Modules
and Abstract Data Types

COS 326
David Walker
Princeton University

Building Abstract Types in OCaml

- We can use the module system of OCaml 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

functor kitten

The Abstraction Barrier

Rule of thumb: use the language to *enforce* the abstraction barrier.

- *Second rule of thumb*: What is not enforced automatically by the controller will be broken some time down the line by a client
- this is what modules, signatures and structures are for
  - reveal as little information about *how* something is implemented as you can.
  - provides 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

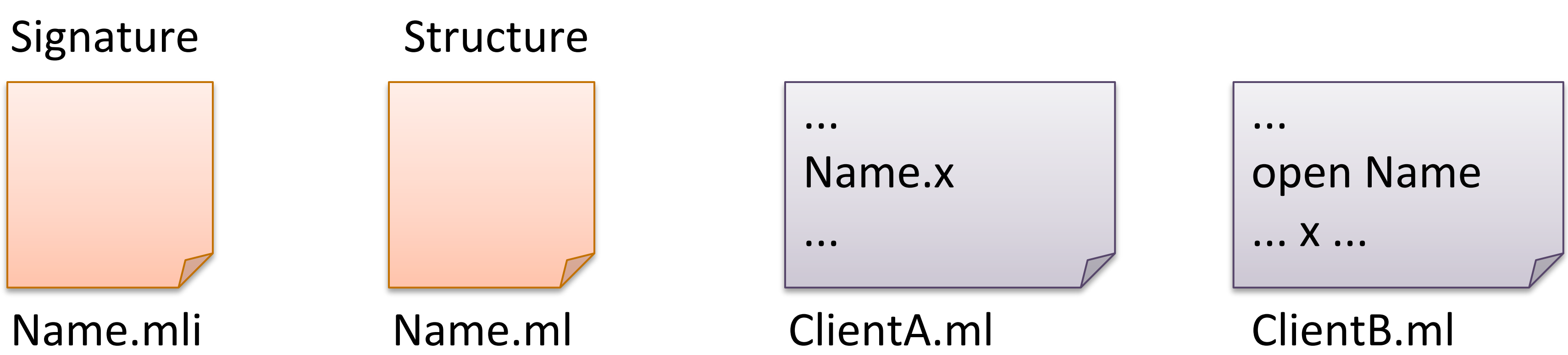
ML is particular good at allowing you to define flexible and yet enforceable abstraction barriers

- precise control over how much of the type is left abstract
- different amounts of information can be revealed in different contexts
- type checker helps you detect violations of the abstraction barrier

Simple Modules

OCaml Convention:

- file Name.ml (or name.ml) is a *structure* implementing a module named Name
- file Name.mli (or 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
    - ie: give none, all or some of the type information to clients
- more structure
  - modules can be defined within modules
  - ie: 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

Example Signature

```
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 option
    val top : stack -> int option
  end
```

Example Signature

```
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 option
    val top : stack -> int option
  end
```

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

Example Signature

```
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 option
    val top : stack ->
  end
```

is_empty is an ***observer*** – useful for determining properties of the ADT.

Example Signature

```
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 option
    val top : stack -> int option
  end
```

pop is sometimes called a ***mutator*** (though it doesn't really change the input)

Example Signature

```
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 option
    val top : stack -> int option
  end
```

top is also an ***observer***, in this functional setting since it doesn't change the stack.

A Better Signature

```
module type INT_STACK =
  sig
    type stack
    (* create an empty stack *)
    val empty : unit -> stack
    (* push an element on the top of the stack *)
    val push  : int -> stack -> stack
    (* returns true iff the stack is empty *)
    val is_empty : stack -> bool
    (* pops top element off the stack; returns None
       if the stack is empty *)
    val pop : stack -> stack option
    (* returns the top element of the stack; returns
        None if the stack is empty *)
    val top : stack -> 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 does it
  - don't reveal implementation details that should be hidden behind the abstraction
- Don't copy signature comments in to 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

Example Structure

```
module ListIntStack : INT_STACK =
  struct
    type stack = 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) =
      match s with
       | [] -> None
       | _::t -> Some t
    let top (s:stack) =
      match s with
       | [] -> None
       | h::_ -> Some h
  end
```

Example Structure

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

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

Example Structure

