Programming Languages and Techniques (CIS1200)

Lecture 11

Abstract types: Finite Maps

Chapter 10

Announcements (1)

- Homework 3 is due tomorrow at 11.59pm
  - Practice with BSTs, generic functions, first-class functions, and abstract types
- Reading: Chapters 8, 9, and 10 of the lecture notes

Announcements (2)

- Midterm 1: Friday, September 27th
  - Coverage: up to Wednesday, Sep 25th (Chapters 1-10)
  - During lecture
    Last names: A – Z Meyerson Hall B1
  - 60 minutes; closed book, closed notes
  - Review Material
    - old exams on the web site ("schedule" tab)
  - **Review Session**
    - **Wednesday, September 25, 7:00-9:00pm, Towne 100** (will be recorded)
    - Review Videos available on canvas

Review: Abstract types (e.g., set)

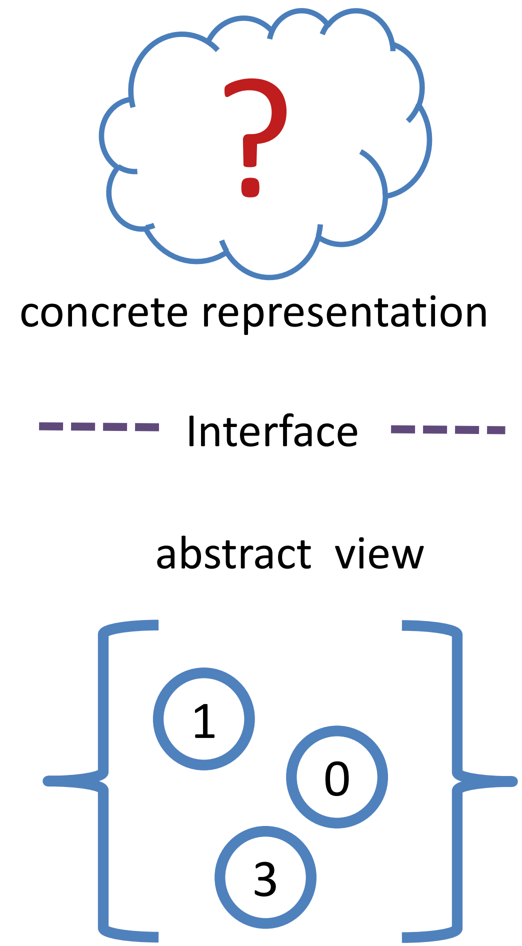
- An abstract type is defined by its *interface* and its *properties*, not its representation.

- Interface: defines operations on the type
  - 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 (possibly with invariants) that satisfies the interface and properties can be a set

- ***Clients of an implementation can only access what is explicitly mentioned in the abstract type's interface***



Review: Abstract vs. Concrete ULSet

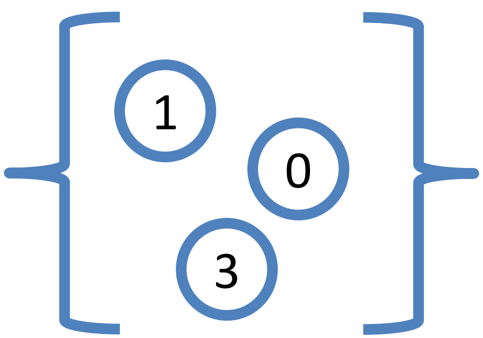
```
s  =   0::3::1::[]
```

```
module ULSet : SET = struct
  type 'a set = 'a list
  let empty : 'a set = []
  let add (x:'a)  (s:'a set) :'a set =
    (* can treat s as a list *)
    x :: s
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 module *)
;; open ULSet

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

Client code doesn’t (can’t!) care about internal representation!

Review: Abstract vs. Concrete OLSet

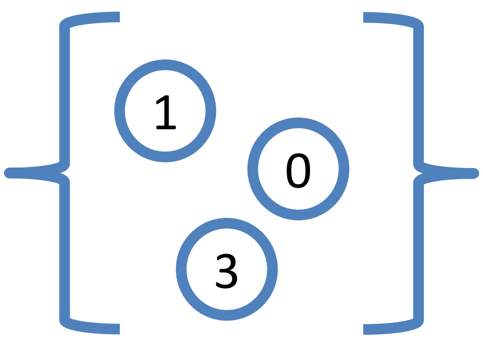
```
module OLSet : SET = struct
  type 'a set = 'a list
  let empty : 'a set = []
  let add (x:'a)  (s:'a set) :'a set =
    (* can treat s as a list, but
       must find right place for x *)
   …
end
```

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

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 OLSet module *)
;; open OLSet

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

Client code doesn't change!

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

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 = 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)

```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 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.

