# OCaml Modules

## Part 3: Functors

Speaker: David Walker

COS 326

Princeton University

# 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.

# Polymorphic Stacks

```
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 po
    val to
  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 Stack.t

  end
```

```
module BreadthFirstSearch : SEARCHER =
  struct
    type to_do : Graph.node Queue.t

  end
```

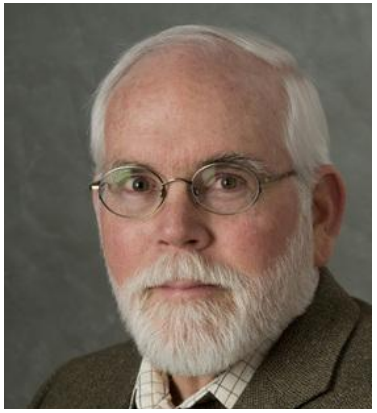
Still repeated code!

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.

# 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.

- Matrix elements may be ints or floats or complex ...
- And the elements still have a collection of operations on them:
  - addition, multiplication, zero element, etc.

What we'll see:

- RING: a signature for matrix elements
- MATRIX: a signature for 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 Type of IntMatrix

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

module IntMatrix = DenseMatrix(IntRing)
```

What is the signature of IntMatrix?

# The Type of IntMatrix

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

module IntMatrix = DenseMatrix(IntRing)
```

What is the signature of IntMatrix?

It depends on both the signatures of DenseMatrix and of it's argument IntRing

# The Type of IntMatrix

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

module IntMatrix = DenseMatrix(IntRing)
```

```
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
```

+

```
type elt = R.t
```

# The Type of IntMatrix

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

module IntMatrix = DenseMatrix(IntRing)
```

```
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
```

+

```
type elt = R.t
```

Recall:

```
module IntRing =
  struct
    type t = int
    let zero = 0
    ...
end
```

# The Type of IntMatrix

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

module IntMatrix = DenseMatrix(IntRing)
```

```
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
```

+

```
type elt = R.t
```

=

```
module type MATRIX =
  sig
    type elt = int
    type matrix
    ...
end
```

```
module IntRing =
  struct
    type t = int
    ...
  end
```

# 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

# 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

```
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!

nonexistent

# The DenseMatrix Functor

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

end
```

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!

nonexistent

```
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)
```

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 = ...
  type matrix = ...
  let matrix_of_list = ...
  let add m1 m2 = ...
  let mul m1 m2 = ...
end
```

To define a functor, just write down a module as its body.

That module has to match the result signature (MATRIX).

This module may refer to the functor arguments, like R.

```
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

Can refer to functor argument *values* or argument *types*!

# 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:

- 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.

# 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 sub-type 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.

# Summary

Functors allow code reuse

- commonly used to implement collection data structures
  - eg: sets, graphs, hash tables, etc
- the module parameter includes operations required by the body
  - to implement collections, we often need (in)equality or hashing

Sharing constraints

- allow the type of the functor result to depend upon its input rather than being opaque
- eg: a matrix has elements of the same type as input to its functor

Signature subtyping

- allows more reuse of modules at different signatures
- including one signature in another ensures the subtyping relation holds and can be useful as code evolves

# Exercise

Implement (or sketch the implementation of) a searcher functor that is parameterized by a container.

The searcher functor should do a traversal of a graph and print out the list of nodes it encounters in the order it encounters them. Bonus: parameterize the searcher functor by the graph representation as well as the ordering.

```
module type CONTAINER = sig ... end

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

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

```
module BreadthFirstSearch : SEARCHER =
  struct
    type to_do : Graph.node Queue.t

  end
```