Programming Languages and Techniques (CIS120e)

Lecture 8

Sep 24, 2010

Interfaces, Abstract Datatypes and Maps
Announcements

• Homework 3 is available on the web.
  – Due next Wednesday at 11:59:59pm
  – Practice with BSTs, generic functions, abstract datatypes
Design Process

1. Understand the problem
   What are the relevant concepts and how do they relate?
2. Formalize the interface
   How should the program interact with its environment?
3. Write test cases
   How does the program behave on typical inputs? On unusual ones? On erroneous ones?
4. Implement the required behavior
   Often by decomposing the problem into simpler ones and applying the same recipe to each

• We’ve seen:
  – Recursive datatypes & recursion as a “design pattern”
  – BSTs as an example that uses that design pattern
  – BSTs are also an example of a datatype + invariant
A Design Problem

• Suppose that at Step 2 (“Formalize the Interface”) we decide that we need to process a *set* of values.
  
  – Examples: *set* of students in a class, *set* of coordinates in a graph, *set* of answers to a survey, *set* of samples from Gulf oil spill probes, ...

• Given what you know of OCaml, what would you do?
Many ways to implement sets

• The concept of a “set” arises in many applications.
• There are many ways to implement sets.
  – lists, trees, arrays, etc.
• 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

• What operations do we want to perform on the sets?
Abstract Datatypes

BIG IDEA: Hide the *concrete representation* of a datatype behind an *abstract interface*.

- **Example:**
  - concrete ‘set’ representation – the *implementation* – is a list or a tree
  - abstract interface defines the operations in terms of a ‘set’ type

- The interface restricts how other parts of the program can interact with the data.

- **Benefits:**
  - **Safety:** The other parts of the program can’t break any invariants that are being maintained behind the interface.
  - **Modularity:** It is possible to change the implementation of the abstract datatype without changing the rest of the program.
Defining 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 ML interface file (.mli)
  – Encapsulation is achieved because the interface can *omit* definitions.
  – Clients can’t mention values not named in the interface.

*In Java, interfaces can also be written down explicitly and encapsulation is achieved by public/private modifiers on object fields. (We’ll cover this in detail later.)
Example Module Interface: set.mli

```ocaml
type 'a set

val empty : 'a set
val is_empty : 'a set -> bool
val mem : 'a -> 'a set -> bool
val add : 'a -> 'a set -> 'a set
val union : 'a set -> 'a set -> 'a set
val remove : 'a -> 'a set -> 'a set
val filter : ('a -> bool) -> 'a set -> 'a set
val equal : 'a set -> 'a set -> bool
val elements : 'a set -> 'a list
```

- OCaml lets you write *interfaces* (as above) in a .mli file
- The corresponding *implementation* goes in the .ml file
Function Types

• In OCaml, the type of functions from input $t$ to output $u$ is written:

$$t \rightarrow u$$

• Functions with multiple arguments use multiple arrows

• Here are some examples we have already seen:

```ocaml
size : tree -> int
hamming_distance : helix -> helix -> int
acids_of_helix : helix -> acids list
length : 'a list -> int
zip : 'a list -> 'b list -> ('a*'b) list
lookup : tree -> int -> bool
```
**Module Implementation: set.ml**

```ml

**type** 'a tree =  (* concrete implementation *)
 | Empty
 | Node of 'a tree * 'a * 'a tree

**type** 'a set = 'a tree (* definition hidden by set.mli *)

**let** empty : 'a set = Empty
**let** is_empty (s:'a set) : bool =
  begin match s with
  | Empty -> true
  | _ -> false
  end

... 
```

- The implementation has to include all of the interface values
  - It can contain *more* functions and type definitions (e.g. auxiliary functions) but those cannot be used outside the module
  - The types of the provided implementations must match the interface
A Client of the Set Module

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

;; assert_eq "" (Set.mem 3 s3) true
;; assert_eq "" (Set.mem 4 s3) true
```

- To use the values defined in the set module use the “dot” syntax:
  ```ocaml
  Set.<member>
  ```
- Alternatively, use “open Set;;” at the top of a file to bring all of the names defined in the interface into scope.
- Note: Module names are always capitalized in OCaml
First-class Functions

• Consider the `filter` member of the set interface:

```plaintext
val filter : ('a -> bool) -> 'a set -> 'a set
```

Filter takes a function value as an argument!

• Desired behavior: return the subset of elements of `s` for which `f` is true. In math notation:

```
Set.filter f s = {x | x ∈ s and f(x)=true}
```
(*) is_even : int -> bool *)
let is_even (n:int) : bool =
  n mod 2 = 0

let s = ... (* build the set {1,2,3,4,5,6} *)

let evens_of_s = Set.filter is_even s
(* evens_of_s is the set {2,4,6} *)

Pass is_even as an argument to Set.filter.
Using a First-Class Function

• How do we implement something like Set.filter?*
• Example: apply a function pointwise to each element of a list:

\[
(* \text{map} : (\text{a} \to \text{b}) \to \text{a list} \to \text{b list} *)
\]

\[
\text{let rec \text{map} (f:\text{a} \to \text{b}) (l:\text{a list}) : \text{b list} =}
\]

\[
\begin{align*}
&\text{begin match } l \text{ with} \\
&| [] \to [] \\
&| \text{h}::t \to (f \text{ h})::(\text{map f t}) \\
&\text{end}
\end{align*}
\]

Just apply the function \(f\) as usual.

• Example of using \text{map}:

\[
\text{map \text{is\_even} [1;2;3;4;5] = [false;true;false;true;false]}
\]

*You will implement two versions of Set.filter in homework 3.
Case Study: Finite Maps
Motivating Scenario

• Suppose you were writing some course-management software and needed to lookup 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 if you also needed to lookup the student’s major?
• Or a list of the student’s project partners?
Finite Maps*

• A finite map is a collection of bindings from distinct keys to values.
  – Operations to add & remove bindings, test for key membership, lookup a value by its key

• Example: an \((ID, \text{int})\) map might map a PennKey ID to the lab section.

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

*There is an unfortunate clash of terminology: don’t confuse the “finite map” datastructure here, with the “map” function of slide 14.
type ('k,'v) map

val empty : ('k,'v) map
val is_empty : ('k,'v) map -> bool
val mem : 'k -> ('k,'v) map -> bool
val find : 'k -> ('k,'v) map -> 'v
val add : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
val remove : 'k -> ('k,'v) map -> ('k,'v) map
val keys : ('k,'v) map -> 'k list
val values : ('k,'v) map -> 'v list
val bindings : ('k,'v) map -> ('k*'v) list
Demo: Implementing Maps with Lists