Modules and Abstract Data Types

COS 326

Theresa Lim

Princeton University

Engineering for Change

Given that we know the software will change, how can we write the code so that doing the changes will be easier?

Engineering for Change

Given that we know the software will change, how can we write the code so that doing the changes will be easier?

The primary trick: use ***data and algorithm abstraction***.

- *Don't* code in terms of *concrete representations* that the language provides.
- *Do* code with *high-level abstractions* in mind that fit the problem domain.

Abstract Data Types

Barbara Liskov
Assistant Professor, MIT
1973

Invented CLU language
that enforced data abstraction

Barbara Liskov
Professor, MIT
Turing Award 2008

"For contributions to practical and theoretical foundations of programming language and system design, especially related to data abstraction, fault tolerance, and distributed computing."

Building Abstract Types in OCaml

OCaml has mechanisms for building new abstract data types:

- ***module type*** *(or "signature")*: an interface.
  - specifies the abstract type(s) without specifying their implementation
  - specifies the set of operations on the abstract types
- ***module*** *(or "structure")*: an implementation.
  - a collection of type and value definitions
- ***parameterized module*** *(or "functor")*:
  - stay tuned :))

Simple Modules

Recall assigment #2:

query.ml

```
type movie = { ... }

let sort_by_studio  = ...
let sort_by_year = ...
```

main.ml

```
open Io
open Query

let main () = ... sort_by_studio ...

let _ = main ()
```

Simple Modules

Recall assigment #2:

query.ml

```
type movie = { ... }

let sort_by_studio  = ...
let sort_by_year = ...
```

main.ml

```
open Io
open Query

let main () = ... sort_by_studio ...

let _ = main ()
```

Each .ml file actually defines an ML module.

Convention: the file foo.ml or Foo.ml defines the module named Foo.

Simple Modules

Recall assigment #2:

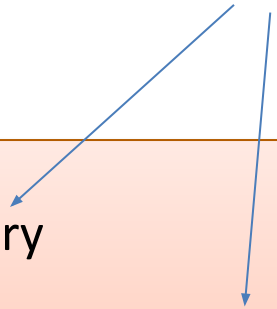
open gives
direct access to
module components

query.ml

```
type movie = { ... }

let sort_by_studio  = ...
let sort_by_year = ...
```

main.ml

```
open Io
open Query

let main () = ... sort_by_studio ...

let _ = main ()
```



Simple Modules

Recall assigment #2:

query.ml

```
type movie = { ... }

let sort_by_studio  = ...
let sort_by_year = ...
```

main.ml

```
open Io
open Query

let main () =
  ... Query.sort_by_studio ...
```

redacted

Can refer to module components using dot notation

Simple Modules

query.ml

```
type movie = { ... }

let sort_by_studio  = ...
let sort_by_year = ...
```

main.ml

```
open Io
open Query

let main () =
  ... Query.sort_by_studio ...
```

query.mli

```
type movie

val sort_by_studio : movie list -> movie list
val sort_by_year : movie list -> movie list
```

You can add interface files (.mli)
(also called *signatures* in ML)

These interfaces can hide
module components
or render types abstract.

Simple Modules

query.ml

```
type movie = { ... }

let sort_by_studio  = ...
let sort_by_year = ...
```

main.ml

```
open Io
open Query

let main () =
  ... Query.sort_by_studio ...
```

