Programming Languages and Techniques (CIS120)

Lecture 9
February 6, 2019

Lists and Higher-order functions
Lecture notes: Chapter 9
Announcements

• Homework 3 is available now on Codio
  – due *Tuesday, Feb 12 at 11:59pm*
  – Practice with BSTs (last week), generic functions (Monday), HOFs (today) and abstract types (Friday)
  – Read Chapters 7, 8, 9 and 10 of lecture notes
List transformations

A fundamental design pattern using first-class functions
Phone book example

type entry = string * int
let phone_book = [("Pat", 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
let rec helper (f: entry->'b) (p: entry list) : 'b list =
begin match p with
| (e::rest) -> f e :: 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 \( f \) determines
what happens with the entry at the
head of the list
Going even more generic

```ocaml
let rec helper (f: entry->'b) (p: entry list) : 'b list = 
    begin
        match p with
        | (e::rest) -> f e :: 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

```ocaml
let rec helper (f: 'a->'b) (p: 'a list) : 'b list =
  begin
    match p with
    | (e::rest) -> f e :: 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
• used over and over again
• part of OCaml standard library (called List.map)

*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?

\[
\begin{align*}
&\text{transform } (\text{fun (x:int) -> x > 0}) \\
&[0; -1; 1; -2] \\
&[1] \\
&[1; 1; 0; 1] \\
&[\text{false}; \text{false}; \text{true}; \text{false}] \\
&\text{runtime error}
\end{align*}
\]
What is the value of this expression?

```
transform (fun (x: int) -> x > 0)
[0 ; -1; 1; -2]
```

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

**ANSWER: 4**
The ‘fold’ design pattern
Refactoring code, again

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

\[
\text{let rec acid_length (l : acid list) : int =}
\begin{align*}
\text{begin match l with} \\
\text{| []} & \rightarrow 0 \\
\text{| h :: t} & \rightarrow 1 + \text{acid_length t}
\end{align*}
\]

\[
\text{let rec exists (l : bool list) : bool =}
\begin{align*}
\text{begin match l with} \\
\text{| []} & \rightarrow \text{false} \\
\text{| h :: t} & \rightarrow h \text{ || exists t}
\end{align*}
\]

• Can we factor out this 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 result of the recursive call
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
let rec helper (l : bool list) : bool = 
  begin match l with 
  | [] -> false 
  | h :: t -> h || helper t 
  end 

let exists (l : bool list) = helper l 

let rec helper (l : acid list) : int = 
  begin match l with 
  | [] -> 0 
  | h :: t -> 1 + helper t 
  end 

let acid_length (l : acid list) = helper l
let rec helper (l : bool list) : bool =
    begin match l with
    | [] -> false
    | h :: t -> h || helper t
    end

let exists (l : bool list) = helper l

let rec helper (l : acid list) : int =
    begin match l with
    | [] -> 0
    | h :: t -> 1 + helper t
    end

let acid_length (l : acid list) = helper l
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
```
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 (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
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

- 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)

```ocaml
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
```
How would you rewrite this function using fold? What should be the arguments for base and combine?

1. combine is: (fun (h:int) (acc:int) -> acc + 1)
   base is: 0
2. combine is: (fun (h:int) (acc:int) -> h + acc)
   base is: 0
3. combine is: (fun (h:int) (acc:int) -> h + acc)
   base is: 1
4. sum can't be written with fold.
How would you rewrite this function

```
let rec sum (l : int list) : int =
begin match l with
  | []  -> 0
  | h :: t -> h + sum t
end
```

using fold? What should be the arguments for base and combine?

1. combine is: \( (\text{fun} (h:\text{int}) (acc:\text{int}) \rightarrow acc + 1) \)
   base is: \( 0 \)

2. combine is: \( (\text{fun} (h:\text{int}) (acc:\text{int}) \rightarrow h + acc) \)
   base is: \( 0 \)

3. combine is: \( (\text{fun} (h:\text{int}) (acc:\text{int}) \rightarrow h + acc) \)
   base is: \( 1 \)

4. sum can’t be written with fold.

Answer: 2
How would you rewrite this function

```
let rec reverse (l : int list) : int list =
    begin
    match l with
    | [] -> []
    | h :: t -> reverse t @ [h]
    end
```

using fold? What should be the arguments for base and combine?

1. combine is: (fun (h:int) (acc:int list) -> h :: acc)
   base is: 0

2. combine is: (fun (h:int) (acc:int list) -> acc @ [h])
   base is: 0

3. combine is: (fun (h:int) (acc:int list) -> acc @ [h])
   base is:

4. reverse can't be written by with fold.
How would you rewrite this function using fold? What should be the arguments for base and combine?

1. combine is: \( (\text{fun } (h:\text{int}) (\text{acc:\text{int list}}) \rightarrow h :: \text{acc}) \)
   base is: \( 0 \)

2. combine is: \( (\text{fun } (h:\text{int}) (\text{acc:\text{int list}}) \rightarrow \text{acc @ [h]}) \)
   base is: \( 0 \)

3. combine is: \( (\text{fun } (h:\text{int}) (\text{acc:\text{int list}}) \rightarrow \text{acc @ [h]}) \)
   base is: \( [] \)

4. reverse can’t be written by with fold.

Answer: 3
Functions as Data

• We’ve seen a number of ways in which functions can be treated as data in OCaml

• Everyday programming practice offers many more examples
  – objects bundle “functions” (a.k.a. methods) with data
  – iterators (“cursors” for walking over data structures)
  – event listeners (in GUIs)
  – etc.

• Also heavily used at “large scale”: Google’s MapReduce
  – Framework for transforming (mapping) sets of key-value pairs
  – Then “reducing” the results per key of the map
  – Easily distributed to 10,000 machines to execute in parallel!