Programming Languages and Techniques (CIS1200)

Lecture 10

Abstract types: Sets

Chapter 10

Announcements (1)

- Homework 3 available, due Tuesday at 11.59pm
  - Practice with BSTs, generic functions, first-class functions, and abstract types
  - *Start early!*
    - *Problems 1-4 can be done already*
    - *Problems 5-8 can be done after class today*

- Reading: Chapters 8, 9, and 10 of the lecture notes

- Please complete the Intro Survey (details on Ed)

Announcements (2)

- Midterm 1: Friday, Feb 14th
  - Coverage: up to Wednesday, Feb 12th (Chapters 1-10)
  - During lecture
    Last names: A – Z Meyerson Hall B1
  - 60 minutes; closed book, 1 sheet handwritten (not ipad) notes
  - Review Material
    - old exams on the web site (“schedule” tab)
  - Review Session
    - Wednesday, Feb 12, 7:00-9:00pm, Towne 100 (will be recorded)
    - Review Videos will be posted this weekend

Sets as Abstract Types

Mathematical Sets

In math, we typically write sets like this:

$\emptyset$ {1,2,3,4}   {true,false} {X,Y,Z}

with operations

$S \cup T$ for *union* and

$S \cap T$ for *intersection*;

and write $x \in S$ for the predicate

"x is a member of the set S"

Set properties

Certain facts hold of set operations:

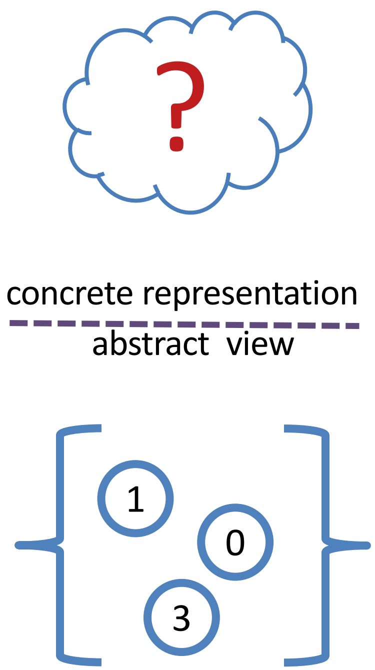
1. If $x \in S$ then $x \in (S \cup T)$ for any other set T.
2. If $x \in T$ then $x \in (S \cup T)$ for any other set S.
3. $x \notin \emptyset$ (the empty set contains no elements)
4. $x \in \{x\}$ (the element x is in its singleton set)

5. $S \cup T = T \cup S$ (union is commutative)
6. $(S \cup T) \cup V = S \cup (T \cup V)$ (union is associative)
7. $S \cup S = S$ (union is idempotent)
8. $S \cup \emptyset = S$ ($\emptyset$ is the “right unit” of union)

…

A Set is an Abstract Type

- 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 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 listing of elements
  - Adding elements in a different order doesn’t change the listing of elements
- *When we use a set, we can forget about the representation!*



This *is* abstraction!!

Sets in OCaml

OCaml directly supports the declaration of abstract types via *signatures*

Set *Signature*

The name of the signature

The `sig` keyword indicates an interface declaration

Type declaration has no "right-hand side" – its representation is *abstract*!

```
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 interface members are the (only!) means of manipulating the set type.

Signature (a.k.a. interface): defines operations on the type

Math notation vs. Code

∅	~	`empty`	`: 'a set`
{x}	~	`add x empty`	`: 'a set`
{x} ∪ S	~	`add x s`	`: 'a set`
x ∈ S	~	`member x s`	`: bool`
{x} ∪ {y} = {y} ∪ {x}	~	`equals (add x (add y empty)) (add y (add x empty))`	`: bool`