Equality of Sets

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

- The built-in "structural equality" (written =) 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.)

Equality of Sets

- The SET interface includes

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

This function should return true when both sets contain same elements

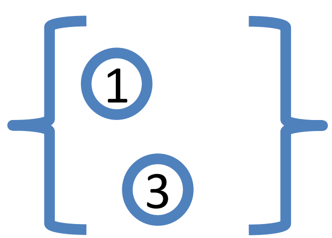
- Can we use OCaml's built-in `=` to compare sets?
  - This generic, built-in equality operation = compares the *structure* of its two inputs to see whether they are the same
- With unordered lists, `NO!`

These two values *are not* = as lists

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

concrete representation

abstract view

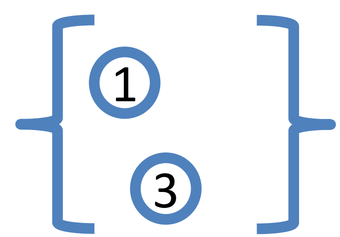


`add 3 (add 1 (add 1 empty))`

`1::3::[]`

concrete representation

abstract view



`add 1 (add 3 empty)`

These two values *are* `equal` as sets

Equality of Sets

- The SET interface includes

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

This function should return true when both sets contain same elements

- Can we use OCaml's built-in `=` to compare sets?
  - This generic, built-in equality operation = compares the *structure* of its two inputs to see whether they are the same

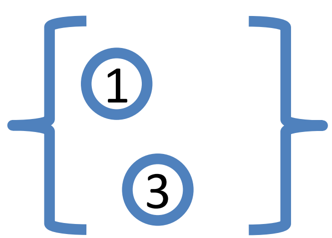
- With strictly ordered lists, YES!

These two values *are* = as lists

`1::3::[]`

concrete representation

abstract view

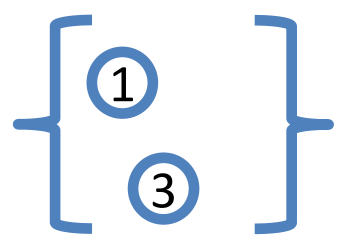


`add 3 (add 1 (add 1 empty))`

`1::3::[]`

concrete representation

abstract view



`add 1 (add 3 empty)`

These two values *are* `equal` as sets

Abstract types: BIG IDEA

Hide the *concrete representation* of a type behind an *abstract interface* to preserve **representation invariants**

- Example representation invariants
  - Sets implemented as lists, which must be strictly ordered (no duplicates)
  - Sets implemented as binary tree, satisfying the BST invariant

