Modules
and Abstract Data Types

COS 326
David Walker
Princeton University
The Reality of Development

We rarely know the *right* algorithms or the *right* data structures when we start a design project.

– When implementing a search engine, what data structures and algorithms should you use to build the index? To build the query evaluator?

Reality is that *we often have to go back and change our code*, once we’ve built a prototype.

– Often, we don’t even know what the *user wants* (requirements) until they see a prototype.

– Often, we don’t know where the *performance problems* are until we can run the software on realistic test cases.

– Sometimes we just want to change the design -- come up with *simpler* algorithms, architecture later in the design process.
Given that we know the software will change, how can we write the code so that doing the changes will be easier?
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*.
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.
- Implement the abstractions using a **well-defined interface**.
- Swap in **different implementations** for the abstractions.
- *Parallelize* the development process.
Goal: Implement a query engine.

Requirements: Need a scalable *dictionary* (a.k.a. index)
   – maps words to *set* of URLs for the pages on which words appear.
   – want the index so that we can efficiently satisfy queries
     • e.g., all links to pages that contain “Dave” and “Jill”.

Wrong way to think about this:
   – Aha! A *list* of pairs of a word and a *list* of URLs.
   – We can look up “Dave” and “Jill” in the *list* to get back a *list* of URLs.
Example

dtype query =
  Word of string |
  And of query * query |
  Or of query * query ;;

dtype index = (string * (url list)) list ;;

let rec eval(q:query)(h:index) : url list =
  match q with
  | Word x ->
    let (_,urls) = List.find (fun (w,urls) -> w = x) h in
    urls
  | And (q1,q2) ->
    merge_lists (eval q1 h) (eval q2 h)
  | Or (q1,q2) ->
    (eval q1 h) @ (eval q2 h)
Example

type query =
    Word of string
| And of query * query
| Or of query * query ;;

type index = (string * (url list)) list ;;

let rec eval (q:query) (h:index) : url list =
    match q with
    | Word x ->
        let (_,urls) = List.find (fun (w,urls) -> w = x) in urls
    | And (q1,q2) ->
        merge_lists (eval q1 h) (eval q2 h)
    | Or (q1,q2) ->
        (eval q1 h) @ (eval q2 h)

merge expects to be passed sorted lists.
Example

```ml
open List

type query =
  Word of string
| And of query * query
| Or of query * query ;;

type index = (string * (url list)) list ;;

let rec eval (q:query) (h:index) : url list =
  match q with
  | Word x ->
    let (_,urls) = List.find (fun (w,urls) -> w = x) in
    urls
  | And (q1,q2) ->
    merge_lists (eval q1 h) (eval q2 h)
  | Or (q1,q2) ->
    (eval q1 h) @ (eval q2 h)

merge expects to be passed sorted lists.
Oops!
```
```ocaml
(* Example *

```
type query =  
    Word of string  
  | And of query * query  
  | Or of query * query
```

```
type index = string (url list) hashtable ;;
```

```
let rec eval (q:query) (h:index) : url list =
  match q with
  | Word x ->
    let i = hash_string x in
    let l = Array.get h [i] in
    let urls = assoc_list_find l x in
    urls
  | And (q1,q2) -> ...
  | Or (q1,q2) -> ...
```
type query =
  Word of string
| And of query * query
| Or of query * query ;;

type index = string url_set dictionary ;;

let rec eval(q:query) (d:index) : url_set =
  match q with
  | Word x -> Dict.lookup d x
  | And (q1,q2) -> Set.intersect (eval q1 h) (eval q2 h)
  | Or (q1,q2) -> Set.union (eval q1 h) (eval q2 h)
A Better Way

The problem domain talked about an abstract type of **dictionaries** and **sets of URLs**.

type query =
  Word of string
| And of query * query
| Or of query * query ;;

type index = string url_set dictionary ;;

let rec eval(q:query)(d:index) : url_set =
  match q with
  | Word x -> Dict.lookup d x
  | And (q1,q2) -> Set.intersect (eval q1 h) (eval q2 h)
  | Or (q1,q2) -> Set.union (eval q1 h) (eval q2 h)
The problem domain talked about an abstract type of *dictionaries* and *sets of URLs*.

Once we’ve written the client, we know what operations we need on these abstract types.
A Better Way

```ocaml
type query =
    Word of string
  | And of query * query
  | Or of query * query ;;

