Programming Languages and Techniques (CIS120)

Lecture 11

February 4, 2013

First-class functions
Announcements

• Guest lecturer (Richard Eisenberg) Wednesday
  – Weirich OH cancelled today, see schedule for TA office hours

• Homework 4 is available
  – Due Monday, February 11th at 11:59:59pm
  – HigherOrderFunctions (today)
  – Quad-trees & mutable state: nbody simulation (Wed)
  – Lecture notes updated on website

• Midterm 1
  – Scheduled in class on Friday, February 15th
  – Review session Wednesday, February 13th, 6-8PM in Wu & Chen
  – More details to follow!
Abstracting with first-class functions
Finite Map Interface

type ('k,'v) map

val empty : ('k,'v) map
val mem : 'k -> ('k,'v) map -> bool
val find : 'k -> ('k,'v) map -> 'v
val add : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
val remove : 'k -> ('k,'v) map -> ('k,'v) map
val bindings : ('k,'v) map -> ('k * 'v) list
• Suppose you are given a finite map from students to majors, but you wanted a map that includes only students in the engineering school? Or only students in wharton?

```ocaml
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 (r : roster) : roster = ???
let only_wharton (r : 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:

```ml
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.

```ml
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

```ocaml
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 ]
```

```ocaml
let func_list : (int -> int) list =
[ make_incr 1; make_incr 2; make_incr 3 ]
```
Evaluating First-Class Functions

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

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

let twice add_one 3
  ⟷ add_one (add_one 3)
  ⟷ add_one (3 + 1)
  ⟷ add_one 4
  ⟷ 4 + 1
  ⟷ 5
```

substitute add_one for f, 3 for x
substitute 3 for z in add_one
because 3+1 ⇒4
substitute 4 for z in add_one
because 4+1 ⇒5
Evaluating First-Class Functions

```ocaml
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:

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

really has two parts:

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

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

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

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

into two parts:

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

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

```plaintext
val sum : int -> int -> int
```
Partial Application

\[
\text{let sum} \ (x:\text{int}) \ (y:\text{int}) \ : \ \text{int} = x + y
\]

\[
\text{sum} \ 3
\]

\[\mapsto (\text{fun} \ (x:\text{int}) \to \text{fun} \ (y:\text{int}) \to x + y) \ 3 \quad \text{definition}
\]

\[\mapsto \text{fun} \ (y:\text{int}) \to 3 + y \quad \text{substitute 3 for x}
\]
Evaluating Partial Application

```
let sum = fun (x:int) (y:int) -> x + y
let add_three = sum 3
let answer = add_three 39

⟵
let sum = fun (x:int) -> fun (y:int) -> x + y
let add_three = (fun (x:int) -> fun (y:int) -> x + y) 3
let answer = add_three 39

⟵
let sum = fun (x:int) -> fun (y:int) -> x + y
let add_three = fun (y:int) -> 3 + y
let answer = add_three 39

⟵
let sum = fun (x:int) -> fun (y:int) -> x + y
let add_three = fun (y:int) -> 3 + y
let answer = (fun (y:int) -> 3 + y) 39
```
Evaluating Partial Application

```
let sum = fun (x:int) -> fun (y:int) -> x + y
let add_three = fun (y:int) -> 3 + y
let answer = (fun (y:int) -> 3 + y) 39

|-->

let sum = fun (x:int) -> fun (y:int) -> x + y
let add_three = fun (y:int) -> 3 + y
let answer = 3 + 39

|-->

let sum = fun (x:int) -> fun (y:int) -> x + y
let add_three = fun (y:int) -> 3 + y
let answer = 42
```
List transformations

Fundamental design pattern using first-class functions
Refactoring code: Keys and Values

```ocaml
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?
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

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 f1 (l : string list) : string list =
   transform String.uppercase l

let f2 (l : int list) : bool list =
   transform (fun (x:int) -> x > 0) l

let f3 (l : (int*int) list) : int list =
   transform (fun (x:(int*int)) -> (fst x)*(snd x)) l

f1 ["a"; "b"; "c"] ⇒ ["A"; "B"; "C"]

f2 [0 ; -1; 1; -2] ⇒ [false; false; true; false]

f3 [(1,2); (3,4)] ⇒ [2; 12]
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

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

base case: Simple answer when the list is empty
combine step: Do something with the head of the list and the recursive call
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!