

OCaml Modules

Part 4: Module Evaluation

Speaker: David Walker

COS 326

Princeton University



Last Time --> This Time

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

```
module FloatRing =  
  struct  
    type t = float  
    let zero = 0.0
```

```
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
```

We know how to define modules and functors.

How do we execute them?



Evaluating the contents of a module

A structure is a series of declarations:

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

let x = e

← evaluate the expression e
bind the value to x

- Remember, this kind of let: let f x = e
- Is the same as: let f = fun x -> e
- The right-hand side is already a value (fun x -> e) so evaluation is immediately "done"

How does one evaluate an entire structure?

- evaluate each declaration in order from first to last



Evaluating the contents of a module

main.ml

```
let x = 326
```

```
let main () =
```

```
  Printf.printf "Hello COS %d\n" x
```

```
let foo =
```

```
  Printf.printf "Byeeee!\n"
```


```
let _ =
```

```
  main ()
```



Evaluating the contents of a module

main.ml



```
let x = 326

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

let foo =
  Printf.printf "Byeee!\n"

let _ =
  main ()
```

Step 1:

evaluate the 1st declaration

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



Evaluating the contents of a module

main.ml

```
let x = 326  
let main () =  
  Printf.printf "Hello COS %d\n" x  
  
let foo =  
  Printf.printf "Byeeee!\n"  
  
let _ =  
  main ()
```

Step 2:

evaluate the 2nd declaration
this is slightly trickier:

let main () = ...

really declares a function.
It's equivalent to:

let main = fun () -> ...

“**fun () -> ...**” is already
a value, like 326.

So there's nothing to do 

Evaluating the contents of a module

main.ml

```
let x = 326

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

let foo =
  Printf.printf "Byeeee!\n"

let _ =
  main ()
```

Step 3:

evaluate the 3rd declaration

let foo = ...

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

the resulting value is () which is bound to foo



Evaluating the contents of a module

main.ml

```
let x = 326

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

let foo =
  Printf.printf "Byeee!\n"

let _ =
  main ()
```

Step 4:

evaluate the 4th declaration

let _ = ...

evaluation main ()
causes another effect.

“Hello ...” is printed

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



A Variation

main.ml

```
let x = 326

let main =
  (fun () ->
    Printf.printf "Hello COS %d\n" x)

let foo =
  Printf.printf "Byeeee!\n"

let _ =
  main ()
```

This evaluates exactly
the same way

We just replaced

`let main () = ...`

with the equivalent

`let main = fun () -> ...`



A Variation

main.ml

```
let x = 326

let main =
  Printf.printf "Hello COS %d\n" x;
  (fun () -> ())

let foo =
  Printf.printf "Byeeee!\n"

let _ =
  main ()
```

This rewrite does something different.

On the 2nd step, it prints because that's what evaluating this expression does:

```
Printf.printf "Hello COS %d\n" x;
(fun () -> ())
```

The result of the expression is:

```
fun () -> ()
```

which is bound to main.

This is a pretty silly functi



A Variation

main.ml

```
module C326 =  
struct  
  let x = 326  
  
  let main =  
    Printf.printf "Hello COS %d\n" x;  
    (fun () -> ())  
  
  let foo = Printf.printf "Byeee!\n"  
  
  let _ = main ()  
end  
  
let _ =  
  Printf.printf "Done\n"
```

Now what happens?



A Variation

main.ml

```
module C326 =  
struct  
  let x = 326  
  
  let main =  
    Printf.printf "Hello COS %d\n" x;  
    (fun () -> ())  
  
  let foo = Printf.printf "Byeeee!\n"  
  
  let _ = main ()  
end  
  
let done =  
  Printf.printf "Done\n"
```

Now what happens?

The entire file contains 2 decls:

- module C326 = ...
- let done = ...

We execute both of them in order.



A Variation

main.ml

```
module C326 =  
struct  
  let x = 326  
  
  let main =  
    Printf.printf "Hello COS %d\n" x;  
    (fun () -> ())  
  
  let foo = Printf.printf "Byeeee!\n"  
  
  let _ = main ()  
end  
  
let done =  
  Printf.printf "Done\n"
```

Now what happens?

The entire file contains 2 decls:

- module C326 = ...
- let done = ...

We execute both of them in order.

Executing the module declaration has the effect of executing every declaration within it in order.

Executing let done = ... is as before



A Variation

main.ml

```
module C326 =  
struct  
  exception Unimplemented  
  let x = raise Unimplemented  
  
  let main =  
    Printf.printf "Hello COS %d\n" x;  
    (fun () -> ())  
  
  let foo = Printf.printf "Byeeee!\n"  
  
  let _ = main ()  
end  
  
let done =  
  Printf.printf "Done\n"
```

Now what happens?



A Variation

main.ml

```
module C326 =  
struct  
  exception Unimplemented  
  let x = raise Unimplemented  
  
  let main =  
    Printf.printf "Hello COS %d\n" x;  
    (fun () -> ())  
  
  let foo = Printf.printf "Byeeee!\n"  
  
  let _ = main ()  
end  
  
let done =  
  Printf.printf "Done\n"
```

Now what happens?

The entire file contains 2 decls:

- module C326 = ...
- let done = ...

We execute both of them in order.

Executing the module declaration has the effect of executing every declaration within it in order.

The first declaration within it raises an exception which is not caught! That is the only result.



A Variation

main.ml

```
module type S =
```

```
sig
```

```
  type t = int
```

```
  val x : t
```

```
end
```

```
module F (M:S) : S =
```

```
struct
```

```
  let wow = Printf.printf "%d\n" M.x
```

```
  let t = M.t
```

```
  let x = M.x
```

```
end
```

```
let done = Printf.printf "Done\n"
```

Now what happens?

The entire file contains 2 decls:

- module type = ...
- module F (M:S) : S = ...
- let done = ...



A Variation

main.ml

```
module type S =
```

```
sig
```

```
  type t = int
```

```
  val x : t
```

```
end
```

```
module F (M:S) : S =
```

```
struct
```

```
  let wow = Printf.printf "%d\n" M.x
```

```
  let t = M.t
```

```
  let x = M.x
```

```
end
```

```
let done = Printf.printf "Done\n"
```

The signature declaration has no (run-time) effect.

The functor declaration is like declaring a function value.

The body of the functor is not executed until it is applied.

The functor is not applied here so M.x is not printed.

Only "Done\n" is printed.



A Variation

main.ml

```
module type S = sig ... end

module F (M:S) : S =
struct
  let wow = Printf.printf "%d\n" M.x
  let t = M.t
  let x = M.x
end

let module M1 = F (
  struct
    type t = int
    val x = 3
  end)

let done = Printf.printf "Done\n"
```

What happens now?



A Variation

main.ml

```
module type S = sig ... end

module F (M:S) : S =
struct
  let wow = Printf.printf "%d\n" M.x
  let t = M.t
  let x = M.x
end

let module M1 = F (
  struct
    type t = int
    val x = 3
  end)

let done = Printf.printf "Done\n"
```

What happens now?

When M1 is declared,
F is applied to an argument.

This creates a new structure and
its components are executed.

This has the effect of printing 3.



SUMMARY



Summary

Functors allow code reuse

- commonly used to implement collection data structures
 - eg: sets, graphs, hash tables, etc)
 - the module parameter includes operations required to implement a collection of objects
 - eg: equality or inequality or hashing

It is important to understand module evaluation semantics

- evaluate every declaration in order
- functions are values
 - a function has no effect until applied to an argument
- functors are module values!
 - a functor has no effect until applied to a module argument

