Programming Languages and Techniques (CIS120)

Lecture 9

September 16th, 2015

HOF patterns: transform & fold

Announcements

- Homework 3 is available
  - Due THURSDAY, September 24th at 11:59:59pm
  - Practice with BSTs, generic functions, HOFs and *abstract types*
- If you added CIS 120 recently, make sure that you can see your scores online.
  - If you get feedback about your scores, you are in our database.
  - If not, please send mail to tas120@lists.seas.upenn.edu
  - If you see unsubmitted "quizzes", you may need to register your clicker
- Read chapters 9 & 10 of the lecture notes

First-class Functions

Higher-order Programs

or

How not to repeat yourself, Part II.

First-class Functions

- You can pass a function as an *argument* to another function:

```
let twice (f:int -> int) (x:int) : int =
  f (f x)

let add_one (z:int) : int = z + 1
let add_two (z:int) : int = z + 2
let y = twice add_one 3
let w = twice add_two 3
```

function type: argument of type int and result of type int

- You can *return* a function as the result of another function.

```
let make_incr (n:int) : int -> int =
  let helper (x:int) : int =
        n + x
  in
  helper
let y = twice (make_incr 1) 3
```

First-class Functions

- You can store functions in data structures

```
let add_one   (x:int) : int = x+1
let add_two   (x:int) : int = x+2
let add_three (x:int) : int = x+3

let func_list : (int -> int) list =
   [ add_one; add_two; add_three ]
```

A list of (int -> int) functions.

```
let func_list1 : (int -> int) list =
   [ make_incr 1; make_incr 2; make_incr 3 ]
```

Simplifying First-Class Functions

```
let twice (f:int -> int) (x:int) : int =
  f (f x)

let add_one (z:int) : int = z + 1
```

	twice add_one 3	
⟼	add_one (add_one 3)	*substitute add_one for f, 3 for x*
⟼	add_one (3 + 1)	*substitute 3 for z in add_one*
⟼	add_one 4	*3+1⇒4*
⟼	4 + 1	*substitute 4 for z in add_one*
⟼	5	*4+1⇒5*

Evaluating First-Class Functions

```
let make_incr (n:int) : int -> int =
  let helper (x:int) : int = n + x in
  helper
```

```
    make_incr 3
```

substitute 3 for n

```
⟼ let helper (x:int) = 3 + x in helper
⟼ ???
```

Evaluating First-Class Functions

```
let make_incr (n:int) : int -> int =
  let helper (x:int) : int = n + x in
  helper
```

```
    make_incr 3
```

substitute 3 for n

```
⟼ let helper (x:int) = 3 + x in helper
⟼ fun (x:int) -> 3 + x
```

Anonymous function value

keyword “fun”

“->” after arguments
no return type annotation

Named function values

A standard function definition:

```
let add_one (x:int) : int = x+1
```

really has two parts:

```
let add_one : int -> int = fun (x:int) -> x+1
```

define a name for the value

create a function value

Both definitions have the same type and behave exactly the same.

Anonymous functions

```
let add_one (z:int) : int = z + 1
let add_two (z:int) : int = z + 2
let y = twice add_one 3
let w = twice add_two 3
```

```
let y = twice (fun (z:int) -> z+1) 3
let w = twice (fun (z:int) -> z+2) 3
```

Multiple Arguments

We can decompose a standard function definition:

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

into two parts:

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

define a variable with that value

create a function value

Both definitions have the same interface and behave exactly 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
⟼ fun (y:int) -> 3 + y
```

definition

substitute 3 for x

What is the value of this expresssion?

```
let f (x:bool) (y:int) : int =
    if x then 1 else y in

f true
```

1. `1`
2. `true`
3. `fun (y:int) -> if true then 1 else y`
4. `fun (x:bool) -> if x then 1 else y`

Answer: 3

What is the value of this expression?

```
let f (g : int -> int) (y: int) :int =
   g 1 + y in

f (fun (x:int) -> x + 1) 3
```

1. 1
2. 2
3. 3
4. 4
5. 5

Answer: 5

What is the type of this expression?

```
let f (g : int -> int) (y: int) : int =
   g 1 + y in

f (fun (x:int) -> x + 1)
```

1. `int`
2. `int -> int`
3. `int -> int -> int`
4. `(int -> int) -> int -> int`
5. `ill-typed`

Answer: 2

What is the type of this expresssion?

```
[ (fun (x:int) -> x + 1);
  (fun (x:int) -> x – 1) ]
```

1. `int`
2. `int -> int`
3. `(int -> int) list`
4. `int list -> int list`
5. `ill typed`

Answer: 3

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` controls what happens with the binding 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
- occurs over and over again
- part of OCaml standard library (called List.map)

Example of using transform:

```
transform is_engr ["FNCE";"CIS";"ENGL";"DMD"] =
    [false;true;false;true]
```

*confusingly, 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".

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

What is the value of this expresssion?

```
transform String.uppercase [“a”;“b”;“c”]
```

1. []
2. [“a”; “b”; “c”]
3. [“A”; “B”; “C”]
4. runtime error

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

List processing

The ‘fold’ design pattern

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 recursive call

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

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 (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
```

```
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
```

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
```

- `fold` (a.k.a. Reduce)
  - Like transform, foundational function for programming with lists
  - Captures the pattern of recursion over lists
  - Also part of OCaml standard library (`List.fold_right`)
  - Similar operations for other recursive datatypes (`fold_tree`)