query.mli

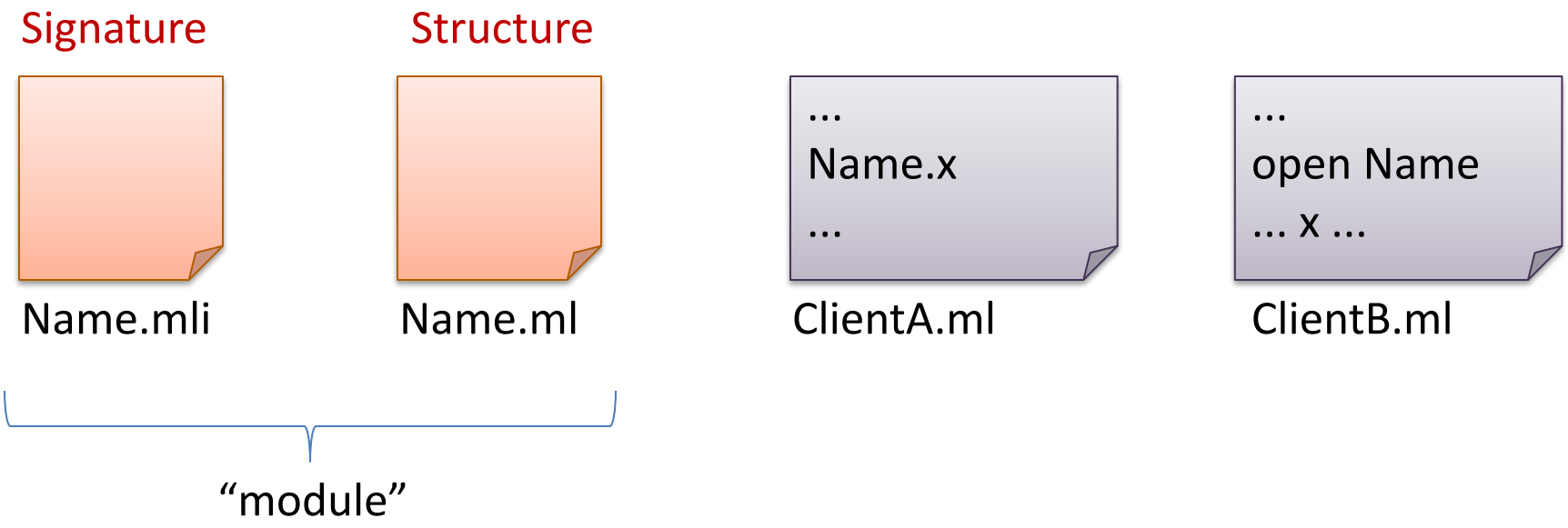
```
type movie

val sort_by_studio : movie list -> movie list
val sort_by_year : movie list -> movie list
```

If you have no signature file, then the default signature is used: all components are fully visible to clients.

Simple Modules

Simple summary:

- file Name.ml is a *structure* implementing a module named Name
- file Name.mli is a *signature* for the module named Name
  - if there is no file Name.mli, OCaml infers the default signature



Signature Definitions Inside Files

```
module type INT_STACK =
  sig
    type stack
    val empty : unit -> stack
    val push  : int -> stack -> stack
    val is_empty : stack -> bool
    val pop : stack -> stack
    val top : stack -> int option
  end
```

Example Structure Inside a File

```
module ListIntStack : INT_STACK =
  struct
    type stack = int list
    let empty () : stack = []
    let push (i:int) (s:stack) : stack = i::s
    let is_empty (s:stack) =
      match s with
        | [] -> true
        | _::_ -> false
    let pop (s:stack) : stack =
      match s with
        | [] -> []
        | _::t -> t
    let top (s:stack) : int option =
      match s with
        | [] -> None
        | h::_ -> Some h
  end
```

Example Structure Inside a File

```
module ListIntStack : INT_STACK =
  struct
    type stack = int list
    let empty () : stack = []
    let push (i:int) (s:stack) = i::
    let is_empty (s:stack) =
      match s with
        | [] -> true
        | _::_ -> false
    let pop (s:stack) : stack =
      match s with
        | [] -> []
        | _::t -> t
    let top (s:stack) : int option =
      match s with
        | [] -> None
        | h::_ -> Some h
  end
```

Inside the module, we know the ***concrete type*** used to implement the abstract type.

Example Structure Inside a File

```
module ListIntStack : INT_STACK =
  struct
    type stack = int list
    let empty () : stack = []
    let push (i:int) (s:stack) = i
    let is_empty (s:stack) =
      match s with
        | [] -> true
        | _::_ -> false
    let pop (s:stack) : stack =
      match s with
        | [] -> []
        | _::t -> t
    let top (s:stack) : int option =
      match s with
        | [] -> None
        | h::_ -> Some h
  end
```

But by giving the module the INT_STACK interface, which does not reveal how stacks are being represented, we prevent code outside the module from knowing stacks are lists.

