Programming Languages and Techniques (CIS120)

Lecture 9
September 19, 2018

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

• Homework 3 is available now on Codio
  – due *Tuesday, Sep 25 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 = [ ("Steve", 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 determines what happens with the entry at the head of the list
Going even more generic

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

```ocaml
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
```

'\texttt{a} stands for (\texttt{string}*\texttt{int})

'\texttt{b} stands for \texttt{int}

\texttt{snd} : (\texttt{string}*\texttt{int}) \to \texttt{int}
Transforming Lists

```ocaml
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?

\[
\begin{align*}
[0; -1; 1; -2] \\
[1] \\
[1; 1; 0; 1] \\
[\text{false; false; true; false}] \\
\text{runtime error}
\end{align*}
\]
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?

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

`base case:`
Simple answer when the list is empty

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

`combine step:`
Do something with the head of the list and the result of the recursive call

• Can we factor out this pattern using first-class functions?
let rec acid_length (l : acid list) : int =
  begin match l with
    [ ] -> 0
    h :: t -> 1 + acid_length t
  end

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

```plaintext
let rec helper (base : bool) (l : bool list) : bool =
begin match l with
  | [] -> base
  | h :: t -> h || helper base t
end

let acid_length (l : acid list) = helper 0 l
```

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

CIS120
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 | [] -> 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 -> l + 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 acid_length (l : acid list) =
helper (fun (h:acid) (acc:int) -> 1 + acc) 0 l

let exists (l : bool list) =
helper (fun (h:bool) (acc:bool) -> h || acc) false 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
```
Rewrite using 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: `(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 using fold? What should be the arguments for base and combine?

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

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

3. combine is: \( \text{(fun (h:int) (acc:int) -> h + acc)} \)
   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: \[
   \emptyset
   \]
2. combine is: (fun (h:int) (acc:int list) -> acc @ [h])
   base is: \[
   \emptyset
   \]
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

```ocaml
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.

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!
Abstract Collections
Are you familiar with the idea of a set from mathematics?

In math, we typically write sets like this:

- \( \emptyset \) \{1,2,3\} \{true,false\}

with operations:

- \( S \cup T \) for union and
- \( S \cap T \) for intersection;

we write \( x \in S \) for the predicate

“\( x \) is a member of the set \( S \)”
A set is an abstraction

- A set is a collection of data
  - we have operations for forming sets of elements
  - we can ask whether elements are in a set
- A set is a lot like a list, except:
  - Order doesn't matter
  - Duplicates don't matter
    - An element’s presence or absence in the set is all that matters...
  - It isn't built into OCaml
- Sets show up frequently in applications
  - Examples: set of students in a class, set of coordinates in a graph, set of answers to a survey, set of data samples from an experiment, ...
Abstract type: set

• A BST can *implement (represent)* a set
  – there is a way to represent an empty set (Empty)
  – there is a way to list all elements contained in the set (inorder)
  – there is a way to test membership (lookup)
  – Can define union/intersection (with insert and delete)

• BSTs are *not the only* way to implement sets
Three Example Representations of Sets

**Alternate representation:**

- **BST:**
  - Concrete representation:
    - 1
    - <
    - 0
    - 3
  - Abstract view:
    - \{1, 0, 3\}

- **Alternate representation:**
  - Unsorted linked list:
    - 3::0::1::null

- **Alternate representation:**
  - Reverse sorted array with index of next slot:
    - [3, 1, 0, x, x]
Abstract types (e.g. set)

- An abstract type is defined by its *interface* and its *properties*, not its representation.

**Interface:** defines operations on the type
- There is an empty set
- There is a way to add elements to a set to make a bigger set
- There is a way to list all elements in a set
- There is a way to test membership

**Properties:** define how the operations interact with each other
- Elements that were added can be found in the set
- Adding an element a second time doesn’t change the elements of a set
- Adding in a different order doesn’t change the elements of a set

- **Any** type (possibly with invariants) that satisfies the interface and properties can be a set.
Sets in OCaml
The name of the signature.

module type SET = sig

  type 'a set

  val empty : 'a set
  val add : 'a -> 'a set -> 'a set
  val member : 'a -> 'a set -> bool
  val equals : 'a set -> 'a set -> bool
  val set_of_list : 'a list -> 'a set

end

The sig keyword indicates an interface declaration

Type declaration has no “right-hand side” – its representation is abstract!

The interface members are the (only!) means of manipulating the abstract type.

Signature (a.k.a. Interface): defines operations on the type
Implementing sets

- There are many ways to implement sets.
  - lists, trees, arrays, etc.

- **How do we choose which implementation?**
  - Depends on the needs of the application...
  - How often is ‘member’ used vs. ‘add’?
  - How big can the sets be?

- Many such implementations are of the flavor “a set is a ... with some invariants”
  - A set is a *list* with no repeated elements.
  - A set is a *tree* with no repeated elements
  - A set is a *binary search tree*
  - A set is an *array of bits*, where 0 = absent, 1 = present

- **How do we preserve the invariants of the implementation?**
A *module* implements an interface

- An implementation of the set interface will look like this:

```plaintext
module BSTSet : SET = struct

(* implementations of all the operations *)

end
```

**Name of the module**

**Signature that it implements**

**The `struct` keyword indicates a module implementation**
Implement the Set Module

```ocaml
module BSTSet : SET = struct

  type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree

  type 'a set = 'a tree

  let empty : 'a set = Empty

end
```

• The implementation has to include everything promised by the interface
  – It can contain more functions and type definitions (e.g. auxiliary or helper functions) but those cannot be used outside the module
  – The types of the provided implementations must match the interface
Abstract vs. Concrete BSTSet

```ocaml
module BSTSet : SET = struct
  type 'a tree = ...
  type 'a set = 'a tree
  let empty : 'a set = Empty
  let add (x:'a) (s:'a set) :'a set =
    ...
    (* can treat s as a tree *)
end

(* A client of the BSTSet module *)

;; open BSTSet

let s : int set
  = add 0 (add 3 (add 1 empty))
```

Concrete representation:

```
 s = 1 < 0 > 3
```

Abstract view:

```
{ 1, 0, 3 }
```
module ULSet : SET =
struct
  type 'a set = 'a list
  let empty : 'a set = []
  ...
end

A different definition for the type set
Abstract vs. Concrete ULSet

`s = 0::3::1::[]`

Concrete representation

Abstract view

```
module ULSet : SET = struct
  type 'a set = 'a list
  let empty : 'a set = []
  let add (x:'a) (s:'a set) :'a set =
    x::s (* can treat s as a list *)
end

(* A client of the ULSet module *)
;; open ULSet

let s : int set
  = add 0 (add 3 (add 1 empty))
```

Client code doesn’t change!