l with
   | [] -> base
   | h :: t -> combine h (helper combine base t)
   end

let exists (l : bool list) =
  helper (fun (h:bool) (acc:bool) -> h || acc) false l
```

The helpers now have the *same* type.

```
let rec helper (combine : 'a -> 'b -> 'b)
               (base : 'b) (l : 'a list) : 'b =
   begin match l with
   | [] -> base
   | h :: t -> combine h (helper combine base t)
   end

let acid_length (l : acid list) =
  helper (fun (h:acid) (acc:int) -> 1 + acc) 0 l
```

But they are instantiated differently for the two uses.

List Fold

```
let rec fold (combine: 'a -> 'b -> 'b)
             (base:'b) (l : 'a list) : 'b =
   begin match l with
   | [] -> base
   | x :: t -> combine x (fold combine base t)
   end

let exists (l : bool list) : bool =
    fold (fun (h:bool) (acc:bool) -> h || acc) false l

let acid_length (l : acid list) : int =
    fold (fun (h:acid) (acc:int) -> 1 + acc) 0 l
```

Just rename "helper" to "fold".

`fold` (a.k.a. "reduce")

- Like `transform`, foundational function for programming with lists
- Captures the pattern of *recursion over lists*
- Part of OCaml standard library (`List.fold_right`)
- Similar operations for other recursive datatypes (`fold_tree`)

Using List Fold

```
let rec fold (combine: 'a -> 'b -> 'b)
             (base:'b) (l : 'a list) : 'b =
   begin match l with
   | [] -> base
   | x :: t -> combine x (fold combine base t)
   end

let exists (l : bool list) : bool =
    fold (fun (h:bool) (acc:bool) -> h || acc) false l
```

fold:
general-purpose recursion

combine function:
computes the result given h the head of the list and acc the "*accumulated*" answer given by recursion

base case:
answer when the list is empty

9: Rewrite using fold

0

1

0%

2

0%

3

0%

4

0%

How would you rewrite this function

```
let rec sum (l : int list) : int =
   begin match l with
   | [] -> 0
   | h :: t -> h + sum t
   end
```

using fold? What should be the arguments for base and combine?

1. combine is: `(fun (h:int) (acc:int) -> acc + 1)`
   base is: `0`
2. combine is: `(fun (h:int) (acc:int) -> h + acc)`
   base is: `0`
3. combine is: `(fun (h:int) (acc:int) -> h + acc)`
   base is: `1`
4. sum can't be written with `fold`.

Answer: 2

9: Rewrite using fold

0

1	0%
2	0%
3	0%
4	0%

How would you rewrite this function

```
let rec reverse (l : int list) : int list =
   begin match l with
   | [] -> []
   | h :: t -> reverse t @ [h]
   end
```

using fold? What should be the arguments for base and combine?

1. combine is: `(fun (h:int) (acc:int list) -> h :: acc)`
   base is: `0`

2. combine is: `(fun (h:int) (acc:int list) -> acc @ [h])`
   base is: `0`

3. combine is: `(fun (h:int) (acc:int list) -> acc @ [h])`
   base is: `[]`

4. reverse can't be written by with `fold`.

Answer: 3

Functions as Values

- We’ve seen many ways in which functions can be treated as values in OCaml
- Everyday programming practice (in many languages, not just OCaml!) offers many more examples
  - objects bundle “functions” (a.k.a. methods) with data
  - iterators (“cursors” for walking over data structures)
  - event listeners (in GUIs)
  - etc.
- Also heavily used for large-scale computing: Google’s MapReduce
  - Framework for transforming (mapping) sets of key-value pairs
  - Then “reducing” the results per key of the map
  - Easily distributed to 10,000 machines to execute in parallel!