```
module ListIntStack : INT_STACK =
  struct
    type stack = int list
    let empty () : stack = []
    let push (i:int) (s:stack) =
    let is_empty (s:stack) =
      match s with
       | [] -> true
       | _::_ -> false
    let pop (s:stack) =
      match s with
       | [] -> None
       | _::t -> Some t
    let top (s:stack) =
      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;;
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;;
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;;
ListIntStack.top s2;;
- : int option = 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;;
ListIntStack.top s2;;
- : int option = Some 4
ListIntStack.top (ListIntStack.pop s2) ;;
- : int option = 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;;
ListIntStack.top s2;;
- : int option = Some 4
ListIntStack.top (ListIntStack.pop s2) ;;
- : int option = 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;;
ListIntStack.top s2;;
- : int option = Some 4
ListIntStack.top (ListIntStack.pop s2) ;;
- : int option = Some 3
open ListIntStack;;
top (pop (pop s2));;
- : int option = None
```

An Example Client

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

module ListIntStack : INT_STACK

let s2 = ListIntStack.push 4
…
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 stack = 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
       | [] -> raise EmptyStack
       | _::t -> t
    let top (s:stack) =
      match s with
       | [] -> raise EmptyStack
       | h::_ -> 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.

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;;

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

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

Example Structure

```
module ListIntStack : INT_STACK =
  struct
    type stack = int list
    let empty () : stack = []
    let push (i:int) (s:stack) =
    let is_empty (s:stack) =
      match s with
       | [ ] -> true
       | _::_ -> false
    exception EmptyStack
    let pop (s:stack) =
      match s with
       | [] -> raise EmptyStack
       | _::t -> t
    let top (s:stack) =
      match s with
       | [] -> raise EmptyStack
       | h::_ -> h
  end
```

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.)

Example Structure

```
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)

ANOTHER EXAMPLE

Polymorphic Queues

```
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

```
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 -> queue
    val front : ‘a queue -> ‘a
  end
```

These queues are re-usable for different element types.

Here's an exception that client code might want to catch

One Implementation

```
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
```

One Implementation

```
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
```

One Implementation

```
module AppendListQueue : QUEUE =
  struct
    type 'a queue = 'a list
    let empty() = []
    let enqueue(x:'a)(q:'a queue) : 'a
    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 que
    let front(q:'a queue) : 'a = fst (de
end
```

Notice deq is a helper function that doesn't show up in the signature.

You can't use it outside the module.

One Implementation

```
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 *          que
      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
```

Notice enqueue takes time proportional to the length of the queue

Dequeue runs in constant time.

An Alternative Implementation

```
module DoubleListQueue : QUEUE =
  struct
    type ‘a queue = {front:’a list; rear:’a list}




    ...




end
```

In Pictures

abstraction → implementation

a, b, c, d, e → `{front=[a; b];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]}
```

An Alternative Implementation

```
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
```

REUSING INTERFACES

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.)

Common Interfaces

- The stack and queue interfaces are quite similar:

```
module type STACK =
  sig
    type ‘a stack
    val empty : unit -> ‘a stack
    val push  : int -> ‘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
```

It's a good idea to factor out patterns

- Stacks and Queues share common features.
- Both can be considered "containers"
- Create a reuseable container interface!

```
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

```
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

Matrices

- 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 matix 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

Ring Signature

```
module type RING =
  sig
    type t
    val zero : t
    val one  : t
    val add  : t -> t -> t
    val mul  : t -> t -> t
  end
```

Some Rings

```
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
```

Matrix Signature

```
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
```

The DenseMatrix Functor

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

  ...







end
```

The DenseMatrix Functor

```
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

The DenseMatrix Functor

```
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

The DenseMatrix Functor

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

  ...

end
```

redacted

```
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
```

abstract = unknown!

non-existant

```
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

end

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

redacted

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

```
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
```

abstract = unknown!

non-existant

The DenseMatrix Functor

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

   ...

end

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

sharing constraint

```
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
```

known to be int when R.t = int like when R = IntRing

list of list of ints

The DenseMatrix Functor

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

end
```

sharing constraint

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
```

known to be int when R.t = int like when R = IntRing

list of list of ints

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

Matrix Functor

```
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?

Another Functor Example

```
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) ;;
```

Anonymous structures

SIGNATURE SUBTYPING

Subtyping

A module matches any interface as long as it provides *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 deq function in the Queue modules

Basic principle of *subtyping* for modules:

- consider signatures S and S'
- S is a subtype of S' if S provides all of the functionality that S' provides.
  - ie: wherever you are expecting a module with signature S', you can use a module with signature S instead.
  - notation: S <= S' is read "S is a subtype of S'"
- For example:
  - S <= S' if S contains more components than S'
  - S <= S' if S' hides (renders abstract) some of the types that appear in S

Groups versus Rings

```
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
```

Groups versus Rings

```
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 subtype of GROUP.

Groups versus Rings

```
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.

Groups versus Rings

```
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.

Groups versus Rings

```
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.