```
equals
  (add x (add y empty))
  (add y (add x empty))
                        : bool
```

Examples of corresponding notions in math vs. OCaml

Implementing sets

- There are many ways to implement sets
  - lists, trees, arrays, etc.
  - each of these could be a suitable *representation type*
- *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 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*
- *How do we preserve the invariants of the implementation?*

Invariant: a property that remains unchanged when a specified transformation is applied.

A *module* implements an interface

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

Name of the module

Signature that it implements

The `struct` keyword indicates a module implementation

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

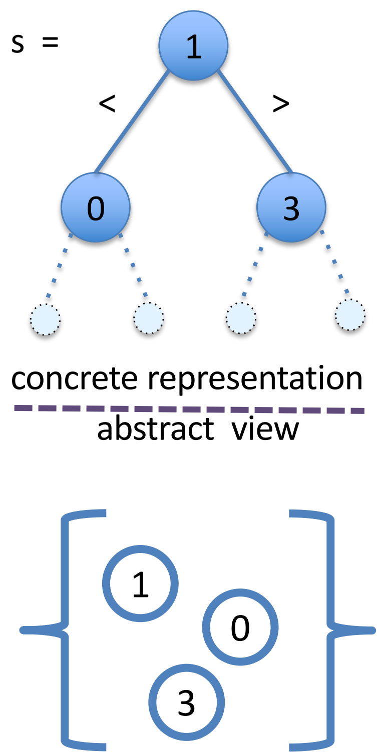
Implement the BSTSet Module

```
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

  …  (* implementations of add,
        member, etc. *)
end
```

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

- 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 implementations must match the signature

Abstract vs. Concrete BSTSet




```
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 *)
(* Cannot treat a set as a tree  *)
;; open BSTSet

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

A *Different* 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

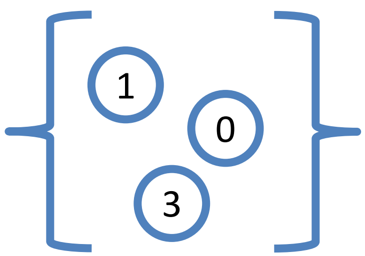
```
s  =   0::3::1::[]
```

```
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 *)
(* Cannot treat a set as a list *)
;; open ULSet

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

Client code doesn’t change!

Implementing ULSet

See sets.ml

Testing (and using) sets

- Use "open" to bring all names defined in the interface into scope
- Any names in the interface that were already in scope are shadowed

```
;; 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
```

Brings the type **'a set** and values **empty**, **add**, and **member** into scope

Testing (and using) sets

- Alternatively, use the "dot" syntax:
  `ULSet.`*<member>*
- Note: Module names must be capitalized in OCaml
- Useful when two modules define the same operations

```
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
```

10: Does this code typecheck?

0

yes

0%

no

0%

```
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

10: Does this code typecheck?

0

yes

0%

no

0%

```
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

10: Does this code typecheck?

0

yes

0%

no

0%

```
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 (though performance may be different)

```
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 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

Equality of Sets

- Note that the interface for our abstract sets includes:

```
val equals : 'a set -> 'a set -> bool
```

  – This function defines what it means for two sets to be "equal".

- Why can't we just use OCaml's built-in `=` to compare?
  – This generic, built-in equality operation = compares the *structure* of its two inputs to see whether they are the same.
  – BUT(!) two values with *different* structure may represent the *same* collection of elements.
- In `ULSet:`

`3::0::1::[]` `0::1::3::[]`

concrete representation | concrete representation

abstract view | abstract view

These two values *are not equal* as lists.

These two values *are equal* as sets.

When defining an abstract type, you may need to define a different notion of equality

- The built-in "structural equality" written as = may not be appropriate
- Be sure to use the '`equals`' function when comparing, e.g., sets
- (Other generic operations, like < and > may also be affected.)

What Should You Test?

- Interface: defines operations on the type
- Properties: define how the operations interact
  - 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

Test the properties!

A *property* is a general statement about the behavior of the interface: For *any* set `s` and *any* element `x`:

```
member x (add x s) = true
```

A (good) test case checks a specific instance of the property:

```
let s1 = add 3 empty
let test () : bool = (member 3 s1)
;; run_test "ULSet.member 3 s1" test
```

Property-based Testing

1. Translate informal requirements into general statements about the interface.

> Example: "Order doesn't matter" becomes
> For *any* set `s` and *any* elements `x` and `y`,
> `add x (add y s)` equals `add y (add x s)`

2. Write tests for the "interesting" instances of the general statement.

> Example. "interesting" choices:
> `s` = `empty`, `s` = nonempty,
> `x = y`, `x <> y`
> one or both of `x`, `y` already in `s`

Notes:

- one can't (usually) exhaustively test all possibilities (too many!) so instead, cover the "interesting" possibilities
- be careful with equality! `ULSet.equals` is *not* the same as =