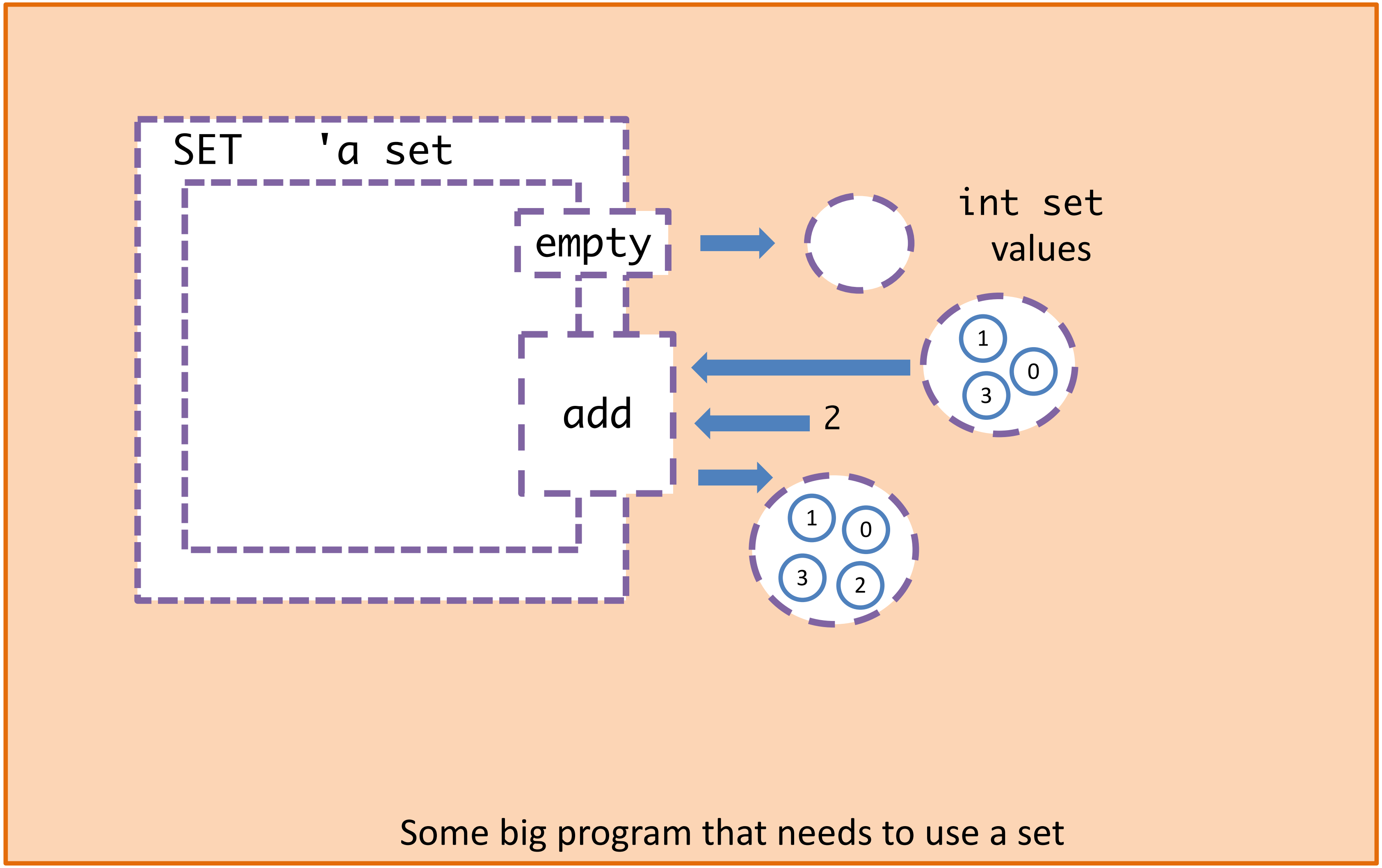
- If the set type is abstract, and *all* operations preserve invariants, then invariants **must** hold for *all* sets in the program!
  - Example: if all sets implemented as lists are strictly ordered, then the `=` operation implements set equality
  - Example: if all sets implemented as trees satisfy the BST invariant, then the lookup function can *assume* that its input is a BST

Abstract types: **BIG IDEA**

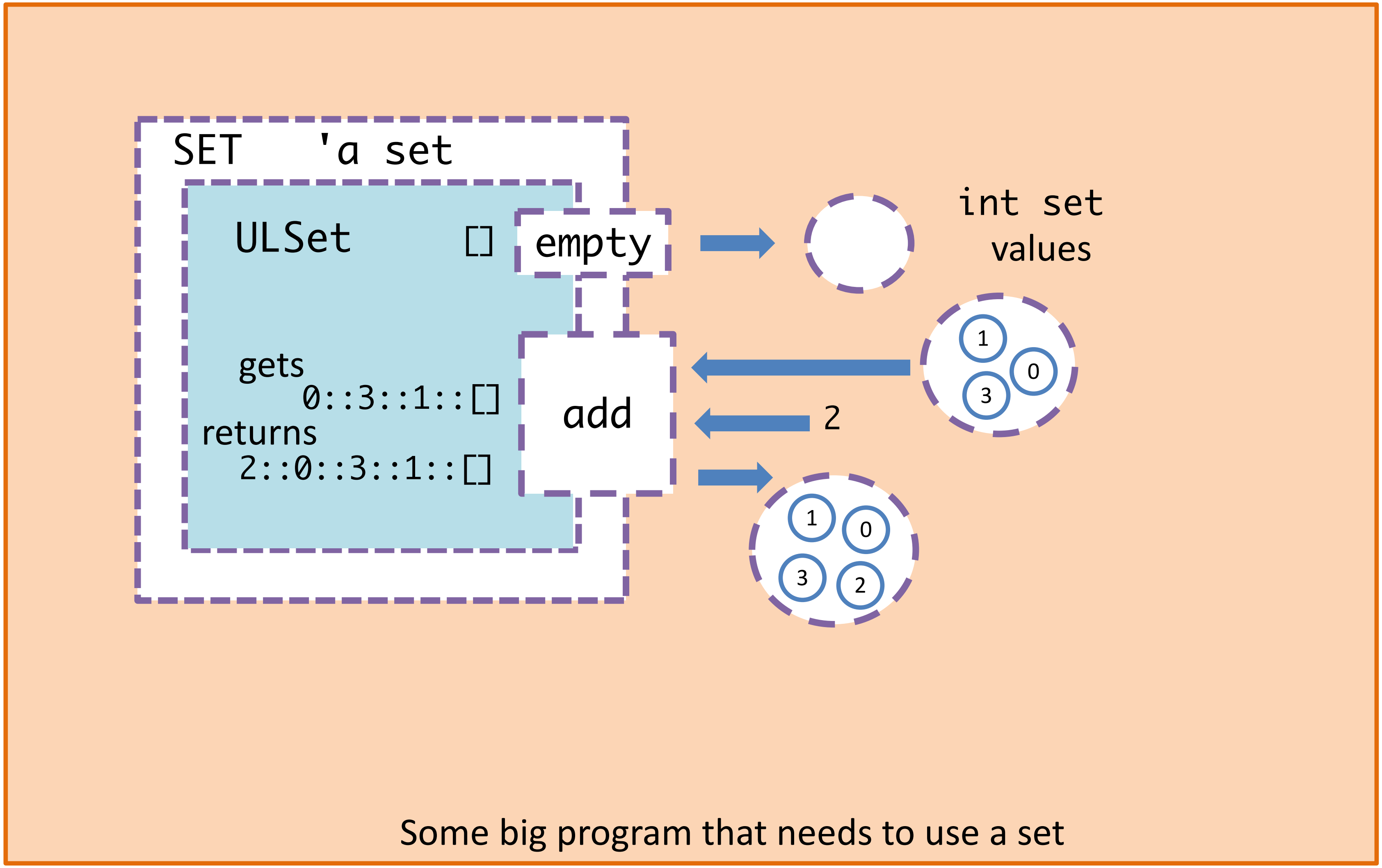
Hide the *concrete representation* of a type behind an *abstract interface* to preserve **representation invariants**

