CSCI 2041: Modules and Functors

Chris Kauffman

Last Updated:
Mon Nov 19 22:02:42 CST 2018

Logistics

Reading

- OCaml System Manual: Ch 2
- Practical OCaml: Ch 13

Goals

- Modules
- Signatures
- Functors
- Birthday wishes

Assignment 4

- Shorter project
- Implement a persistent **tree**
- Used for Sets and Key-Val Maps
- Review Binary Search Trees
- Just a normal BST
- Due Sun 11/11

Module so Far

- Have been using **Modules** since nearly the very beginning
- Every source file `thing.ml` automatically defines a module (`Thing` in this case)
- Modules serve two roles in OCaml
  - As a **namespace** for functions, data, types
  - As a means to create **interfaces**
- Will discuss both these
- Will also discuss more specifics associated with modules including **functors**, functions of modules

Namespaces

- Much of programming deals with name management,
  - What thing is bound to a name at a given code position
  - How conflicts are resolved
- Top-level name management gives programming languages flavors programming language as bitter (C), sour (Java), sweet (Clojure), and salty (OCaml)
- A **namespace** is an abstract entity to group names for function, data, and type bindings

Language	**Namespace Approach**
C	No namespaces, cannot have multiple top-level names (e.g. functions)
Java	Classes are a namespace, packages further group classes
C++	Explicit namespaces a la `using namespace std;`
OCaml	Modules are namespaces; modules can be nested
Clojure	Explicit namespaces with advanced name management

Implicit and Explicit OCaml Modules

- Source file `thing.ml` automatically creates module `Thing`
- These are OCaml's **compilation units**:
  - A source file produces some associated compiled **object files**
  - Several object files may linked to create programs
  - "Object" here means machine code, not Classes/OO
- Explicit modules can be declared using the following `module/struct` syntax

```
module Mymodule = struct
  let val1 = ...;;
  let func1 a b = ...;;
end
```

- Explicitly defined modules are always **nested** within their housing source file module
- After compiling will only see `.cmo` files for source file modules, **not** for nested modules

Example: Topmod and Nestmod modules

```
(* topmod.ml: demonstrate both bindings at the top level in a module
   and syntax for a nested module. *)

let top_val1 = "hi";;                (* value bindings in Topmod *)
let top_val2 = 42;;
let top_func1 x y =                  (* function binding in Topmod *)
  x+y + x*y
;;

let same_name = "Co-exist";;         (* same as a later name *)

module Nestmod = struct              (* a nested module *)
  let nest_val1 = 1.23;;             (* value vinding in Nestmod *)
  let nest_val2 = true;;
  let nest_func1 a b =               (* function binding Nestmod *)
    (a*a,2*a*b,b*b)
  ;;

  let same_name = "Peacefully";;     (* same as previous name *)
end;;                                (* end of Nestmod module *)

let top_val3 = "tada";;              (* another value binding in Topmod *)
```

- Compile with `ocamlc -c topmod.ml`
- Only produces `topmod.cmo`, `Nestmod` is internal to this file

Exercise: Module Qualification

1. Give two ways to call the `printf` function in the `Printf` module
   - A "long" way and
   - A "short" way
2. How can one call `top_func1` or access `top_val3` in `topmod.ml` from another source file?
3. Speculate on how one can call `nest_func1` or access `nest_val1` from another source file.
4. Are the two bindings for `same_name` conflicting or can they be separately accessed?

```
(* topmod.ml: demonstrate both bi
   and syntax for a nested module

let top_val1 = "hi";;
let top_val2 = 42;;
let top_func1 x y =
  x+y + x*y
;;

let same_name = "Co-exist";;

module Nestmod = struct
  let nest_val1 = 1.23;;
  let nest_val2 = true;;
  let nest_func1 a b =
    (a*a,2*a*b,b*b)
  ;;

  let same_name = "Peacefully";;
end;;

let top_val3 = "tada";;
```