type index = string url_set dictionary

let rec eval (q:query) (d:index) : url_set =
    match q with
    | Word x -> Dict.lookup d x
    | And (q1,q2) -> Set.intersect (eval q1 h) (eval q2 h)
    | Or (q1,q2) -> Set.union (eval q1 h) (eval q2 h)
```

The problem domain talked about an abstract type of *dictionaries* and *sets of URLs*.

Once we’ve written the client, we know what operations we need on these abstract types.

So we can define an interface, and send a pal off to implement the abstract types dictionary and set.

Later on, when we find out linked lists aren’t so good for sets, we can replace them with balanced trees.
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.”
Rule of thumb: Use the language to enforce the abstraction barrier.

- Murphy’s law for unenforced data abstraction: What is not enforced, will be broken, some time down the line, by a client
- this is what modules, signatures and structures are for
  • reveal little information about how something is implemented
  • provide maximum flexibility for change moving forward.
  • pays off down the line

Like all design rules, we must be able to recognize when the barrier is causing more trouble than it’s worth and abandon it.
- may want to reveal more information for debugging purposes
  • eg: conversion to string so you can print things out
Language-enforced Abstraction

**Rule of thumb:** Use the language to enforce an abstraction.

- **Murphy’s law for unenforced data abstraction:**
  - What is not enforced, will be broken at some point, by a client

- **This is what module systems are for!**
  - reveal little information about *how* something is implemented
  - provide maximum flexibility for change moving forward.
  - pays off down the line

- **Like all design rules, break it when necessary**
  - recognize when a barrier is causing more trouble than it’s worth

- **ML has a particularly great module system**
Building Abstract Types in OCaml

Use OCaml modules to build new abstract data types!

– **signature**: an interface.
  - specifies the abstract type(s) without specifying their implementation
  - specifies the set of operations on the abstract types

– **structure**: an implementation.
  - a collection of type and value definitions
  - notion of an implementation matching or satisfying an interface
    – gives rise to a notion of sub-typing

– **functor**: a parameterized module
  - really, a function from modules to modules
  - allows us to factor out and re-use modules
OCaml Convention:

- 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
- Other modules, like ClientA or ClientB can:
  
  • use *dot notation* to refer to contents of Name. eg: Name.val
  
  • open Set: get access to all elements of Name
    
    – opening a module puts lots of names in your namespace

    *Open modules with discretion!*
At first glance: OCaml modules = C modules?

C has:
- .h files (signatures) similar to .mli files?
- .c files (structures) similar to .ml files?

But ML also has:
- tighter control over type abstraction
  - define abstract, transparent or translucent types in signatures
    - *i.e.*, give none, all or some of the type information to clients
- more structure
  - modules can be defined within modules
  - *i.e.*, signatures and structures can be defined inside files
- more reuse
  - multiple modules can satisfy the same interface
  - the same module can satisfy multiple interfaces
  - modules take other modules as arguments (functors)
- fancy features: dynamic, first class modules!
At first glance: OCaml modules = C modules?

C has:
- .h files (signatures) similar to .mli files?
- .c files (structures) similar to .ml files?

But ML also has:
- tighter control over type abstraction
  - define abstract, transparent, or translucent types in signatures
    - i.e., give none, all, or some type information to clients
  - more reuse
    - modules
    - i.e., signatures can be defined inside files
  - more modules can satisfy the same interface
  - the same module can satisfy multiple interfaces
  - modules take other modules as arguments (functors)
- fancy features: dynamic, first class modules!

ML = Winning!
module type INT_STACK =

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

  sig
    type t
    val empty : unit -> t
    val push : int -> t -> t
    val is_empty : t -> bool
    val pop : t -> t
    val top : t -> int option
  end

convention: when the module is about 1 data type, use t as the name of the type.

clients refer to Stack.t
module type INT_STACK =

  sig
    type t
    val empty : unit -> t
    val push : int -> t -> t
    val is_empty : t -> bool
    val pop : t -> t
    val top : t -> int option
  end

empty and push are abstract *constructors*: functions that build our abstract type.
module type INT_STACK =

    sig
        type t
        val empty : unit -> t
        val push : int -> t -> t
        val is_empty : t -> bool
        val pop : t -> t
        val top : t -> int option
    end

is_empty is an observer — useful for determining properties of the ADT.
module type INT_STACK =

sig

  type t

  val empty : unit -> t

  val push : int -> t -> t

  val is_empty : t -> bool

  val pop : t -> t

  val top : t -> int option

end

pop is sometimes called a mutator (though it doesn’t really change the input)
module type INT_STACK =

sig
  type t
  val empty : unit -> t
  val push : int -> t -> t
  val is_empty : t -> bool
  val pop : t -> t
  val top : t -> int option
end

Top is also an observer, in this functional setting since it doesn’t change the stack.
module type INT_STACK =
  sig
    type t
    (* create an empty stack *)
    val empty : unit -> t

    (* push an element on the top of the stack *)
    val push  : int -> t -> t

    (* returns true iff the stack is empty *)
    val is_empty : t -> bool

    (* pops top element off the stack; returns empty stack if the stack is empty *)
    val pop : t -> t

    (* returns the top element of the stack; returns None if the stack is empty *)
    val top : t -> int option
  end
Signature Comments

• Signature comments are for clients of the module
  – explain what each function should do
    • how it manipulates abstract values (stacks)
  – not how it manipulates concrete values
  – don’t reveal implementation details that should be hidden behind the abstraction

• Don’t copy signature comments into your structures
  – your comments will get out of date in one place or the other
  – an extension of the general rule: don’t copy code

• Place implementation comments inside your structure
  – comments about implementation invariants hidden from client
  – comments about helper functions
module ListIntStack : INT_STACK =

struct
  type t = 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
Inside the module, we know the \textit{concrete type} used to implement the abstract type.
Example Structure

module ListIntStack : INT_STACK =

  struct
    type t = int list
    let empty () : stack = []
    let push (i:int) (s: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

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.
module ListIntStack : INT_STACK =
  struct
    ...
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let i = ListIntStack.top s2
module ListIntStack : INT_STACK =
  struct
    ...
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let i = ListIntStack.top s2
module ListIntStack : INT_STACK =
    struct
    ...
    end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let i = ListIntStack.top s2
    (* i : option int = Some 4 *)
module ListIntStack : INT_STACK =
  struct
    ...
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let i = ListIntStack.top s2
  (* i : option int = Some 4 *)
let j = ListIntStack.top (ListIntStack.pop s2)
  (* j : option int = Some 3 *)
module ListIntStack : INT_STACK =
    struct
        ...
    end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let i = ListIntStack.top s2
    (* i : option int = Some 4 *)
let j = ListIntStack.top (ListIntStack.pop s2)
    (* j : option int = Some 3 *)
open ListIntStack
module ListIntStack : INT_STACK =

  struct
    ...
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let i = ListIntStack.top s2
  (* i : option int = Some 4 *)
let j = ListIntStack.top (ListIntStack.pop s2)
  (* j : option int = Some 3 *)
open ListIntStack
let k = top (pop (pop s2))
  (* k : option int = None *)
module type INT_STACK =

  sig
  
  type t
  
  val push : int -> t -> t
  
  ...

module ListIntStack : INT_STACK

let s2 = ListIntStack.push 4 s1

...

let l = 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.
Example Structure

module ListIntStack (* : INT_STACK *) =

  struct
    type t = int list
  let empty () : stack = []
  let push (i:int) (s:stack) = i::s
  let is_empty (s:stack) =
    match s with
    | [] -> true
    | _::_ -> false

exception EmptyStack

let pop (s:stack) =
  match s with
  | [] -> []
  | _::t -> t

let top (s:stack) =
  match s with
  | [] -> None
  | h::_ -> Some h

end

Note that when you are debugging, you may want to comment out the signature ascription so that you can access the contents of the module.
module ListIntStack (* : INT_STACK *) =
  struct
    ...
  end

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

... 
let l = List.rev s2
  (* l : int list = [3; 4] *)
When you put the signature on here, you are restricting client access to the information in the signature (which does not reveal that stack = int list.) So clients can only use the stack operations on a stack value (not list operations.)
module type INT_STACK =
  sig
    type stack
    ...
    val inspect : stack -> int list
    val run_unit_tests : unit -> unit
  end

module ListIntStack : INT_STACK =
  struct
    type stack = int list
    ...
    let inspect (s: stack) : int list = s
    let run_unit_tests () : unit = ...
  end

Another technique:
Add testing components to your signature.

Another option we will see:
have 2 signatures, one for testing and one for the rest of the code)
“CORNER CASES”
module type INT_STACK =
  sig
    type t
    (* create an empty stack *)
    val empty : unit -> t

    (* push an element on the top of the stack *)
    val push : int -> t

    (* returns true if the stack is empty *)
    val is_empty : t -> bool

    (* pops top element off the stack; returns empty stack if the stack is empty *)
    val pop : stack -> stack

    (* returns the top element of the stack; returns None if the stack is empty *)
    val top : t -> int option
  end

Isn’t that a bit bogus?
Design choices

sig
type t
(* pops top element; returns empty if empty *)
val pop : t -> t end

sig
type t
(* pops top element; returns \textit{arbitrary stack} if empty *)
val pop : t -> t end

sig
type t
(* pops top element; returns option *)
val pop : t -> t option end

sig
type t
exception EmptyStack
(* pops top element; raises \texttt{EmptyStack} if empty *)
val pop : t -> t end
Design choices

For some functions, there are some input values outside the domain of the function.

It’s reasonable to say the function returns an arbitrary result on those inputs.

When proving things about the program, there’s an extra proof obligation: Prove that the input is in the domain of the function.