- An abstract 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 (empty, add, etc.)
  - Type checking ensures that clients cannot depend on whether the sets are implemented as trees or lists
- Benefits
  - **Safety**: The other parts of the program can't violate invariants, which would cause bugs
  - **Modularity**: It is possible to change the implementation without changing the rest of the program

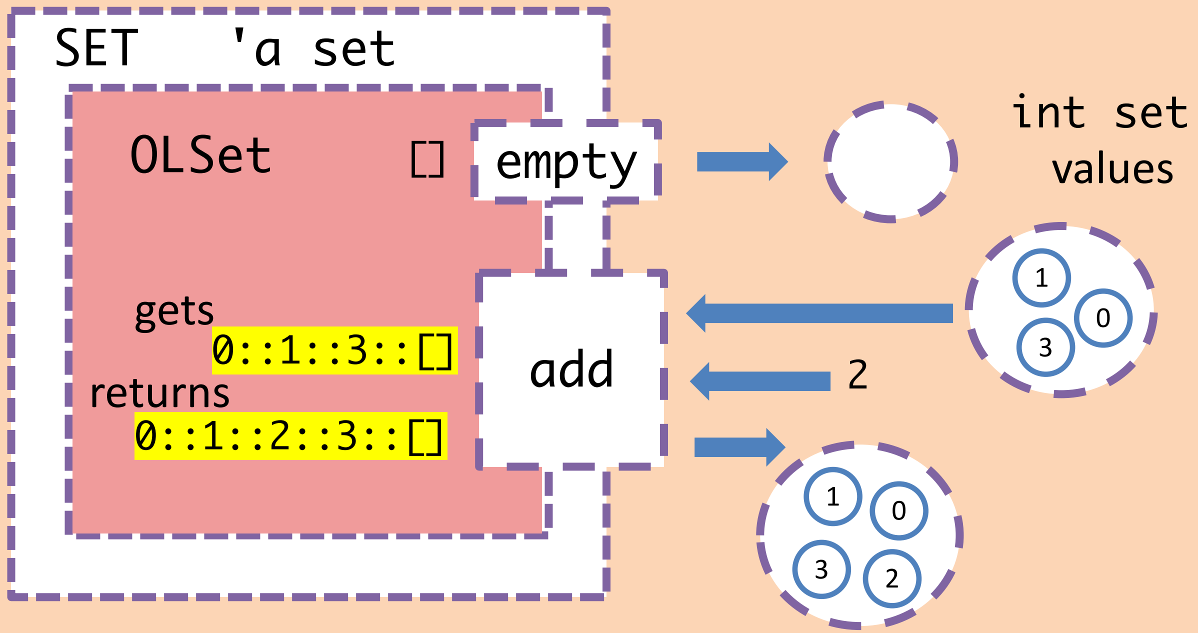
Encapsulation and Modularity



Implementation



Implementation



Abstraction Boundary – "preserves the invariants"

