Lecture 11
September 24, 2018

Review: Abstract types
Finite Maps
Announcements

• Homework 3
  – due *Tuesday* at 11:59:59pm

• Homework 4
  – available soon
  – due on October 9th

• Dr. Sheth will be traveling 9/26 and 9/28 (Dr. Fouh will cover)
Announcements

• Midterm 1
  – *October 12*\textsuperscript{th} *in Class*
  – Where? Last Names:
    A – M   Leidy Labs 10 (Here)
    N – Z   College Hall 200
  – Covers lecture material through Chapter 13
  – Review materials (old exams) on course website
  – Review session
    • Wednesday, Oct 10, 6-8pm, Towne 100
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.

Clients of an implementation can only access what is explicitly in the abstract type’s interface.
Finite Maps

Another example of *abstract datatype interfaces* & *concrete implementations*
Motivating Scenario

• Suppose you were writing some course-management software and needed to look up the lab section for a student given the student’s PennKey?
  – Students might add/drop the course
  – Students might switch lab sections
  – Students should be in only *one* lab section

• How would you do it? What data structure would you use?
Each key is associated with a value.

- No two keys are identical
- Values can be repeated

Given the key “stephanie" we want to find / lookup the value 15

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>“stephanie”</td>
<td>15</td>
</tr>
<tr>
<td>“mitch”</td>
<td>05</td>
</tr>
<tr>
<td>“ezaan”</td>
<td>10</td>
</tr>
<tr>
<td>“likat”</td>
<td>15</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Finite Maps

• A finite map (a.k.a. dictionary) is a collection of bindings from distinct keys to values.
  – Operations to add & remove bindings, test for key membership, look up the value bound to a particular key

• Example: a (string, int) map might map a PennKey to the lab section.
  – The map type is generic in two arguments

• Like sets, finite maps appear in many settings to map:
  – domain names to IP addresses
  – words to their definitions (a dictionary)
  – user names to passwords
  – game character unique identifiers to dialog trees
  – ...

CIS120
module type MAP = sig

  type ('k,'v) map

  val empty : ('k,'v) map
  val add : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
  val remove : 'k -> ('k,'v) map -> ('k,'v) map
  val mem : 'k -> ('k,'v) map -> bool
  val get : 'k -> ('k,'v) map -> 'v
  val entries : ('k,'v) map -> ('k * 'v) list
  val equals : ('k,'v) map -> ('k,'v) map -> bool

end
Tests for Finite Map abstract type

;; open Assert

(* Specifying the properties of the MAP abstract type via test cases. *)

(* A simple map with one element. *)
let m1 : (int, string) map = add 1 "uno" empty

(* list entries for this simple map *)
;; run_test "entries m1" (fun () -> entries m1 = [(1, "uno")])

(* access value for key in the map *)
;; run_test "find 1 m1" (fun () -> (get 1 m1) = "uno")

(* find for value that does not exist in the map? *)
;; run_failing_test "find 2 m1" (fun () -> (get 2 m1) = "dos")

let m2 : (int, string) map = add 1 "un" m1

(* find after redefining value, should be new value *)
;; run_test "find 1 m2" (fun () -> (get 1 m2) = "un")

(* entries after redefining value, should only show new value *)
;; run_test "entries m2" (fun () -> entries m2 = [(1, "un")])
module Assoc : MAP = struct

(* Represent a finite map as a list of pairs. *)
(* Representation invariant: *)
(* - no duplicate keys (helps get, remove) *)
(* - keys are sorted (helps equals, helps get) *)

type ('k,'v) map = ('k * 'v) list

let empty : ('k,'v) map = []

let rec mem (key:'k) (m : ('k,'v) map) : bool =
  begin match m with
  | [] -> false
  | (k,v)::rest ->
    (key >= k) &&
    ((key = k) || (mem key rest))
  end

;; run_test "mem test" (fun () -> mem "b" ["a",3; "b",4])
Implementation: Ordered Lists

let rec get (key:'k) (m : ('k,'v) map) : 'v =
begin match m with
| []    -> failwith "key not found"
| (k,v)::rest ->
    if key < k then failwith "key not found"
    else if key = k then v
    else get key rest
end

let rec remove (key:'k) (m : ('k,'v) map) : ('k,'v) map =
begin match m with
| []    -> []
| (k,v)::rest ->
    if key < k then m
    else if key = k then rest
    else (k,v)::remove key rest
end
Completing module implementation

finiteMap.ml
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
  - Type checking ensures that the **only** way to create a set is with the operations in the interface
  - If all operations preserve invariants, then all sets in the program must satisfy invariants
  - Example: all BST-implemented sets must satisfy the BST invariant, therefore the lookup function can assume that its input satisfies the invariant

- **Benefits:**
  - **Safety:** The other parts of the program can’t cause bugs in the set implementation
  - **Modularity:** It is possible to change the implementation without changing the rest of the program
Summary: Abstract Types

- Different programming languages have different ways of letting you define abstract types

- At a minimum, this means providing:
  - A way to specify (write down) an interface
  - A means of hiding implementation details (*encapsulation*)

- In OCaml:
  - Interfaces are specified using a *signature* or *interface*
  - Encapsulation is achieved because the interface can *omit* information
    - type definitions
    - names and types of auxiliary functions
  - Clients *cannot* mention values or types not named in the interface
You’ve already been using signatures and modules in OCaml.

A series of type and val declarations stored in a file foo.mli is considered as defining a signature F00

A series of top-level definitions stored in a file foo.ml is considered as defining a module Foo
module type FOO = sig
  type t
  val z : t
  val f : t -> int
end

module Foo : FOO = struct
  type t = int
  let z : t = 0
  let f (x : t) : int = x + 1
end

module Test = struct
  ;; open Foo
  ;; print_int (Foo.f Foo.z)
end