Abstract types: Sets
Chapter 10
• Homework 3
  – due *Tuesday* at 11:59:59pm

• Midterm 1
  – *Friday, February 15, in class*
  – Covers material through Chapter 11
  – Review materials (old exams) on course website
  – Review session
    • Wednesday, Feb 13, 6-8pm, Towne 100
Abstract Collections
Abstract types (e.g. set)

• An abstract type is defined by its *interface* and its *properties*, not its representation

• **Interface**: defines the type and operations
  – There is a type of sets
  – 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 that satisfies the interface and properties can be a set
Sets in OCaml

OCaml directly supports the declaration of abstract types via *signatures*
The name of the signature

The `sig` keyword indicates an interface declaration

Type declaration has no “right-hand side” – its representation is abstract!

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

Signature (a.k.a. interface): defines operations on the type
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’?
  – How big can the sets 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:

```plaintext
module BSTSet : SET = struct
...
  (* implementations of type and operations *)
...
end
```

- **Name of the module**
- **Signature that it implements**
- **The `struct` keyword indicates a module implementation**
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 implementation must 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 signature

Module must define (give a concrete representation to) the type declared in the signature
Abstract vs. Concrete BSTSet

Abstract view

concrete representation

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

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

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

```haskell
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

```
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
```

```
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 *)

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

Client code doesn’t change!

Abstract view

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

Concrete representation

```
{1, 0, 3}
```
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.

```
;; 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
```
Does this code typecheck?

```
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

;; open BSTSet
let s1 : int set = add 1 empty
```
Does this code type check?

```ocaml
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

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

1. yes
2. no

Answer: yes
Does this code typecheck?

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

Yes

No
Does this code type check?

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

;; open BSTSet
let s1 = add 1 empty
let i1 = size s1
Does this code type check?

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

1. yes
2. no

Answer: no, cannot access helper functions outside the module
Does this code typecheck?

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

;; open BSTSet
let s1 : int set = Empty
Does this code type check?

```ocaml
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

;; 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:

```plaintext
;; open ULSet
```

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

```plaintext
;; open BSTSet
```

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

1. yes
2. no

Answer: yes (though performance may be different)
Is it possible for a client to call `member` with a tree that is not a BST?

1. yes
2. no

No: the BSTSet operations preserve the BST invariants. There is no way to construct a non-BST tree using the interface.
Completing ULSet

See sets.ml
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
Bonus Material: OCaml Details

module and interface files
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 `FOO`

A series of top-level definitions stored in a file `foo.ml` is considered as defining a module `Foo`
foo.mli

```
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
```