- inputs to the SET module satisfy the representation invariants
- as long as the created outputs do

Some big program that needs to use a set

Property-Based Testing

Testing Styles

- “From the inside”…
  - If we know the concrete representation of our data, we can test the effect of each operation on that representation
  - Useful for checking that invariants are maintained

- “From the outside”…
  - If the concrete representation is hidden, this doesn’t work!
  - We need a different way to think about testing

What Should We Test?

- Interface: Names and types of operations on the abstract type
- Properties: How the operations behave and interact
  - "Elements that were added can be found by `lookup`"
  - "Adding an element a second time doesn't change the elements of a set
  - "Adding elements in a different order doesn't change the outcome of later operations"

Test the properties!

A *property* is a general statement about the behavior of functions in the interface. E.g.,

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 test () : bool = (member 3 (add 3 empty))
;; run_test "member 3 (add 3 empty)" 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` is `empty` *vs.* `s` is nonempty
   > - `x = y` *vs.* `x <> y`
   > - `x` and/or `y` already in `s`
   >   *vs.* `x` and `y` different from what's in `s`

Notes:

- Not usually possible to exhaustively test all possibilities (too many!):
  so just try to cover the "interesting" choices
- Be careful with equality! `ULSet.equals` is *not* the same as =.

11: How comfortable are you with abstract types, interfaces, and implementations?

Totally Lost	0%
Still working on understanding	0%
I think I mostly get it	0%
Pretty comfortable - it makes sense	0%
Ready to design my own ADTs	0%

Finite Maps

A case study on abstract interfaces

and concrete implementations

Motivating Scenario

- Suppose you were writing a course-management system 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?

Key/Value store

Key	Value
“stephanie”	15
“mitch”	05
“ezaan”	10
“likat”	15
…	…

- Each key is associated with a value.
  - No two keys are identical
  - Values can be repeated
- Given the key “stephanie”, we want to find / lookup the value 15

Finite Maps

Design Process Step 1: Understand the problem

- A *finite map* (a.k.a. *dictionary*) is a collection of *entries* from distinct *keys* to *values*.
  - Operations to *add* a new entry, *test* for key membership, *get* the value bound to a particular key, *list* all entries stored in the map

- Example: we might use a finite map to look up the lab section of a CIS 1200 student

- Like sets, *finite maps* appear in many settings:
  - domain names to IP addresses
  - words to their definitions (a dictionary)
  - user names to passwords
  - …

Signature: Finite Map

Design Process Step 2: specify the interface

```
module type MAP = sig

  type ('k,'v) map

  val empty   : ('k,'v) map
  val add     : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
  val mem     : 'k -> ('k,'v) map -> bool
  val get     : 'k -> ('k,'v) map -> 'v
  val equals  : ('k,'v) map -> ('k,'v) map -> bool

end
```

The map type is generic in *two* ways: type of keys and type of values

Properties of Finite Maps

Design Process Step 3: write test cases

For any finite map m, key k, and value v:

1. `get k (add k v m) = v`
2. If `k1 <> k2` then
   `get k1 (add k2 v2 (add k1 v1 m)) = v1`
3. If `mem k m = true` then
   there is a `v` such that `get k m = v`
4. If `mem k m = false` then
   `get k m = v` fails
5. `mem k (add k v m) = true`

(among others...)

Tests for Finite Map abstract type

Design Process Step 3: write test cases

```
;; open Assert

(* Specifying the properties of the MAP abstract type via test cases. *)

(* A simple map with one element. *)
let m1 : (int,string) map = add 1 "uno" empty

(* access value for key in the map *)
;; run_test "find 1 m1" (fun () -> (get 1 m1) = "uno")

(* find for value that does not exist in the map? *)
;; run_failing_test "find 2 m1" (fun () -> (get 2 m1) = "dos" )

let m2 : (int, string) map = add 1 "un" m1

(* find after redefining value, should be new value *)
;; run_test "find 1 m2" (fun () -> (get 1 m2) = "un")

(* test membership *)
;; run_test "mem test" (fun () ->
        mem 1 (add 2 "dos" (add 1 "uno" empty)))
```

Using an anonymous function avoids making up a (redundant) function name for the test