```
sig
  type t
  (* pops top element; returns arbitrary stack if empty *)
  val pop : t -> t
end
```
Design choices

For some functions, there are some input values outside the *domain* of the function.

This is *not completely crazy*. One might still be able to guarantee that the input is always in the domain of the function.

It’s what the C language does, for example.

```ml
sig
  type t
  (* pops top element; 
     crashes the program 
     if empty *)
  val pop : t -> t
end
```
Design choices

For some functions, there are some input values outside the domain of the function.

This is not completely crazy. One might still be able to guarantee that the input is always in the domain of the function.

It’s what the C language does, for example.

But it’s almost completely crazy. This is the biggest source of security vulnerabilities ever. It’s why the hackers can drive your car, steal your money, read your e-mails, ...
It’s also reasonable to say the function returns a *specified, convenient*, result on those inputs. This is pretty much the same thing, in practice.
Design choices

For some functions, there are some input values outside the *domain* of the function.

That’s what exceptions are for! Raise an exception for values not in the domain.

```ocaml
sig
  type stack
  exception EmptyStack
  (* pops top element; raises EmptyStack if empty *)
  val pop : stack -> stack
end
```
let const (x: 'a) (y: 'b) : 'a = x

Claim: for all expressions e, const 7 e == 7.

Is it true?

sig
  type stack
  exception EmptyStack
  (* pops top element;
     raises EmptyStack if empty *)
  val pop : stack -> stack
end
let const (x: ‘a) (y: ‘b) : ‘a = x

Claim: for all expressions e, const 7 e == 7.

Is it true?

**No!**

const 7 (pop (empty())) ≠ 7

To reason about expressions, you must prove the exception will not be raised in the particular case (i.e., input in domain).
Finally, you can just use option types in the obvious way.