Answers: Module Qualification

1. Give two ways to call the `printf` function in the `Printf` module
   - A "long" way: `Printf.printf "Fully Qualified\n";`
   - A "short" way: `open Printf;; printf "Bare name\n";`
2. How can one call `top_func1` or access `top_val3` in `topmod.ml` from another source file?
   ```
   let i = Topmod.top_func1 7 2 in ...
   printf "%s\n" Topmod.top_val3;
   ```
3. Speculate on how one can call `nest_func1` or access `nest_val1` from another source file.
   ```
   let tup = Topmod.Nested.nest_func 7 2 in ...
   printf "%f\n" Topmod.Nested.nest_val1;
   ```
4. Are the two bindings for `same_name` conflicting or can they be separately accessed? *Not conflicting: separately accessible*
   ```
   # printf "%s %s" Topmod.same_name Topmod.Nested.same_name;
   Co-exist Peacefully
   ```

Module Qualification and Nesting

- Modules can always be reached via **qualifying** the binding with the module name: e.g. **dot syntax** like

```
List.head list
Printf.printf "hi\n"
Array.length arr
```

- Nested modules can be reached via further dots as in

```
Top.Nested.Deeply.some_value
```

Why would I nest a module?

- Occurs infrequently
  Kauffman wrote OCaml code for 3 years with no nesting
- Module nesting can be organizational for grouping in some libraries like Lacaml (nested modules for number types)
- Most often associated with **functors**: producing new modules by filling in a template (later)

Exercise: Module Review

1. What is a **namespace** and what does it do? Is it something that is only in OCaml?
2. How does OCaml deal with namespaces for things like libraries and functions in other source files?
3. How does one create a module in OCaml? What specific syntax is used?
4. We have been using modules since our first assignment: why didn't we need to use the syntax in #3 for our assignments?
5. How does one refer to a binding in another module? What does it mean to **qualify** a binding by its module?

Answers: Module Review

1. What is a **namespace** and what does it do? Is it something that is only in OCaml?
   A place to put name-binding values; all programming languages have them; good PLs provide namespace management
2. How does OCaml deal with namespaces for things like libraries and functions in other source files?
   Through its module system which allows name/value bindings to be retained in specific module so as to be accessible and not conflict with similarly named bindings
3. How does one create a module in OCaml? What specific syntax is used?

   ```
   module MyMod = struct
     let life = 42;;
     let liberty s = s^"work";;
     let happiness = ref true;;
   end
   ```

4. We have been using modules since our first assignment: why didn't we need to use the syntax in #3 for our assignments?
   Source code files automatically create modules as compilation units; source file `foobar.ml` creates module `Foobar`
5. How does one refer to a binding in another module? What does it mean to **qualify** a binding with its module?
   Use the module name and dot syntax like `MyMod.happiness`

Namespace Management

- Qualifying names for values with modules can be a drag

  `let x = Long.Module.Name.func y z in ...`
- All programming languages deal with such issues
- **Namespace management** techniques allow one to rename or otherwise mangle names for convenience
- OCaml has several of these
  1. Aliasing values/modules
  2. `open`-ing modules globally
  3. Local module opens

Value and Module Aliasing

- Single value or function can be **aliased**
- Refers to an existing entity via a new name

```
(* mod_alias.ml: show value and module aliasing syntax *)

(* Alias for specific values *)
let llen = List.length;;
let alen = Array.length;;

let an = alen [|1;2;3;4|];;
let ln = llen [5;6;7];;

(* Alias for a whole module *)
module L = List;;
module A = Array;;

let c = A.get [|"a";"b";"c"|] 2;;
let d = L.hd ["d";"e"];;
```

open-ing Modules

- To shorten frequent access, can use open to resolve **bare name** references to module bindings

```
open List;;
let lst = [1;2;3] in
printf "%d\n" (length lst) (* resolves to List.length *)
;;

open Array;;
let arr = [|1;2;|] in
printf "%d\n" (length arr) (* resolves to Array.length *)
;;
```