Finite Map Demo

Implementing the module

finiteMap.ml

Implementation: Ordered Lists

Design Process Step 4: implement it!

```
module Assoc : MAP = struct
  (* Represent a finite map as a list of pairs.          *)
  (* Representation invariant:                           *)
  (*   - no duplicate keys (helps get, remove)           *)
  (*   - keys are sorted (helps equals, get)             *)

type ('k,'v) map = ('k * 'v) list

  let empty : ('k,'v) map = []

  let rec mem (key:'k) (m : ('k,'v) map) : bool =
    begin match m with
    | [] -> false
    | (k,v)::rest ->
      (key >= k) &&
         ((key = k) || (mem key rest))
    end

;; run_test "mem test" (fun () -> mem "b" [("a",3); ("b",4)])
```

Implementation: Ordered Lists

```
let rec get (key:'k) (m : ('k,'v) map) : 'v =
  begin match m with
  | [] -> failwith "key not found"
  | (k,v)::rest ->
    if key < k then failwith "key not found"
    else if key = k then v
    else get key rest
  end

let rec remove (key:'k) (m : ('k,'v) map) : ('k,'v) map =
  begin match m with
  | [] -> []
  | (k,v)::rest ->
    if key < k then m
    else if key = k then rest
    else (k,v)::remove key rest
  end
```

Summary: Abstract Types

- Different programming languages support different ways of defining 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: the interface can *omit* information
    - type definitions
    - names of auxiliary functions
  - Clients *cannot* mention values or types not named in the interface

Typechecking

How does OCaml* typecheck your code?

*Historical aside: the algorithm we are about to see is known as the Damas-Hindley-Milner type inference algorithm. Turing Award winner Robin Milner was, among other things, the inventor of "ML" (for "meta language"), from which OCaml gets its "ml".

OCaml Typechecking Errors

```
type ('k,'v) map = ('k * 'v) list

(* A finite map that contains no entries. *)
let empty () = []

let rec me
  begin ma
  | [] ->
  | (k,v):
    if key
    (key = k) || (mem key rest)
  end

;; run_test "mem test" (fun () ->
  mem "b" [("a",3); ("b",4)]
)

let rec get (key:'k) (m : ('k,'v) map) : 'v =
  begin match m with
  | [] -> failwith "not found"
```

```
Signature mismatch:
       ...
       Values do not match:
         val empty : unit -> 'a list
       is not included in
         val empty : ('k, 'v) map
       File "finiteMap.ml", line 13, characters 2-27: Expected
declaration
       File "finiteMap.ml", line 60, characters 6-11: Actual declaration
```

Typechecking

How do we determine the type of an expression?

1. Recursively determine the types of *all* sub-expressions
   - Constants have "obvious" types

     ```
     3 : int     "foo" : string     true : bool
     ```
   - Identifiers may have type annotations
     - `let` and function arguments
     - Module signatures/interfaces

2. Expressions that *construct* structured values have compound types built from the types of sub-expressions

   ```
   (3, "foo")                       : int * string
   (fun (x:int) -> x + 1)           : int -> int
   Node(Empty, (3, "foo"), Empty)   : (int * string) tree
   ```

Typechecking Functions

To typecheck a function:

```
fun (x:int) -> x + x
```

??

Typechecking Functions

To typecheck a function:

```
fun (x:int) -> x + x
```

$$\underbrace{\hphantom{\texttt{fun (x:int) -> x + x}}}_{T_{arg} \texttt{ -> } T_{ans}}$$

Make up "new names" for the input (argument) and output (answer) types.

Typechecking Functions

To typecheck a function:

$$\underbrace{\texttt{fun (x:int) -> x + x}}_{T_{arg}\ \texttt{->}\ T_{ans}}$$

Take the argument type from the type annotation (if any*): T_{arg} = `int`

*If there is no annotation, just use the "fresh" name…

Typechecking Functions

To typecheck a function:

$$\underbrace{\texttt{fun (x:int) -> }\underbrace{\underbrace{\texttt{x}}_{\texttt{int}}\texttt{ + }\underbrace{\texttt{2}}_{}}_{}}_{\texttt{int -> }T_{ans}}$$

Recursive typecheck the body of the function in a "typing context" where the argument has the input type:
`(x : int)`

Typechecking Functions

To typecheck a function:

$$\underbrace{\texttt{fun (x:int) -> }\underbrace{\underbrace{\texttt{x}}_{\texttt{int}}\texttt{ + }\underbrace{\texttt{2}}_{\texttt{int}}}_{}}_{\texttt{int -> }T_{ans}}$$

Literals like 2 have unique types:

`(2 : int)`

Typechecking Functions

To typecheck a function:

Built-in operations like (+) also have types:

```
(+) : int -> int -> int
```

$$\underbrace{\texttt{fun (x:int) -> }\underbrace{\underbrace{\texttt{x}}_{\texttt{int}}\texttt{ + }\underbrace{\texttt{2}}_{\texttt{int}}}_{\texttt{int}}}_{\texttt{int -> }T_{ans}}$$

Function application has the result type, assuming the input types are correct.