```
sig
  type stack
  (* pops top element; returns option *)
  val pop:
    stack -> stack option
end
```
All of these are reasonable design choices!
ANOTHER EXAMPLE
module type QUEUE =
    sig
        type 'a queue
        val empty : unit -> 'a queue
        val enqueue : 'a -> 'a queue -> 'a queue
        val is_empty : 'a queue -> bool
        exception EmptyQueue
        val dequeue : 'a queue -> 'a queue
        val front : 'a queue -> 'a
    end
Polymorphic Queues

These queues are re-usable for different element types.

Here's an exception that client code might want to catch.
module AppendListQueue : QUEUE =

  struct
    type 'a queue = 'a list
    let empty() = []
    let enqueue(x:'a)(q:'a queue) : 'a queue = q @ [x]
    let is_empty(q:'a queue) =
      match q with
      | [] -> true
      | _:::_ -> false

  ...

end
module AppendListQueue : QUEUE =

  struct
    type 'a queue = 'a list

    let empty() = []

    let enqueue(x:'a)(q:'a queue) : 'a queue = q @ [x]

    let is_empty(q:'a queue) = ...

    exception EmptyQueue

    let deq(q:'a queue) : ('a * 'a queue) =
      match q with
      | [] -> raise EmptyQueue
      | h::t -> (h,t)

    let dequeue(q:'a queue) : 'a queue = snd (deq q)

    let front(q:'a queue) : 'a = fst (deq q)

  end
module AppendListQueue : QUEUE =

struct
  type `a queue = `a list
  let empty() = []
  let enqueue(x:`a)(q:`a queue) : `a queue = q @ [x]
end

exception EmptyQueue

let deq(q:`a queue) : (`a * `a queue) =
  match q with
  | [] -> raise EmptyQueue
  | h::t -> (h,t)

let dequeue(q:`a queue) : `a queue = snd (deq q)
let front(q:`a queue) : `a = fst (deq q)

Notice deq is a helper function that doesn’t show up in the signature.
You can't use it outside the module.
module AppendListQueue : QUEUE =
struct
  type 'a queue = 'a list
  let empty() = []
  let enqueue(x:'a)(q:'a queue) : 'a queue = q @ [x]
  let is_empty(q:'a queue) = ...

exception EmptyQueue

let deq(q:'a queue) : ('a * 'a queue) =
  match q with
  | [] -> raise EmptyQueue
  | h::t -> (h,t)

let dequeue(q:'a queue) : 'a queue = snd (deq q)
let front(q:'a queue) : 'a = fst (deq q)
end

enqueue takes time proportional to the length of the queue 😞

Dequeue runs in constant time 😊
module DoubleListQueue : QUEUE =
    struct
        type 'a queue = {front: 'a list; rear: 'a list}
    ...
end
In Pictures

abstraction

a, b, c, d, e

implementation

\{\text{front}=[a; b]; \text{rear}=[e; d; c]\}

- let q0 = empty \{front=[]; rear=[]\}
- let q1 = enqueue 3 q0 \{front=[]; rear=[3]\}
- let q2 = enqueue 4 q1 \{front=[]; rear=[4; 3]\}
- let q3 = enqueue 5 q2 \{front=[]; rear=[5; 4; 3]\}
- let q4 = dequeue q3 \{front=[4; 5]; rear=[]\}
- let q5 = dequeue q4 \{front=[5]; rear=[]\}
- let q6 = enqueue 6 q5 \{front=[5]; rear=[6]\}
- let q7 = enqueue 7 q6 \{front=[5]; rear=[7; 6]\}
module DoubleListQueue : QUEUE =

  struct
    type 'a queue = {front:'a list;
         rear:'a list}
  let empty() = {front=[]; rear=[]}

  let enqueue x q = {front=q.front; rear=x::q.rear}

  let is_empty q =
    match q.front, q.rear with
    | [], [] -> true
    | _, _ -> false
    ...
  end
An Alternative Implementation

module DoubleListQueue : QUEUE =
  struct
    type 'a queue = {front:'a list;
                    rear:'a list}
  exception EmptyQueue

let deq (q:'a queue) : 'a * 'a queue =
  match q.front with
  | h::t -> (h, {front=t; rear=q.rear})
  | [] -> match List.rev q.rear with
         | h::t -> (h, {front=t; rear=[]})
         | [] -> raise EmptyQueue

let dequeue (q:'a queue) : 'a queue = snd(deq q)
let front (q:'a queue) : 'a = fst(deq q)
end
How would we design an abstraction?

Think:
  – what data do you want?
    • define some types for your data
  – what operations on that data do you want?
    • define some types for your operations

Write some test cases:
  – example data, operations

From this, we can derive a signature:
  – list the types
  – list the operations with their types
  – don’t forget to provide enough operations that you can debug!

Then we can build an implementation:
  – when prototyping, build the simplest thing you can.
  – later, we can swap in a more efficient implementation.
  – (assuming we respect the abstraction barrier.)
The stack and queue interfaces are quite similar:

```ocaml
module type STACK =
  sig
    type 'a stack
    val empty : unit -> 'a stack
    val push  : 'a -> 'a stack -> 'a stack
    val is_empty : 'a stack -> bool
    exception EmptyStack
  val pop  
  val top  
end

module type QUEUE =
  sig
    type 'a queue
    val empty : unit -> 'a queue
    val enqueue : 'a -> 'a queue -> 'a queue
    val is_empty : 'a queue -> bool
    exception EmptyQueue
    val dequeue : 'a queue -> 'a queue
    val front : 'a queue -> 'a
end
```
The stack and queue interfaces are *quite* similar:

```ocaml
module type STACK =
  sig
    type 'a t
    val empty : unit -> 'a t
    val insert : 'a -> 'a t -> 'a t
    val is_empty : 'a t -> bool
  exception Empty
  val remove : 'a t -> 'a t
  val first : 'a t -> 'a
  end

module type QUEUE =
  sig
    type 'a t
    val empty : unit -> 'a t
    val insert : 'a -> 'a t -> 'a t
    val is_empty : 'a t -> bool
  exception Empty
  val remove : 'a t -> 'a t
  val first : 'a t -> 'a
  end
```
It’s a good idea to factor out patterns

Stacks and Queues share common features.

Both can be considered “containers”

Create a reusable container interface!

```ocaml
module type CONTAINER =
  sig
    type 'a t
    val empty : unit -> 'a t
    val insert : 'a -> 'a t -> 'a t
    val is_empty : 'a t -> bool
    exception Empty
    val remove : 'a t -> 'a t
    val first : 'a t -> 'a
  end
```
It’s a good idea to factor out patterns

Stacks and Queues share common features.

Both can be considered "containers". Create a reusable container interface!

```
module type STACK =
  sig
    type 'a t
    val empty : unit -> 'a t
    val insert : 'a -> 'a t -> 'a t
    val is_empty : 'a t -> bool
    exception Empty
    val remove : 'a t -> 'a t
    val first : 'a t -> 'a
  end
```

More than quite similar! Exactly the same

```
module type CONTAINER =
  sig
    type 'a t
    val empty : unit -> 'a t
    val insert : 'a -> 'a t -> 'a t
    val is_empty : 'a t -> bool
    exception Empty
    val remove : 'a t -> 'a t
    val first : 'a t -> 'a
  end
```
It’s a good idea to factor out patterns

Stacks and Queues share common features. Both can be considered “containers.” Create a reusable container interface!

```ocaml
module type CONTAINER = 
  sig
    type 'a t
    val empty : unit -> 'a t
    val insert : 'a -> 'a t -> 'a t
    val is_empty : 'a t -> bool
    exception Empty
    val remove : 'a t -> 'a t
    val first : 'a t -> 'a
  end
```