An Example Client

```
module ListIntStack : INT_STACK =
  struct
    …
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let x = ListIntStack.top s2
```

An Example Client

```
module ListIntStack : INT_STACK =
  struct
    …
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let x = ListIntStack.top s2
```

```
s0 : ListIntStack.stack
s1 : ListIntStack.stack
s2 : ListIntStack.stack
```

An Example Client

```
module ListIntStack : INT_STACK =
  struct
    …
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let x = ListIntStack.top s2
x : option int = Some 4
```

An Example Client

```
module ListIntStack : INT_STACK =
  struct
    …
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let x = ListIntStack.top s2
x : option int = Some 4
let x = ListIntStack.top (ListIntStack.pop s2)
x : option int = Some 3
```

An Example Client

```
module ListIntStack : INT_STACK =
  struct
    …
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let x = ListIntStack.top s2
x : option int = Some 4
let x = ListIntStack.top (ListIntStack.pop s2)
x : option int = Some 3
open ListIntStack
```

An Example Client

```
module ListIntStack : INT_STACK =
  struct
    …
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let x = ListIntStack.top s2
x : option int = Some 4
let x = ListIntStack.top (ListIntStack.pop s2)
x : option int = Some 3
open ListIntStack
let x = top (pop (pop s2))
x : option int = None
```

An Example Client

```
module type INT_STACK =
  sig
    type stack
    val push  : int -> stack -> stack
    …
  end
module ListIntStack : INT_STACK

let s2 = ListIntStack.push 4 s1
…
List.rev s2
Error: This expression has type stack but an
expression was expected of type 'a list.
```

Notice that the client is not allowed to know that the stack is a list.

The Client without the Signature

```
module ListIntStack (* : INT_STACK *) =
  struct
    …
  end

let s = ListIntStack.empty()
let s1 = ListIntStack.push 3 s
let s2 = ListIntStack.push 4 s1

…
let x = List.rev s2
x : int list = [3; 4]
```

If we don't seal the module with a signature, the client **can know** that stacks are lists.

MODULE EVALUATION

Evaluating the contents of a module

A structure is a series of declarations

- How does one evaluate a type declaration? We'll ignore it (because it doesn't do anything at run time).
- How does one evaluate a let declaration?

How does one evaluate an entire structure?

- evaluate each declaration in order from first to last

Evaluating the contents of a module

main.ml

```
let x = 326

let main () =
   Printf.printf “Hello COS %d\n” x

let foo =
   Printf.printf “Byeee!\n”

let _ =
    main ()
```

Evaluating the contents of a module

main.ml

```
let x = 326

let main () =
   Printf.printf “Hello COS %d\n” x

let foo =
   Printf.printf “Byeee!\n”

let _ =
    main ()
```

<u>Step 1:</u>
evaluate the 1st declaration

but the RHS (326)
is already a value so there’s nothing to do except remember that x is bound to the integer 326

Evaluating the contents of a module

main.ml

```
let x = 326

let main () =
   Printf.printf “Hello COS %d\n” x

let foo =
   Printf.printf “Byeee!\n”

let _ =
    main ()
```

<u>Step 2:</u>
evaluate the 2nd declaration
this is slightly trickier:

let main () = ...

really declares a function.
It’s equivalent to:

let main = fun () -> ...

“fun () -> ...” is already
a value, like 326.
So there’s nothing to do again.

Evaluating the contents of a module

main.ml

```
let x = 326

let main () =
   Printf.printf “Hello COS %d\n” x

let foo =
   Printf.printf “Byeee!\n”

let _ =
    main ()
```

<u>Step 3:</u>
evaluate the 3rd declaration

let foo = ...

evaluation of this expression has an effect – it prints out “Byeee!\n” to the terminal.

the resulting value is () which is bound to foo

Evaluating the contents of a module

main.ml

```
let x = 326

let main () =
   Printf.printf “Hello COS %d\n” x

let foo =
   Printf.printf “Byeee!\n”

let _ =
    main ()
```

<u>Step 4:</u>
evaluate the 4th declaration

let _ = ...

evaluation main ()
causes another effect.