Typechecking Functions

To typecheck a function:

$$\underbrace{\texttt{fun (x:int) -> }\underbrace{\underbrace{\texttt{x}}_{\texttt{int}}\texttt{ + }\underbrace{\texttt{2}}_{\texttt{int}}}_{\texttt{int}}}_{\texttt{int -> }T_{ans}}$$

The "answer" type is the type of the body.

T_{ans} = `int`

Typechecking Functions

To typecheck a function:

$$\underbrace{\texttt{fun (x:int) -> }\underbrace{\underbrace{\texttt{x}}_{\texttt{int}}\texttt{ + }\underbrace{\texttt{2}}_{\texttt{int}}}_{\texttt{int}}}_{\texttt{int -> int}}$$

Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:
   - Given a function `f` : T_{arg} `->` T_{ans}
   - and an argument `e` : T_{arg} of the input type
   - the application `(f e)` : T_{ans} has the answer type

```
((fun (x:int) (y:bool) -> y)  3)  : ??
```

Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:
   - Given a function `f` : T_{arg} `->` T_{ans}
   - and an argument `e` : T_{arg} of the input type
   - the application `(f e)` : T_{ans} has the answer type

```
((fun (x:int) (y:bool) -> y)  3)  : ??
```

??

Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:
   - Given a function `f` : T_{arg} `->` T_{ans}
   - and an argument `e` : T_{arg} of the input type
   - the application `(f e)` : T_{ans} has the answer type

```
((fun (x:int) (y:bool) -> y)  3)  : ??
```

```
int -> bool -> ??
```

```
??
```

Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:
   - Given a function `f` : T_{arg} `->` T_{ans}
   - and an argument `e` : T_{arg} of the input type
   - the application `(f e)` : T_{ans} has the answer type

$$\underbrace{\texttt{(}\underbrace{\texttt{(fun (x:int) (y:bool) -> }\underbrace{\texttt{y)}}_{\texttt{bool}}}_{\texttt{int -> bool -> ??}}\texttt{ 3)}}_{\texttt{??}}\texttt{ : ??}$$

Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:
   - Given a function $\texttt{f} \;:\; T_{arg} \texttt{ -> } T_{ans}$
   - and an argument $\texttt{e} \;:\; T_{arg}$ of the input type
   - the application $\texttt{(f e)} \;:\; T_{ans}$ has the answer type

$$\underbrace{(\underbrace{(\texttt{fun (x:int) (y:bool) -> }\underbrace{\texttt{y)}}_{\texttt{bool}}}_{\texttt{int -> bool -> bool}}\ \texttt{3)}}_{\texttt{??}} \;:\; \texttt{??}$$

Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:
   - Given a function `f` : T_{arg} `->` T_{ans}
   - and an argument `e` : T_{arg} of the input type
   - the application `(f e)` : T_{ans} has the answer type

$$\underbrace{\texttt{(}\underbrace{\texttt{(fun (x:int) (y:bool) -> }\underbrace{\texttt{y}}_{\texttt{bool}}\texttt{)}}_{\texttt{int -> bool -> bool}}\;\underbrace{\texttt{3}}_{\texttt{int}}\texttt{)}}_{\texttt{??}} \;\texttt{: ??}$$

Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:
   - Given a function `f` : T_{arg} `->` T_{ans}
   - and an argument `e` : T_{arg} of the input type
   - the application `(f e)` : T_{ans} has the answer type

```
((fun (x:int) (y:bool) -> y)  3)  : ??
```

`bool`

`int -> bool -> bool`   `int`

`bool -> bool`

Here:
T_1 = `int`
T_2 = `bool -> bool`

Typechecking III

- What about generics? i.e., what if `f:'a ->'a`?
- For generic types we *unify*
  - Given a function `f` : T_1 -> T_2
  - and an argument `e` : U_1 of the input type

  Can *"match up"* T_1 and U_1 to obtain information about type parameters in T_1 and U_1 based on their usage
- *Unification:*
  - try to match up corresponding parts of the type