Groups versus Rings

```
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.

OCAML COLLECTION LIBRARIES

Collection Data Types

- Many different kinds of collections:
  - Sets
  - Maps (Symbol Tables)
  - Queues
  - Graphs
  - ...
- Modules with two abstract types:
  - element type
  - collection type
- We often build collections using functors

A Common Structure for Collection Modules

```
module MyCollection =
 struct

   module type Element =
     sig  type t  ... end

   module type S =
     sig ... end

   module Make (Arg : Element) : S with type t = Arg.t =
     struct ... end

 end
```

Type and necessary operations on collection element

Type and necessary operations of collection as a whole

Functor to create the collection

Modules in the Wild

- The OCaml Map Module
  - creates polymorphic symbol tables

http://caml.inria.fr/pub/docs/manual-ocaml/libref/Map.html

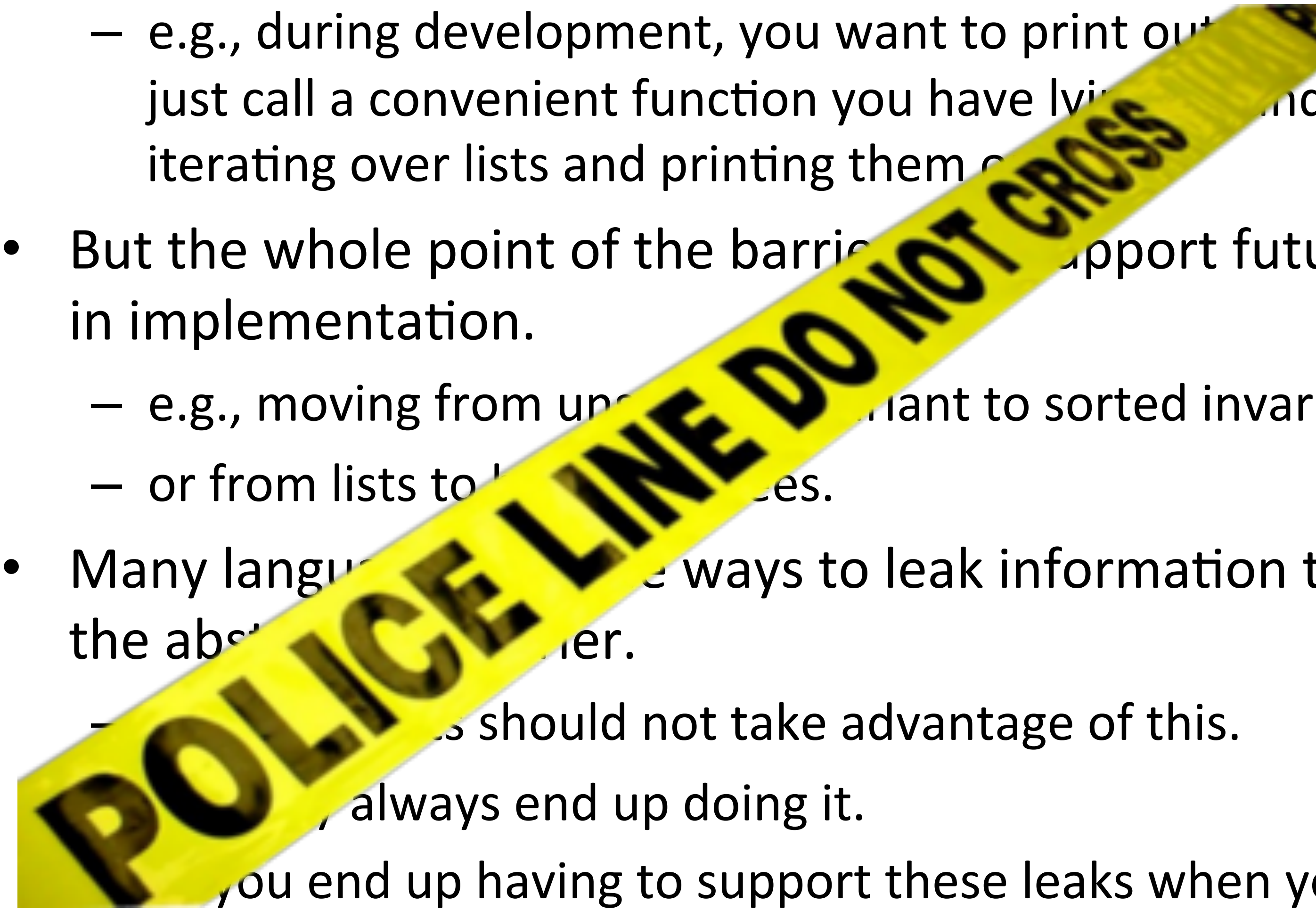
SUMMARY

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 ou t, so you just call a convenient function you have lyi nd for iterating over lists and printing them o
- But the whole point of the barri pport future change in implementation.
  - e.g., moving from un iant to sorted invariant.
  - or from lists to es.
- Many langu e ways to leak information through the abs ier.
  - s 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.



Key Points

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

END