```ocaml
module type STACK = 
  module type QUEUE = 
    sig
      type 'a t
      val empty : unit -> 'a t
      val insert : 'a -> 'a t -> 'a t
      val is_empty : 'a t -> bool
      exception Empty
      val remove : 'a t -> 'a t
      val first : 'a t -> 'a
    end
```

still exactly the same!
It’s a good idea to factor out patterns

module type CONTAINER = sig ... end

module Queue : CONTAINER = struct ... end
module Stack : CONTAINER = struct ... end

module DepthFirstSearch : SEARCHER =
  struct
    type to_do : Graph.node Queue.t
  end

module BreadthFirstSearch : SEARCHER =
  struct
    type to_do : Graph.node Stack.t
  end

Still repeated code!

Breadth-first and depth-first search code is the same!

Just use different containers!

Need parameterized modules!
FUNCTORS

David MacQueen
Bell Laboratories 1983-2001
U. of Chicago 2001-2012

Designer of ML module system,
functors,
sharing constraints, etc.
Suppose I ask you to write a generic package for matrices.
  – e.g., matrix addition, matrix multiplication

The package should be *parameterized* by the element type.
  – We may want to use ints or floats or complex numbers or binary values or ... for the elements.
  – And the elements still have a collection of operations on them:
    • addition, multiplication, zero element, etc.

What we'll see:
  – **RING**: a signature to describe the type (and necessary operations) for matrix elements
  – **MATRIX**: a signature to describe the available operations on matrices
  – **DenseMatrix**: a functor that will generate a MATRIX with a specific RING as an element type
module type RING =

sig

  type t
  val zero : t
  val one : t
  val add : t -> t -> t
  val mul : t -> t -> t

end
module IntRing =
  struct
    type t = int
    let zero = 0
    let one = 1
    let add x y = x + y
    let mul x y = x * y
  end

module FloatRing =
  struct
    type t = float
    let zero = 0.0
    let one = 1.0
    let add = (+.)
    let mul = ( *. )
  end

module BoolRing =
  struct
    type t = bool
    let zero = false
    let one = true
    let add x y = x || y
    let mul x y = x && y
  end
module type MATRIX =
  sig
    type elt
    type matrix
    val matrix_of_list : elt list list -> matrix
    val add : matrix -> matrix -> matrix
    val mul : matrix -> matrix -> matrix
  end
module DenseMatrix (R:RING) : (MATRIX with type elt = R.t) =
struct
  ...
end
module DenseMatrix (R:RING) : (MATRIX with type elt = R.t) =

struct

...

end

Argument R must be a RING
Result must be a MATRIX
Specify Result.elt = R.t
module DenseMatrix (R:RING) : (MATRIX with type elt = R.t) =
struct
...
end

module IntMatrix = DenseMatrix(IntRing)
module FloatMatrix = DenseMatrix(FloatRing)
module BoolMatrix = DenseMatrix(BoolRing)

Use DenseMatrix like it is a function from modules to modules
module DenseMatrix (R:RING) : (MATRIX with type elt = R.t) =
struct
  ...
end

module type MATRIX =
sig
  type elt
  type matrix

  val matrix_of_list : elt list list -> matrix

  val add : matrix -> matrix -> matrix
  val mul : matrix -> matrix -> matrix
end

module IntMatrix = DenseMatrix(IntRing)
module FloatMatrix = DenseMatrix(FloatRing)
module BoolMatrix = DenseMatrix(BoolRing)
The DenseMatrix Functor

module DenseMatrix (R:RING) : (MATRIX with type elt = R.t) =
struct

If the "with" clause is redacted then IntMatrix_elt is abstract -- we could never build a matrix because we could never generate an elt.

end

module type MATRIX =
sig
  type elt
  type matrix

  val matrix_of_list : elt list list -> matrix

  val add : matrix -> matrix -> matrix
  val mul : matrix -> matrix -> matrix
end

module IntMatrix = DenseMatrix(IntRing)
module FloatMatrix = DenseMatrix(FloatRing)
module BoolMatrix = DenseMatrix(BoolRing)
module DenseMatrix (R:RING) : (MATRIX with type elt = R.t) =
struct
...
end

module type MATRIX =
sig
  type elt = int
  type matrix

  val matrix_of_list : elt list list -> matrix
  val add : matrix -> matrix -> matrix
  val mul : matrix -> matrix -> matrix
end

module IntMatrix = DenseMatrix(IntRing)
module FloatMatrix = DenseMatrix(FloatRing)
module BoolMatrix = DenseMatrix(BoolRing)
module DenseMatrix (R:RING) : (MATRIX with type elt = R.t) =
struct

  The "with" clause makes IntMatrix.elt equal to int -- we can build a matrix from any int list list

module type MATRIX =
sig
  type elt = int
  type matrix

  val matrix_of_list : elt list list -> matrix
  val add : matrix -> matrix -> matrix
  val mul : matrix -> matrix -> matrix
end

module IntMatrix = DenseMatrix(IntRing)
module FloatMatrix = DenseMatrix(FloatRing)
module BoolMatrix = DenseMatrix(BoolRing)
module DenseMatrix (R:RING) : (MATRIX with type elt = R.t) =
struct
  type elt = R.t
  type matrix = (elt list) list
  let matrix_of_list rows = rows
  let add m1 m2 =
    List.map (fun (r1,r2) ->
      List.map (fun (e1,e2) -> R.add e1 e2))
      (List.combine r1 r2))
    (List.combine m1 m2)
  let mul m1 m2 = (* good exercise *)
end

module IntMatrix = DenseMatrix(IntRing)
module FloatMatrix = DenseMatrix(FloatRing)
module BoolMatrix = DenseMatrix(BoolRing)

Satisfies the sharing constraint
ANONYMOUS STRUCTURES
Another Example

module type UNSIGNED_BIGNUM =

sig
  type ubignum

  val fromInt : int -> ubignum
  val toInt : ubignum -> int
  val plus : ubignum -> ubignum -> ubignum
  val minus : ubignum -> ubignum -> ubignum
  val times : ubignum -> ubignum -> ubignum

  ...

end
An Implementation

module My_UBignum_1000 : UNSIGNED_BIGNUM =

struct

  let base = 1000

  type ubignum = int list

  let toInt(b:ubignum):int = ...

  let plus(b1:ubignum)(b2:ubignum):ubignum = ...

  let minus(b1:ubignum)(b2:ubignum):ubignum = ...

  let times(b1:ubignum)(b2:ubignum):ubignum = ...

  ...

end

What if we want to change the base? Binary? Hex? $2^{32}$? $2^{64}$?
module type BASE =
  sig
    val base : int
  end

module UbignumGenerator(Base:BASE) : UNSIGNED_BIGNUM =
  struct
    type ubignum = int list
    let toInt(b:ubignum):int =
      List.fold_left (fun a c -> c*Base.base + a) 0 b   ...
  end

module Ubignum_10 =
  UbignumGenerator(struct let base = 10 end)

module Ubignum_2 =
  UbignumGenerator(struct let base = 2 end)
SIGNATURE SUBTYPING
Subtyping

• A module matches any interface as long as it provides \textit{at least} the definitions (of the right type) specified in the interface.

• But as we saw earlier, the module can have more stuff.
  – e.g., the \texttt{deq} function in the Queue modules

• Basic principle of subtyping for modules:
  – wherever you are expecting a module with signature $S$, you can use a module with signature $S'$, as long as all of the stuff in $S$ appears in $S'$.
  – That is, $S'$ is a bigger interface.
module type GROUP =
  sig
    type t
    val zero : t
    val add : t -> t -> t
  end
module type RING =
  sig
    type t
    val zero : t
    val one : t
    val add : t -> t -> t
    val mul : t -> t -> t
  end
module IntGroup : GROUP = IntRing
module FloatGroup : GROUP = FloatRing
module BoolGroup : GROUP = BoolRing
module type GROUP =
  sig
    type t
    val zero : t
    val add : t -> t -> t
  end
module type RING =
  sig
    type t
    val zero : t
    val one : t
    val add : t -> t -> t
    val mul : t -> t -> t
  end
module IntGroup : GROUP = IntRing
module FloatGroup : GROUP = FloatRing
module BoolGroup : GROUP = BoolRing

RING is a sub-type of GROUP.
module type GROUP =
  sig
    type t
    val zero : t
    val add : t -> t -> t
  end
module type RING =
  sig
    type t
    val zero : t
    val one : t
    val add : t -> t -> t
    val mul : t -> t -> t
  end
module IntGroup : GROUP = IntRing
module FloatGroup : GROUP = FloatRing
module BoolGroup : GROUP = BoolRing

There are more modules matching the GROUP interface than the RING one.
module type GROUP =
  sig
    type t
    val zero : t
    val add : t -> t -> t
  end
module type RING =
  sig
    type t
    val zero : t
    val one : t
    val add : t -> t -> t
    val mul : t -> t -> t
  end
module IntGroup : GROUP = IntRing
module FloatGroup : GROUP = FloatRing
module BoolGroup : GROUP = BoolRing

Any module expecting a GROUP can be passed a RING.
module type GROUP =
  sig
    type t
    val zero : t
    val add : t -> t -> t
  end
module type RING =
  sig
    include GROUP
    val one : t
    val mul : t -> t -> t
  end
module IntGroup : GROUP = IntRing
module FloatGroup : GROUP = FloatRing
module BoolGroup : GROUP = BoolRing

The include primitive is like cutting-and-pasting the signature’s content here.
module type GROUP =
  sig
    type t
    val zero : t
    val add : t -> t -> t
  end

module type RING =
  sig
    include GROUP
    val one : t
    val mul : t -> t -> t
  end

module IntGroup : GROUP = IntRing
module FloatGroup : GROUP = FloatRing
module BoolGroup : GROUP = BoolRing

That **ensures** we will be a sub-type of the included signature.
ANOTHER EXAMPLE OF FUNCTORS
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
    val union : set -> set -> set
    val intersect : set -> set -> set
    val remove : elt -> set -> set
    val member : elt -> set -> bool
    val choose : set -> (elt * set) option
    val fold : (elt -> 'a -> 'a) -> 'a -> set -> 'a

  end
Our Set Implementation is a Functor:

```ocaml
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)
```
Our Set Implementation is a Functor:

```ocaml
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 – given a module argument for Elt, it generates a new module.
Our Set Implementation is a Functor:

```plaintext
module ListSet (Elt : sig type t end) : (SET with elt = Elt.t) =
value
  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.
Our Set Implementation is a Functor:

```ocaml
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
Our Set Implementation is a Functor:

```ocaml
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.
Our Set Implementation is a 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.
module ListSet (Elt : sig type t end) : (SET with elt = Elt.t) =

match xs with
| [] -> true
| _::_ -> false

We know that IntListSet.elt = int.
Our Set Implementation is a Functor:

```ocaml
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
```

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

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

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

equal to int so we can actually build a set using insertions!
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]
    let insert (x:elt) (s:set) : set =
        if List.mem x s then s else x:::s
    ...
end
module ListSet (Elt : sig type t end) :
  :(SET with elt = Elt.t) =

struct
    type elt = Elt.t
    type set = elt list
    ...
    let insert (x:elt) (s:set) : set =
        if List.mem x s then s else x::s
    let union (s1:set) (s2:set) : set = ???
end
Let’s Write the Rest of the Functor

```ocaml
module ListSet (Elt : sig type t end) : (SET with elt = Elt.t) =

struct
  type elt = Elt.t
  type set = elt list
  ...
  let insert (x:elt) (s:set) : set =
    if List.mem x s then s else x::s
  let union (s1:set) (s2:set) : set =
    s1 @ s2
  ...
end
```

Ugh. Wastes space if s1 and s2 have duplicates. (Also, makes remove harder...)
module ListSet (Elt : sig type t end) :
  (SET with elt = Elt.t) =

struct
  type elt = Elt.t
  type set = elt list
  ...
  let insert (x:elt) (s:set) : set =
    if List.mem x s then s else x::s
  let union (s1:set) (s2:set) : set =
    List.fold_right insert s1 s2
  ...
end

gets rid of the duplicates. Now remove can stop once it finds the element.
Let’s Write the Rest of the Functor

module ListSet (Elt : sig type t end) :
  (SET with elt = Elt.t) =

struct
  type elt = Elt.t
  type set = elt list
...

let insert (x:elt) (s:set) : set =
  if List.mem x s then s else x::s

let union (s1:set) (s2:set) : set =
  List.fold_right insert s1 s2
...
end

Gets rid of the duplicates. Now remove can stop once it finds the element.
Let’s Write the Rest of the Functor