    `(int list) tree` ⇔ `'a tree`
  - Obtain an *instantiation*: e.g. `'a = int list`
  - *Propagate* that information to all occurrences of `'a`
  - If not possible, unification fails, meaning a type checking error

    `bool tree` ⇔ `int tree`

    ERROR! `bool` ≠ `int`

Example Typechecking Problem

```
empty   : ('k, 'v) map
add     : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```

??

Example Typechecking Problem

```
empty   : ('k, 'v) map
add     : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```

$\underbrace{\hspace{12em}}_{\texttt{'v -> ??}}$

Example Typechecking Problem

```
empty   : ('k, 'v) map
add     : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```

```
'v -> ??
```

Example Typechecking Problem

```
empty   : ('k, 'v) map
add     : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```

```
int
```

```
'v -> ??
```

Example Typechecking Problem

```
empty   : ('k, 'v) map
add     : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```

```
int 'v
```

```
'v -> ??
```

Example Typechecking Problem

```
empty   : ('k, 'v) map
add     : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```

int 'v ('k, 'v) map

‘v -> ??

Example Typechecking Problem

```
empty   : ('k, 'v) map
add     : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```

```
int 'v ('k, 'v) map
```

```
'v -> ??
```

Example Typechecking Problem

```
empty   : ('k, 'v) map
add     : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```

```
int 'v ('k, 'v) map
```

```
??
```

```
'v -> ??
```

Example Typechecking Problem

```
empty   : ('k, 'v) map
add     : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```

`int 'v ('k, 'v) map`

T_2

Application:

T_1 = 'k

T_2 = 'v -> ('k,'v) map -> ('k,'v) map

Instantiate: 'k = int

`'v -> ??`

Example Typechecking Problem

```
empty   : ('k, 'v) map
add     : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```

`int 'v (int, 'v) map`

T_2

T'_2

Another Application:

T'_1 = 'v

T'_2 = (int, 'v) map -> (int, 'v) map

Instantiate: 'v = 'v

`'v -> ??`

Example Typechecking Problem

```
empty   : ('k, 'v) map
add     : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```

`int 'v (int, 'v) map`

T_2

T'_2

T''_2= (int, 'v) map

A third Application:
T''_1 = (int, 'v) map
T''_2 = (int, 'v) map

Argument and argument type already agree

`'v -> ??`

Example Typechecking Problem

```
empty   : ('k, 'v) map
add     : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```

int 'v (int, 'v) map

T_2

T'_2

U_1 -> U_2

T''_2= (int,'v) map

'v -> ??

Example Typechecking Problem

```
empty   : ('k, 'v) map
add     : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```

int 'v (int, 'v) map

T_2

T'_2

U_1 -> U_2

T''_2= (int,’v) map

??

'v -> ??

Another Application:
U_1 = ('k,'v) map
U_2 = ('k * 'v) list

Unify U_1 with T''_2
 ('k,'v) map ~~ (int,'v) map

Instantiate 'k = int

Example Typechecking Problem

```
empty   : ('k, 'v) map
add     : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```

int 'v (int, 'v) map

T_2

T'_2

U_1 -> U_2

T''_2= (int, 'v) map

U_2= (int * 'v) list

'v -> ??

Another Application:

U_1 = (int, 'v) map

U_2 = (int * 'v) list

Example Typechecking Problem

```
empty   : ('k, 'v) map
add     : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```

int 'v (int, 'v) map

T_2

T'_2

U_1 -> U_2

T''_2= (int, 'v) map

U_2= (int * 'v) list

```
'v -> (int * 'v) list
```

Another Application:

U_1 = (int, 'v) map

U_2 = (int * 'v) list

Ill-typed Expressions?

- An expression is ill-typed if, during this type checking process, inconsistent constraints are encountered:

```
empty   : ('k, 'v) map
add     : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries : ('k, 'v) map -> ('k * 'v) list
```

```
add 3 true (add “foo” false empty)
```

Error: found `int` but expected `string`

12: What is the type of this expression?

✓ 0

```
let e : ______ =
  transform (fun x y -> x + y)
```

int list -> int list

0%

int list -> int list -> int list

0%

int list -> (int -> int) list

0%

None (it doesn’t typecheck)

0%

What is the type of this expression?

```
let e : ______  =
  transform (fun x y -> x + y)
```

1. int list -> int list

2. int list -> int list -> int list

3. int list -> (int -> int) list

4. None (it doesn’t typecheck)

Answer: 3