Programming Languages and Techniques (CIS120)

Lecture 9
February 3, 2016

Higher-order functions
transform & fold
Announcements

• Homework 3 is available
  – Due Tuesday
  – Practice with BSTs, generic functions, HOFs and abstract types

• Exam on Feb 16th
  – Starts at 6:15 (Physics midterm 5-6PM)
  – Sign up for make-up midterm on course website

• 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 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:

```haskell
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
```

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

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

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

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

A list of (int -> int) functions.
Simplifying First-Class Functions

```plaintext
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`
- 3+1⇒4
- substitute 4 for `z` in `add_one`
- 4+1⇒5
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

\[
\begin{align*}
\text{let } & \text{ make_incr } (n:\text{int}) : \text{ int } \to \text{ int } = \\
& \quad \text{let } \text{ helper } (x:\text{int}) : \text{ int } = n + x \text{ in } \\
& \quad \text{helper} \\
\end{align*}
\]

\[
\begin{align*}
\text{make_incr } 3 \\
\quad \text{substitute 3 for } n \\
\rightarrow & \quad \text{let } \text{ helper } (x:\text{int}) = 3 + x \text{ in } \text{helper} \\
\rightarrow & \quad \text{fun } (x:\text{int}) \to 3 + x \\
\end{align*}
\]
Named function values

A standard function definition:

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

really has two parts:

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

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

- define a name for the value
- create a function value
Anonymous functions

```ocaml
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

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

into parts

```plaintext
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

Both definitions have the same interface and behave exactly the same

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

sum 3

(\(x\) : int) \rightarrow (\(y\) : int) \rightarrow x + y)

\(\rightarrow\) fun (y:int) \rightarrow 3 + y

definition

substitute 3 for \(x\)
What is the value of this expression?

```ml
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?

```ocaml
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 expression?

\[
[ \text{(fun } (x:\text{int}) \rightarrow x + 1);x - 1) ]
\]

1. \text{int}
2. \text{int} \rightarrow \text{int}
3. \text{(int} \rightarrow \text{int}) \text{ list}
4. \text{int list} \rightarrow \text{int list}
5. \text{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", 215559092), ... ]
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?
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
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

```ml
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”.
What is the value of this expression?

```
transform String.uppercase ["a";"b";"c"]
```

1. []
2. [“a”; “b”; “c”]
3. [“A”; “B”; “C”]
4. runtime error
What is the value of this expression?

\[
\text{transform } (\text{fun (x:int) -> x > 0}) \quad [0; -1; 1; -2]
\]

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