- Careful with `open` as it can get confusing which function is being used when multiple are open
- Clarification: **confusing for humans** - the type checker has no trouble with `open`, human debuggers may
- Good practice: Qualify all Module names even if `open` is used

Examples Using open with Topmod/Nestmod

```
open Printf;;

(* a main function *)
let _ =
  printf "%s %d\n" Topmod.top_val1 Topmod.top_val2;
  printf "%f %b\n" Topmod.Nestmod.nest_val1 Topmod.Nestmod.nest_val2;
  printf "%s %s\n" Topmod.same_name Topmod.Nestmod.same_name;
;;

(* an equivalent main function after opening Topmod *)
open Topmod;;
let _ =
  printf "%s %d\n" top_val1 top_val2;
  printf "%f %b\n" Nestmod.nest_val1 Nestmod.nest_val2;
  printf "%s %s\n" same_name Nestmod.same_name;
;;

(* an equivalent main function after opening Topmod.Nestmod *)

open Topmod.Nestmod;;
let _ =
  printf "%s %d\n" top_val1 top_val2;
  printf "%f %b\n" nest_val1 nest_val2;
  printf "%s %s\n" Topmod.same_name same_name; (* why qualified? *)
(* Nestmod opened most recently so qualify first same_name *)
;;
```

Local Module Opens

- Middle ground between open and qualifying every name: **local open** for a section of code
- Two equivalent syntax constructs for local opens

```
(* let/open/Mod local open *)
let _ =
  let lst1 = [1;2;3] in
  let lst2 = ["a";"b"] in
  let (len1,len2) =
    let open List in
    (length lst1, length lst2)
  in (* end of local open *)
  printf "%d %d %d\n" len1 len2;

  let open List in
  begin
    printf "%d\n" (hd lst1);
    printf "%s\n" (hd (tl lst2))
  end (* end of local open *)
;;
```

```
(* Mod.(..) local open *)
let _ =
  let lst1 = [1;2;3] in
  let lst2 = ["a";"b"] in
  let (len1,len2) =
    List.(length lst1, length lst2)
              (* end of local open ^ *)
  in
  printf "%d %d %d\n" len1 len2;

  List.(printf "%d\n" (hd lst1);
        printf "%s\n" (hd (tl lst2));
  ); (* end of local open *)
;;
```

Modules and Types

- Aside from functions, modules house `type` declarations
- These have identical qualification semantics to values
- Examples in `altp.ml` and `use_altp.ml`

```
(* altp.ml *)
type fruit =
  | Apple
  | Orange
  | Grapes of int;;

type 'a option =
  | None
  | Some of 'a;;
```

```
(* use_altp.ml *)
let a  = Altp.Apple;;
(* val a  : Altp.fruit *)

let g3 = Altp.Grapes 3;;
(* val g3 : Altp.fruit *)

let apopt = Altp.Some true;;
(* val apopt : boolean Altp.option *)

let spopt = Some true;;
(* val spopt : boolean option *)
```

Redefining Standard Types

- OCaml's standard modules like `Pervasives`, `List`, `etc.` follow the same rules as user-defined modules
- Allows one to redefine **everything** one wants via open

```
(* altp.ml *)
type fruit =
  | Apple
  | Orange
  | Grapes of int;;

type 'a list =
  | []
  | (::) of 'a * 'a list
;;

type 'a option =
  | None
  | Some of 'a
;;
```

```
(* use_altp.ml *)
open Altp;;
(* type constructors now resolve in
   Altp before standard versions *)

let o = Orange;;
(* val o : Altp.fruit *)

let aplst2 = 5::6::[];;
(* val aplst2 : int Altp.list *)

let apopt2 = Some true;
(* val apopt2 : boolean Altp.option *)
```

- Not for the faint of heart: new/old types incompatible
- Done to gain finer control/expanded functionality such as in Jane Street's Core Module, replaces **everything** standard

Quick Query

When compiling a `.ml` file like `topmod.ml`, have seen that several outputs can result

- `topmod.cmo` : compiled OCaml object file
- `a.out` : executable program (maybe)

What else have you frequently seen produced when compiling?

