Programming Languages and Techniques (CIS120)

Lecture 11
Feb 6, 2012

First-class functions
Announcements

• Homework 4 will be available tomorrow
  – due Monday, February 13\textsuperscript{th} at 11:59:59pm
  – n-body physics simulation

• Updated lecture notes also available...

• Midterm 1 will be during class time on Wednesday, February 15\textsuperscript{th}
  – LOCATION: Leidy Lab 10
  – Review material on course website
First-class Functions

• Amazing fact: functions *are* data!
• You can pass a function as an *argument* to another function:

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

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

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

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

\[ \text{let helper (x:int) = 3 + x in helper} \]
```
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
Function values

We can decompose a standard function definition:

```ml
let is-engr (m : major) : bool = to_school m = SEAS
```

into two parts:

```ml
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 interface and behave exactly the same:

```ml
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:

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

into two parts:

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

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

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

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

sum 3

\[
\text{fun (x:int)(y:int) \rightarrow x + y)} \ 3 \quad \text{definition of sum}
\]

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

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

---->

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

---->

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

---->

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

→

let sum = fun (x:int) (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

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 | [] -> [] | (k,v)::t -> f (k,v) :: 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

fst and snd are functions that access the parts of a tuple:
  fst (1,2) = 1
  snd (1,2) = 2
let rec helper (f:('k*'v) -> 'b) (m: ('k*'v) list)
  : 'b list = begin match m with [] -> []
  | (k,v)::t -> f (k,v) :: 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

fst and snd are functions that access the parts of a tuple:
  fst (1,2) = 1
  snd (1,2) = 2
Going even more generic

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

The definition of this function does not depend on operating over a list of bindings. It just passes each binding to the function f.

Let's make it work for ALL lists, not just lists of tuples!
Going even more generic

```ocaml
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”.

CIS120 / Spring 201
Transform examples

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""]
f2 [ 0 ; -1; 1; -2 ]
f3 [ (1,2); (3,4) ]
List processing

Another design pattern for first-class functions
Refactoring code, again

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

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

```ocaml
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!