Programming Languages and Techniques (CIS120)

Lecture 8

Feb 5, 2014

Abstract Types: Sets

Modules and Interfaces

Announcements

- Homework 3 is available
  - Due TUESDAY, February 11th at 11:59:59pm
  - Practice with BSTs, generic functions, and *abstract types*

- If you added CIS 120 recently, make sure that you can see your scores online.
  - If you get feedback about your scores, you are in our database.
  - If not, please send mail to tas120@lists.seas.upenn.edu
  - If you see unsubmitted "quizzes", you may need to register your clicker

- Read chapter 9 of the lecture notes

Do these two declarations produce the same BST?

```
let t1 = insert Empty 2
let t2 = insert (insert Empty 2) 2
```

1. yes
2. no

Answer: yes

Do these two declarations produce the same BST?

```
let t1 = insert (insert (insert Empty 2) 1) 3
let t2 = insert (insert (insert Empty 2) 3) 1
```

1. yes
2. no

Answer: yes

Are you familiar with the idea of a *set* from mathematics?

1. yes
2. no

Answer: about 70% reported yes

Abstract Collections

A *set* is an abstraction

- A set is a collection of data
  - In math, we typically write sets like this: Ø {1,2,3} {true,false} with operations like: $S \cup T$ or $S \cap T$ for union and intersection; we write $x \in S$ to mean that "x is a member of the set S"
- A set is a lot like a list, except:
  - Order doesn't matter
  - Duplicates don't matter
  - *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 binary search tree is an *implementation* of a *set*
  - *there is an empty set*
  - *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 with 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

- BSTs are not the only way to implement sets, let's generalize

Abstract type: set

- An abstract type is defined by its *interface* and its *properties*
- The interface defines how sets can be created and used
  - 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 remove elements from the set to make a smaller set
  - There is a way to test membership
- The properties define how these operations interact with eachother
  - Elements that were added can be found in the set
  - Adding a twice doesn't change the elements of a set
  - Adding in a different order doesn't change the elements of a set
  - ….
- Any type that can implement this interface while satisfying the properties can be a set

Sets in action

A design problem

As a high-school student, Stephanie had the job of reading books and finding which words, out of a list of the 1000-most common SAT vocabulary words, appeared in a particular book. She enjoyed being paid to read, but she would have enjoyed being paid to program more. How could she have automated this task?

1. What are the important concepts or *abstractions* for this problem?
   - The list of words that appear in a book
   - The set of 1000-most common SAT words
   - The set of words from the list that are contained in the set

2. Formalize the Interface

- Suppose we had a generic type of sets:

`'a set`

(We'll get to the details of that in a moment.)

- We can formalize the interface for our problem:

```
let countVocab (text : string list)
               (vocab : string set)
             : int  =
  failwith "write me"
```

3. Write Test Cases

```
let vocab : string set =
  list_to_set ["induce"; "crouching"; "reprieve";
               "indigent"; "arrogate"; "coalesce";
               "temerity"]

let text1 = ["i"; "looked"; "up"; "again"; "at";
  "the"; "crouching"; "white"; "shape"; "and";
  "the"; "full"; "temerity"; "of"; "my"; "voyage"]

let test () : bool =
     countVocab text1 vocab = 2
;; run_test "countVocab" test
```

Test cases specify the *interface* and the *properties* of the necessary abstractions.

4. Implement the Required Behavior

```
let countVocab (text : string list)
               (vocab : string set)
             : int  =
  failwith “write me”
```

- Easy recursive programming task
  - (we’ll leave the details to you)
- Requires set membership test

```
let member (x:'a) (s:'a set) : bool =
   failwith "unimplemented"
```

The set interface in OCaml (a *signature*)

```
module type Set = sig

    type 'a set

    val empty : 'a set
    val add    : 'a -> 'a set -> 'a set
    val remove : 'a -> 'a set -> 'a set
    val list_to_set : 'a list -> 'a set
    val member : 'a -> 'a set -> bool
    val elements : 'a set -> 'a list

end
```

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

Aside: Function Types

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

```
t -> u
```

- Functions with multiple arguments are written with multiple arrows
- Examples:

```
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
insert : ‘a tree -> ‘a -> ‘a tree
```

A *module* of sets

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

Name of the module

Signature that it implements

```
module Myset : Set = struct
  …
  (* implementations of all the operations *)
  …
end
```

Testing (and using) sets

- To use the values defined in the set module use the "dot" syntax:
  `Myset.`*<member>*
- Note: Module names are always capitalized in OCaml

```
let s1 = Myset.add 3 Myset.empty
let s2 = Myset.add 4 Myset.empty
let s3 = Myset.add 4 s1

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

let test () : bool = (Myset.member 4 s3) = true
;; run_test "Myset.member 4 s3" test
```

Testing (and using) sets

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

```
;; open Myset

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

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

let test () : bool = (member 4 s3) = true
;; run_test "Myset.member 4 s3" test
```

Implementing sets

- There are many ways to implement sets.
  - lists, trees, arrays, etc.
- **How do we choose which implementation?**

- 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?**

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.
- 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

Set signature

```
module type Set = sig

  type 'a set

  val empty  : 'a set
  val add    : 'a -> 'a set -> 'a set
  val remove : 'a -> 'a set -> 'a set
  val list_to_set : 'a list -> 'a set
  val member : 'a -> 'a set -> bool
  val elements : 'a set -> 'a list

end
```

Type declaration has no "body" – its representation is *abstract*!

Implement the set Module

```
module MySet : Set =
struct
  type 'a tree =
   | Empty
   | Node of 'a tree * 'a * 'a tree

  type 'a set = 'a tree

  let empty : 'a set = Empty
  …
end
```

Module must define the type declared in the signature

- The implementation has to include everything promised by the interface
  - 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

Another Implementation

```
module MySet2 : Set =
struct

  type 'a set = 'a list

  let empty : 'a set = []
  …

end
```

A different definition for the type set

Does this code type check?

```
;; open MySet
let s1 : int set = Empty
```

1. yes
2. no

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

Does this code type check?

```
;; open MySet
let s1 : int set = add 1 empty
```

1. yes
2. no

Answer: yes

Does this code type check?

```
;; open MySet
let s1 : int tree = add 1 empty
```

1. yes
2. no

Answer: no, add constructs a set, not a tree

If a module works and starts with:

```
;; open MySet
```

will it continue to work if we change that line to:

```
;; open MySet2
```

1. yes
2. no

Answer: yes (caveat: times may be different)