OCaml Interfaces

- Every compilation unit generates a compiled **interface**

```
> ocamlc topmod.ml

> file topmod.cmo
topmod.cmo: OCaml object file (.cmo) (Version 023)

> file topmod.cmi
topmod.cmi: OCaml interface file (.cmi) (Version 023)
```

- Interfaces contain the publicly accessible names in the source file module
- By default, everything is publicly accessible
- Can limit this through defining an `.mli` file explicitly

Example: glist.ml and glist.mli

```
> cat -n glist.ml
 1 (* make these "private" *)
 2 let thelist = ref ["first"];;
 3 let count = ref 1;;
 4
 5 (* make these "public" *)
 6 let add str =
 7   thelist := str :: !thelist;
 8   count := 1 + !count;
 9 ;;
10
11 let get_count () =
12   !count
13 ;;

> cat -n glist.mli
 1 (* The "private" bindings are
 2    not present so will not be
 3    available to other modules.
 4 *)
 5 val add       : string -> unit;;
 6 val get_count : unit -> int;;
```

```
> cat -n use_glist.ml
 1 let _ =
 2   Glist.add "goodbye";
 3   let count = Glist.get_count () in
 4   Printf.printf "count: %d\n" count;
 5 ;;

# compile
> ocamlc glist.mli glist.ml use_glist.ml

# success, run the program
> a.out
count: 2

> cat -n fail_glist.ml
 1 let _ =
 2   let h = List.hd Glist.thelist in
 3   Printf.printf "%s\n" h;
 4 ;;

# compile
> ocamlc glist.mli glist.ml fail_glist.ml
File "fail_glist.ml", line 9, chars 18-31:
Error: Unbound value Glist.thelist

# glist.mli interface file omits thelist
```

Module Signatures

- Module bindings are controlled by their **signature**
- Signatures are essentially **types for modules**
- `.mli` files state signatures for source-level modules
- Nested modules can state them explicitly with `sig/end` syntax
- Example: binding `str` is in module `SomeHidden` but not in its signature; `str` is not externally accessible

```
> cat -n sigexample.ml
 1 module SomeHidden : sig
 2   val x : string;;
 3   val f : int -> int -> string;;
 4 end
 5   =
 6 struct
 7   let x = "doo-da";;
 8   let str = "sum is:";;
 9   let f a b =
10     let c = a + b in
11     sprintf "%s %d" str c;;
12 end;;

> ocaml
# #use "sigexample.ml";;
module SomeHidden :
  sig
    val x : string
    val f : int -> int -> string
  end
# SomeHidden.x;;
- : string = "doo-da"
# SomeHidden.f 1 2;;
- : string = "sum is: 3"
# SomeHidden.str;;
Characters 0-14:
  SomeHidden.str;;
  ^^^^^^^^^^^^^^
```

More on Module Signatures

- Similar functionality to `public/private` access in OO langauges
- Allow hiding of internal helper functions that should NOT be used publicly
- Can name signatures for further use (lab exercise)
- Essential for understanding in discussions of **Functors** as well (next)

Type Safety vs Extensibility

- Build a fancy Data Structure (Extensible Array, Red-Black Tree, etc.)
- Elements of DS must support certain operations (equality, hash function, comparable)

Example

- 500 lines of code to define a Red-Black Tree to track unique Strings in `StringTree.xyz`
- To track unique Integers Could find-replace all `string` with `int` giving `IntTree.xyz`...

