Programming Languages and Techniques (CIS120)

Lecture 11

February 8, 2016

Abstract types
Sets and Finite Maps

Announcements

- Homework 3
  - due *Tuesday* at 11:59:59pm
  - next HW available after the midterm

- Midterm 1
  - Scheduled Tuesday evening*, February 16th*
  - Covers lecture material through Chapter 10
  - Review materials (old exams) on course website
  - Review session, Sunday evening ❤️ 6-8 PM
  - Register for the make-up exam *by Sunday*
  - More details on Wednesday

Set *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 "body" – its representation is *abstract*!

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

Implement the `set` Module

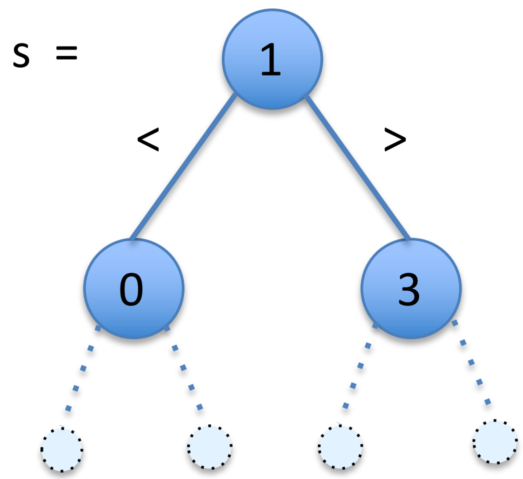
```
module BSTSet : SET = struct

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

  type 'a set = 'a tree

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

The **struct** keyword indicates a module implementation

Module must define (give a *concrete representation* to) the type declared in the signature

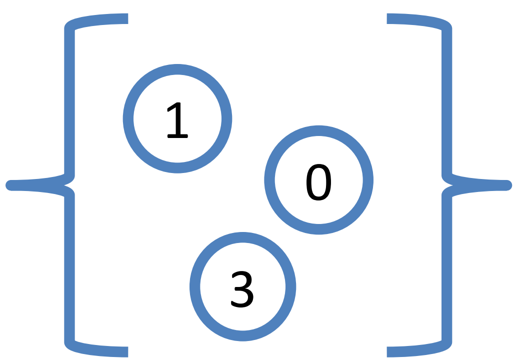
- The implementation has to include everything promised by the interface
  - It can contain *more* functions and type definitions (e.g. auxiliary or helper functions) but those cannot be used outside the module
  - The types of the provided implementations must match the interface

Abstract vs. Concrete BSTSet



concrete representation

abstract view




```
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
```

```
module type SET = sig
  type 'a set
  val empty : 'a set
  val add   : 'a -> 'a set -> 'a set
end
```

```
(* A client of the BSTSet module *)
;; open BSTSet

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

Another Implementation

```
module ULSet : SET =
struct

  type 'a set = 'a list

  let empty : 'a set = []
  …

end
```

A different definition for the type set

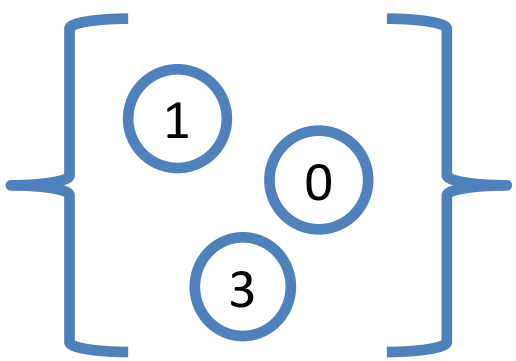
Abstract vs. Concrete ULSet

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

```
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
```

concrete representation

abstract view




```
module type SET = sig
  type 'a set
  val empty : 'a set
  val add   : 'a -> 'a set -> 'a set
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!

```
module type SET = sig
  type 'a set
  val empty : 'a set
  val add   : 'a -> 'a set -> 'a set
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
```

Does this code type check?

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

1. yes
2. no

Answer: yes

```
module type SET = sig
  type 'a set
  val empty : 'a set
  val add   : 'a -> 'a set -> 'a set
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
```

Does this code type check?

```
;; open BSTSet
let s1 = add 1 empty
let i1 = begin match s1 with
         | Node (_,k,_) -> k
         | Empty -> failwith "impossible"
         end
```

1. yes
2. no

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

```
module type SET = sig
  type 'a set
  val empty : 'a set
  val add   : 'a -> 'a set -> 'a set
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
  let size (t : 'a tree) : int = …
  …
end
```

Does this code type check?

```
;; open BSTSet
let s1 = add 1 empty
let i1 = size s1
```

1. yes
2. no

Answer: no, cannot access helper functions outside the module

```
module type SET = sig
  type 'a set
  val empty : 'a set
  val add   : 'a -> 'a set -> 'a set
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
```

Does this code type check?

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

1. yes
2. no

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

If a client module works correctly and starts with:

```
;; open ULSet
```

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

```
;; open BSTSet
```

assuming that ULSet and BSTSet both implement SET and satisfy all of the set properties?

1. yes
2. no

Answer: yes (caveat: performance may be different)

```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
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
```

Is is possible for a client to call **`member`** with a tree that is not a BST?

1. yes
2. no

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

How comfortable to you feel with the concept of an *invariant*?

1. Totally confused (I have no idea what they are)
2. Somewhat unsure (I can only give an example)
3. It's beginning to make sense
4. Pretty confident (I understand how they're used)
5. I've completely got it (I could design my own)

Finite Map Demo

Using module signatures to preserve
data structure invariants

finiteMap.ml

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?

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 a value by its 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:
  - 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
  - …

Finite Map *signature*

```
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
```

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