Programming Languages and Techniques (CIS120)

Lecture 10

February 5th, 2016

Abstract types: sets

Lecture notes: Chapter 10
What is the value of this expression?

```ocaml
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
Announcements

• Homework 3 is available
  – due *Tuesday at midnight*

• Read Chapter 10 of lecture notes

• Midterm 1
  – Register for makeup exam on course website
List processing

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
```

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

```ocaml
let rec acid_length (l : acid list) : int =  
begin match l with  
| [] -> 0  
| h :: t -> 1 + acid_length t  
end
```
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
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
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: \((\text{fun} \ (h:\text{int}) \ (\text{acc}:\text{int}) \Rightarrow \text{acc} + 1)\)
   base is: 0

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

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

1. sum can’t be written by 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: \((\text{fun } (h:\text{int}) (\text{acc:}\text{int list}) \rightarrow h :: \text{acc})\)
   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: \([]\)

1. 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

• 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 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?
1. yes
2. no

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
“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
  – It isn't built into OCaml

  An element’s presence or absence in the set is all that matters...

• 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)
  – could define union/intersection (insert and delete)

• Order doesn't matter
  – We create BSTs by adding elements to an empty BST
  – The BST data structure doesn’t remember what order we added the elements

• Duplicates don't matter
  – Our implementation doesn’t keep track of how many times an element is added
  – BST invariant ensure that each node is unique

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

BST:

Alternate representation: unsorted linked list.

Alternate representation: reverse sorted array with index to next slot.

concrete representation

abstract view

{1, 0, 3}

concrete representation

abstract view

{1, 0, 3}

concrete representation

abstract view

{1, 0, 3}
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 set interface in OCaml (a signature)

```ocaml
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
```

Keyword ‘val’ names values that must be defined and their types.

Type declaration has no “body” – its representation is abstract!
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’ or ‘remove’?
  – How big will the sets need to 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:

```
module ULSet : SET = struct
  ...
  (* implementations of all the operations *)
  ...
end
```
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
Another Implementation

module ULSet : SET =
struct
  type 'a set = 'a list
  let empty : 'a set = []
  ...
end

A different definition for the type set
Testing (and using) sets

- To use the values defined in the set module use the “dot” syntax:

  \[ \text{ULSet.<member>} \]

- Note: Module names must be capitalized in OCaml

```ocaml
let s1 = ULSet.add 3 ULSet.empty
let s2 = ULSet.add 4 ULSet.empty
let s3 = ULSet.add 4 s1

let test () : bool = (ULSet.member 3 s1)
;; run_test "ULSet.member 3 s1" test

let test () : bool = (ULSet.member 4 s3)
;; run_test "ULSet.member 4 s3" test
```
Testing (and using) sets

- Alternatively, use “open” to bring all of the names defined in the interface into scope.

```ocaml
;; open ULSet

let s1 = add 3 empty
let s2 = add 4 empty
let s3 = add 4 s1

let test () : bool = (member 3 s1)
;; run_test "ULSet.member 3 s1" test

let test () : bool = (member 4 s3)
;; run_test "ULSet.member 4 s3" test
```
Abstract types

BIG IDEA: Hide the *concrete representation* of a type behind an *abstract interface* to preserve invariants.

• The interface **restricts** how other parts of the program can interact with the data.
  – Clients must only use what is declared in the SET interface

• Benefits:
  – **Safety**: The other parts of the program can’t break any invariants
  – **Modularity**: It is possible to change the implementation without changing the rest of the program
Does this code type check?

```ocaml
;; open BSTSet
let s1 : int set = Empty
```

1. yes
2. no

Answer: no, the Empty data constructor is not available outside the module