- Modern programming languages provide for
  - Type safety AND
  - Extensibility to new types
- Code **Parameterized on Data Types**
- Write DS once, usable for any type with supporting operations
  - Java Generics: `TreeSet<String>`, `TreeSet<Integer>`
  - C++ Templates: `Vector<string>`, `Vector<int>`
- OCaml also has this via **parameterized modules**

Parameterized Modules via Functors

Several ways to conceptualize this idea

1. A functor is a function when given a module, produces a module
2. A module can be left incomplete, a functor receives a module with information that completes it
3. A functor is a way to specialize a module by filling in its missing parts

The best way to begin understanding is to use existing functors like `Set.Make` and `Map.Make`

Using Library Functors: Set.Make

- Good way to understand basics of functors is to use existing ones
- Start with `Set.Make`: creates a module for manipulating sets of unique items
- Input module required: specify an element type and a comparison function
- Specified as a module with `Set.OrderedType` signature

```
module type OrderedType = sig
  type t;;
  val compare : t -> t -> int;;
end;;
```

```
(* create module for input to
   functor  *)
module IntElem = struct
  type t = int;;     (* elem type *)
  let compare x y = (* comparison *)
    x - y
  ;;
end;;

(* call the functor, get a module *)
module IntSet = Set.Make(IntElem);;

(* use the new module *)
let _ =
  let set1 = IntSet.empty in
  let set2 = IntSet.add 50 set1 in
  let set3 = IntSet.add 75 set2 in
```

Sets Come with Many Methods

`Set.Make` produces a module with a bunch of methods

- `add/remove/mem` for elements
- `union/intersection` for set ops
- `iter/map/fold` for higher-order operations

```
IntSet.iter (fun i->printf "%d " i) set4;
printf "\n";

let int_list = [22; 56; 99; 11;
                33; 44; 34; 89]
in
let set5 =
  let help set i =
    IntSet.add i set
  in
  List.fold_left help set4 int_list
in
IntSet.iter (fun i->printf "%d " i) set5;
printf "\n";
let sum = IntSet.fold (+) set5 0 in
printf "sum: %d\n" sum;
```

Many Types, Many Sets

- Set.Make creates modules for any needed types
- All co-exist, all benefit from same code base

```
(* interface strings with Set.Make *)
module StringElem = struct
  type t = string;;
  let compare = String.compare;;
end;;

(* call the functor, get a module *)
module StringSet =
  Set.Make(StringElem);;

let strset1 =
  StringSet.add "hello"
                StringSet.empty;;
```

```
(* a brand new record type *)
type strint = {
    str : string;
    num : int;
  };;

(* interface string with Set.Make *)
module StrintElem = struct
  type t = strint;;
  let compare x y =
    let diff =
      String.compare x.str y.str in
    if diff<>0 then
      diff
    else
      x.num - y.num
  ;;
end;;

(* call the functor, get a module *)
module StrintSet =
  Set.Make(StrintElem);;

let strintset1 =
  StrintSet.add {str="hi";num=5}
                StrintSet.empty;;
```

Exercise: Call the `Set.Make` Functor

- **Goal:** Create an `IntOptSet` module that can be used with `int option` elements as shown
- **Step 1:** Create an `IntOptElem` module with
  - Type t specified
  - compare function

```
module IntOptElem = ...
```

- **Step 2:** Call `Set.Make` to create the new module

```
module IntOptSet = ...
```

```
let _ =
  let empty = IntOptSet.empty in
  let ios1 =
    IntOptSet.add (Some 5) empty
  in
  let ios2 =
    IntOptSet.add (Some 1) ios1
  in
  let ios3 =
    IntOptSet.add None     ios2
  in
  let ios4 =
    IntOptSet.add (Some 9) ios3
  in
  let sum_some io sum =
    match io with
    | None -> sum
    | Some i -> sum + i
  in
  let sum =
    IntOptSet.fold sum_some ios4 0
  in
  Printf.printf "Sum is %d\n" sum;
;;
```

Answers: Call the `Set.Make` Functor

Step 1: Create an `IntOptElem` module with

```
(* specify interface to Set.Make *)
module IntOptElem = struct
  type t = int option;;
  let compare x y =
    match x,y with
    | None,None -> 0
    | None,_    -> -1
    | _,None    -> +1
    | Some x,Some y -> x-y
  ;;
end;;
```