```
module ListSet (Elt : sig type t end) :
    (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
...
let insert (x:elt) (s:set) : set =
  if List.mem x s then s else x::s
let union (s1:set) (s2:set) : set =
  List.fold_right insert s1 s2
...
end
```

But List.mem and List.fold_right take time proportional to the length of the list. So union is quadratic.

Gets rid of the duplicates. Now remove can stop once it finds the element.
module ListSet (Elt : sig type t end)
  : (SET with elt = Elt.t) =

struct
  type elt = Elt.t
  type set = elt list
  ...
  let insert (x:elt) (s:set) : set =
    if List.mem x s then s else x::s
  let union (s1:set) (s2:set) : set =
    List.fold_right insert s1 s2
  ...
end

If we knew that s1 and s2 were sorted we could use the merge from mergesort to compute the sorted union in linear time.
module type COMPARATOR = sig
  type t
  val compare : t -> t -> Order.order
end

module SortedListSet (Elt : COMPARATOR)
  : (SET with elt = Elt.t) =
struct ...
  let rec insert (x:elt) (s:set) : set =
    match s with
    | [] -> [x]
    | h::t -> (match Elt.compare x h with
                | Less -> x::s
                | Eq -> s
                | Greater -> h::(insert x t)) ...
end
module type COMPARATOR = sig
  type t
  val compare : t -> t -> Order.order
end

module SortedListSet (Elt : COMPARATOR) : (SET with elt = Elt.t) =
  struct ...
    let rec insert (x:elt) (s:set) :
      match s with
      | [] -> [x]
      | h::t -> (match Elt.compare x h with
                  | Less -> x::s
                  | Eq -> s
                  | Greater -> h::(insert x t)) ...
  end

To support the sorting, I’m passing in a comparison operation to go with the element type.
module SortedListSet (Elt : COMPARATOR)

  : (SET with elt = Elt.t) =

struct ...

  let rec union (s1:set) (s2:set) : set =
    match s1, s2 with
      | [], _ -> s2
      | _, [] -> s1
      | h1::t1, h2::t2 ->
        (match Elt.compare h1 h2 with
          | Less -> h1::(union t1 s2)
          | Eq -> h1::(union t1 t2)
          | _ -> h2::(union s1 t2))

  ...
end
module SortedListSet (Elt : COMPARATOR) : (SET with elt = Elt.t) =

  struct ...

  let rec union (s1:set) (s2:set) : set = ...

  let insert (x:elt) (s:set) : set = union [x] s ;;

end
module BitVectorSet (Elt : sig
  type t
  val index : t -> int
  val max : int
end)
  : (SET with elt = Elt.t) =

struct
  type set = bool array
  let empty = Array.create Elt.max false
  let member x s = s.(Elt.index x)
  let union s1 s2 =
    Array.init Elt.max
    (fun i -> s1.(i) || s2.(i))
  let intersect s1 s2 =
    Array.init Elt.max
    (fun i -> s1.(i) && s2.(i))
  ...

module BSTreeSet(Elt : sig type t
                        val compare : t -> t -> Order.order
        end) : (SET with elt = Elt.t) =

struct
  type set = Leaf | Node of set * elt * set
  let empty() = Leaf
  let rec insert (x:elt) (s:set) : set =
    match s with
    | Leaf -> Node(Leaf,x,Leaf)
    | Node(left,e,right) ->
      (match Elt.compare x e with
        | Eq -> s
        | Less -> Node(insert x left, e, right)
        | Greater -> Node(left, e, insert x right))
  let rec member (x:elt) (s:set) : bool =
    match s with
    | Leaf -> false
    | Node(left,e,right) ->
      (match Elt.compare x e with
        | Eq -> true
        | Less -> member x left
        | Greater -> member x right)
    ...
  end
Wrap up and Summary

• It is often tempting to break the abstraction barrier.
  – e.g., during development, you want to print out a set, so you just call a convenient function you have lying around for iterating over lists and printing them out.

• But the whole point of the barrier is to support future change in implementation.
  – e.g., moving from unsorted invariant to sorted invariant.
  – or from lists to balanced trees.

• Many languages provide ways to leak information through the abstraction barrier.
  – “good” clients should not take advantage of this.
  – but they always end up doing it.
  – so you end up having to support these leaks when you upgrade, else you’ll break the clients.
Wrap up and Summary

• It is often tempting to break the abstraction barrier.
  – e.g., during development, you want to print out a set, just call a convenient function you have lying around for iterating over lists and printing them out.
• But the whole point of the barrier is to support future changes in implementation.
  – e.g., moving from unsorted invariant to sorted invariant.
  – or from lists to balanced trees.
• Many languages provide ways to leak information through the abstraction barrier.
  – “good” clients should not take advantage of this. Always end up doing it.
  – you end up having to support these leaks when you upgrade, else you’ll break the clients.
OCaml’s linguistic mechanisms include:
  – *signatures* (interfaces)
  – *structures* (implementations)
  – *functors* (functions from modules to modules)

We can use the module system
  – provides support for *name-spaces*
  – *hiding information* (types, local value definitions)
  – *code reuse* (via functors, reuseable interfaces, reuseable modules)

Information hiding allows design in terms of *abstract* types and algorithms.
  – think “sets” not “lists” or “arrays” or “trees”
  – think “document” not “strings”
  – the less you reveal, the easier it is to replace an implementation
  – use linguistic mechanisms to implement information hiding
    • invariants written down as comments are easy to violate
    • use the type checker to guarantee you have strong protections in place