“Hello ...” is printed

the resulting value is () again.
the “_” indicates we don’t
care to bind () to any variable

FUNCTORS

We're all familiar with functions…

```
let foo (x: 'a) : 'b = …
```

This **function** takes in some value of type 'a and then transforms it into a new value of type 'b

We can do the same for modules!

```
let foo (x: 'a) : 'b = …
```

This **function** takes in some value of type 'a and then transforms it into a new value of type 'b

```
module FOO (BAR: MType) : MType' =
  sig
    ...
  end
```

This **functor** takes in some module of type MType and then transforms it into a new module of type MType'

An Example

```
module type SET =
  sig
    type elt
    type set
    val empty : set
    val is_empty : set -> bool
    val insert : elt -> set -> set
    val singleton : elt -> set
    ...
  end
```

Here's a Set Functor:

```
module ListSet (Elt : sig type t end)
            :  (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  let empty : set = []
  let is_empty (s:set) =
    match xs with
    | [] -> true
    | _::_ -> false
  let singleton (x:elt) : set = [x]
...
end

module IntListSet = ListSet(struct type t = int end)
module StringListSet = ListSet(struct type t = string end)
```

Here's a Set Functor:

```
module ListSet (Elt : sig type t end)
               : (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  let empty : set = []
  let is_empty (s:set)
    match xs with
    | [] -> true
    | _::_ -> false
  let singleton (x:elt) : set = [x]
...
end

module IntListSet = ListSet(struct type t = int end)
module StringListSet = ListSet(struct type t = string end)
```

ListSet is a **parameterized module** aka **functor** – given a module argument for Elt, it generates a new module.

Here's a Set Functor:

```
module ListSet (Elt : sig type t end)
               :  (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  let empty : set = []
  let is_empty (s:set) =
    match xs with
    | [] -> true
    | _::_ -> false
  let singleton (x:elt) : set = [x]
...
end

module IntListSet = ListSet(struct type t = int end)
module StringListSet = ListSet(struct type t = string end)
```

This is a very simple, ***anonymous*** signature (it just specifies there's some type t) for the argument to ListSet

Here's a Set Functor:

```
module ListSet (Elt : sig type t end)
               :   (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  let empty : set = []
  let is_empty (s:set) =
    match xs with
    | [] -> true
    | _::_ -> false
  let singleton (x:elt) : set = [x]
...
end

module IntListSet = ListSet(struct type t = int end)
module StringListSet = ListSet(struct type t = string end)
```

This is the signature of the resulting module – we have a set plus the knowledge that the Set's elt type is equal to Elt.t

Here's a Set Functor:

```
module ListSet (Elt : sig type t end)
               :   (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  let empty : set = []
  let is_empty (s:set) =
    match xs with
    | [] -> true
    | _::_ -> false
  let singleton (x:elt) : set = [x]
...
end

module IntListSet = ListSet(struct type t = int end)
module StringListSet = ListSet(struct type t = string end)
```

Side note: this is called a "sharing constraint" and is a kind of **dependent type**

Here's a Set Functor:

```
module ListSet (Elt : sig type t end)
               :  (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  let empty : set = []
  let is_empty (s:set) =
    match xs with
    | [] -> true
    | _::_ -> false
  let singleton (x:elt) : set = [x]
...
end

module IntListSet = ListSet(struct type t = int end)
module StringListSet = ListSet(struct type t = string end)
```

These are two SET modules that I created with the ListSet functor.

Here's a Set Functor:

```
module ListSet (Elt : sig type t end)
               :  (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  let empty : set = []
  let is_empty (s:set) =
    match xs with
    | [] -> true
    | _::_ -> false
  let singleton (x:elt) : set =
...
end

module IntListSet = ListSet(struct type t = int end)
module StringListSet = ListSet(struct type t = string end)
```

In this case, I'm passing in an anonymous module for Elt that defines t to be int.

Here's a Set Functor:

```
module ListSet (Elt : sig type t end)
               :   (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  let empty : set = []
  let is_empty (s:set) =
    match xs with
    | [] -> true
    | _::_ -> false
  let singleton (x:elt) : set = [x]
...
end

module IntListSet = ListSet(struct type t = int end)
module StringListSet = ListSet(struct type t = string end)
```

We know that IntListSet.elt = int.

SUMMARY

Modules yay! Functors yay!

Modules allow us to package code into abstract units

- A module type or *signature* specifies a module's behavior abstractly
- A module or *structure* is the implementation of a module type

Functors allow us to parametrize modules

- They act like "functions" over modules