Step 2: Call `Set.Make` to create the new module

```
(* call the functor *)
module IntOptSet =
  Set.Make(IntOptElem);;
```

```
let _ =
  let empty = IntOptSet.empty in
  let ios1 =
    IntOptSet.add (Some 5) empty
  in
  let ios2 =
    IntOptSet.add (Some 1) ios1
  in
  let ios3 =
    IntOptSet.add None     ios2
  in
  let ios4 =
    IntOptSet.add (Some 9) ios3
  in
  let sum_some io sum =
    match io with
    | None -> sum
    | Some i -> sum + i
  in
  let sum =
    IntOptSet.fold sum_some ios4 0
  in
  Printf.printf "Sum is %d\n" sum;
;;
```

Maps and `Map.Make`

- **Map** or **Dictionary**: an association of unique keys to values
- Keys are a set (all unique)
- Values may be redundant
- `Map.Make` functor allows creation of maps with a specific key type
- Requires same input module with `OrderedType` signature

```
(* interface for string key to Map.Make *)
module StringKey = struct
  type t = string;; (* elem type *)
  let compare =     (* comparison *)
    String.compare
  ;;
end;;

(* call the functor, get a module *)
module StringMap = Map.Make(StringKey);;

(* use the new module *)
let _ =
  (* string key, int value  *)
  let map1 = StringMap.empty in
  let map2 = StringMap.add "Morty" 80  map1 in
  let map3 = StringMap.add "Rick"  190 map2 in
  let map4 = StringMap.add "Jerry" 35  map3 in
  StringMap.iter (printf "%s->%d\n") map4;

  (* string key, string value *)
  let map2 =
    StringMap.add "Morty" "Smith"   map1 in
  let map3 =
    StringMap.add "Rick"  "Sanchez" map2 in
  let map4 =
    StringMap.add "Jerry" "Smith"   map3 in
  StringMap.iter (printf "%s->%s\n") map4;
;;
```

Basic Functor Syntax

Functions : Value Creation

Recall Function definition had two equivalent syntaxes

```
let funcname param1 param2 =
  ...
  ...
;;

let funcname =
  fun param1 param 2 ->
   ...
   ...
;;
```

Functors: Module Creation

Functors similarly have two equivalent syntaxes

```
module FunctorName (ParaMod : PSig) =
struct
  ...
end;;

module FunctorName =
  functor(ParaMod : PSig) ->
  struct
    ...
  end;;
```

- In both function and functor definitions, the parameter is available in the body for use
- Access parts of `ParaMod` with dot notation like `ParaMod.compare` or `ParaMod.t`

Functors Look like Modules

- Specify interface module signature
- Specify name for parameter module with signature
- Open a `struct`, bind values/types
- Importantly, access bindings in Parameter Module

```
module type INTERFACESIG = sig
  type sometype;;
  val some_func : ...;;
  val some_data : ...;;
end;;

module SomeFunctor(ParaMod : INTERFACESIG) =
struct
  type mytype = ... ParaMod.sometype ...;;

  let x = ...;;
  let func a b = ...;;
  let another_func =
    ... ParaMod.some_data ...
  ;;
  let myval = ... ParaMod.some_func ...;;
end;;
```

Exercise: A "Small" Example

- Examine `listset.ml`
- Has a `Make` functor which takes a parameter module
- Tracks sets of unique elements in a list
- Defines a type for the list which uses parameter module type
- Examine source of `listset.ml`
- Determine where the parameter module is used to within the resulting module
- Demonstrate how to use `Listset.Make` to create a set of unique integers

Answers: A "Small" Example

- Examine source of `listset.ml`
- Determine where the parameter module is used to within the resulting module
  1. In `type set` as the type for `data`
  2. The `compare` function within `add` function
  3. Within `to_string` uses the `elem_string` function
- Demonstrate how to use `Listset.Make` to create a set of unique integers

  From `use_listset.ml`:

```
module IntElem = struct                           (* interface module *)
  type elem_t = int;;
  let compare x y = x-y;;
  let elem_string = string_of_int;;
end;;

module IntListSet = Listset.Make(IntElem);;  (* call functor *)
```