Programming Languages and Techniques (CIS120)

Lecture 10

Feb 10, 2014

First-class functions

Announcements

- HW #3 due Tuesday at midnight
- Read Ch 10 of the lecture notes
- Midterm 1
  - Scheduled *in class* on *Friday, February 21st*
  - Review session Wednesday, February 19th, 7-9PM in Wu & Chen
  - More details to follow!

Abstracting with first-class functions

Motivating design problem

- Suppose you are given an association list from students to majors, but you wanted a list that includes only students in the engineering school? Or only students in wharton?

```
type student = string
type major   = string
type school  = SEAS | WHARTON | SAS | NURSING
type roster  = (student * major) list

let to_school (m : major) : school = …

let is_engr    (m : major) : bool = to_school m = SEAS
let is_wharton (m : major) : bool = to_school m = WHARTON

let only_engr (m : roster) : roster = ???
let only_wharton (m : roster) : roster = ???
```

Demo: majors.ml

First Class Functions

Functions *are* values

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 z = twice add_one 3
```

- 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
```

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 ]
```

```
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

Function values

A standard function definition:

```
let is_engr (m : major) : bool = to_school m = SEAS
```

really has two parts:

```
let is_engr = fun (m:major) -> to_school m = SEAS
```

define a variable with that value

create a function value

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

```
val is_engr : major -> bool
```

Anonymous functions

```
let is_engr (m : major) : bool = to_school m = SEAS
let is_sas  (m : major) : bool = to_school m = SAS

let rec only (f : major -> bool) (r: roster) = …
let only_engr (r : roster) : roster =
   only is_engr r
let only_sas (r : roster) : roster =
   only is_sas r
```

```
let only_engr (r : roster) : roster =
   only
     (fun (m:major) -> to_school m = SEAS) r
let only_sas (r : roster) : roster =
   only
     (fun (m:major) -> to_school m = SAS) r
```

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:

```
val 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
```

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`

What is the value of this expresssion?

```
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

What is the type of this expresssion?

```
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`

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`

List transformations

A fundamental design pattern
using first-class functions

Refactoring code: Keys and Values

```
let rec keys (m:('k*'v) list) : 'k list =
  begin match m with
    | [] -> []
    | (k,v)::rest -> k::(keys rest)
  end

let rec values (m:('k*'v) list) : 'v list =
  begin match m with
    | [] -> []
    | (k,v)::rest -> v::(values rest)
  end
```

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

Keys and Values

```
let rec helper (f:('k*'v) -> 'b) (m: ('k*'v) list)
  : 'b list =
  begin match m with
  | [] -> []
  | h::t -> f h :: helper f t
  end


let keys   (m:('k,'v) map) : 'k list = helper fst m
let values (m:('k,'v) map) : 'v list = helper snd m
```

The argument `f` controls what happens with the binding at the head of the list

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

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

Going even more generic

```
let rec helper (f:(‘k*’v) -> ‘b) (m: (’k*'v) list)
  : ‘b list =
  begin match m with
  | [] -> []
  | h::t -> f h :: helper f t
  end


let keys   (m:(‘k,’v) map) : ‘k list = helper fst m
let values (m:(‘k,’v) map) : ‘v list = helper snd m
```

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

Going even more generic

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


let keys   (m:('k,'v) list) : 'k list = helper fst m
let values (m:('k,'v) list) : 'v list = helper snd m
```

'a stands for ('k*'v)
'b stands for 'k

fst : ('k*'v) -> 'k

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
   | x :: 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?

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 acid_length (l : acid list) : int =
    fold (fun (x:acid) (y:int) -> 1 + y) 0 l
let exists (l : bool list) : bool =
    fold (fun (x:bool) (y:bool) -> x || y) false l
```

- Fold (aka Reduce)
  - Another 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)

Functions as Data

- We've seen a number of ways in which functions can be treated as data in OCaml
- Present-day programming practice offers many more examples at the "small scale":
  - objects bundle "functions" (a.k.a. methods) with data
  - iterators ("cursors" for walking over data structures)
  - event listeners (in GUIs)
  - etc.
- The idiom is useful at the "large scale": Google's MapReduce
  - Framework for mapping across sets of key-value pairs
  - Then "reducing" the results per key of the map
  - Easily distributed to 10,000 machines to execute in parallel!