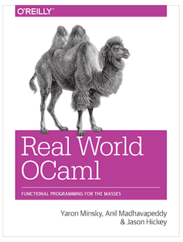


Buy in print and eBook.

Table of Contents

Login with GitHub to view and add comments

Chapter 10. First-Class Modules

You can think of OCaml as being broken up into two parts: a core language that is concerned with values and types, and a module language that is concerned with modules and module signatures. These sublanguages are stratified, in that modules can contain types and values, but ordinary values can't contain modules or module types. That means you can't do things like define a variable whose value is a module, or a function that takes a module as an argument.

OCaml provides a way around this stratification in the form of *first-class modules*. First-class modules are ordinary values that can be created from and converted back to regular modules.

First-class modules are a sophisticated technique, and you'll need to get comfortable with some advanced aspects of the language to use them effectively. But it's worth learning, because letting modules into the core language is quite powerful, increasing the range of what you can express and making it easier to build flexible and modular systems.

WORKING WITH FIRST-CLASS MODULES

We'll start out by covering the basic mechanics of first-class modules by working through some toy examples. We'll get to more realistic examples in the next section.

In that light, consider the following signature of a module with a single integer variable:

```
# module type X_int = sig val x : int end;;
 module type X_int = sig val x : int end
```

OCaml Utop ∗ fcm/main.topscript ∗ all code

We can also create a module that matches this signature:

```
# module Three : X_int = struct let x = 3 end;;
 module Three : X_int
# Three.x;;
 - : int = 3
```

OCaml Utop ∗ fcm/main.topscript , continued (part 1) ∗ all code

A first-class module is created by packaging up a module with a signature that it satisfies. This is done using the `module` keyword, using the following syntax:

```
(module <Module> : <Module_type>)
```

Syntax ∗ fcm/pack.syntax ∗ all code

So, we can convert `Three` into a first-class module as follows:

```
# let three = (module Three : X_int);;
 val three : (module X_int) = <module>
```

OCaml Utop ∗ fcm/main.topscript , continued (part 2) ∗ all code

The module type doesn't need to be part of the construction of a first-class module if it can be inferred. Thus, we can write:

```
# module Four = struct let x = 4 end;;
 module Four : sig val x : int end
# let numbers = [ three; (module Four) ];;
 val numbers : (module X_int) list = [<module>; <module>]
```

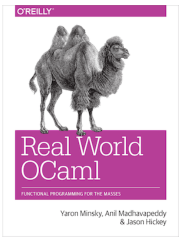
OCaml Utop ∗ fcm/main.topscript , continued (part 3) ∗ all code

We can also create a first-class module from an anonymous module:

```
# let numbers = [three; (module struct let x = 4 end)];;
 val numbers : (module X_int) list = [<module>; <module>]
```

OCaml Utop ∗ fcm/main.topscript , continued (part 4) ∗ all code

In order to access the contents of a first-class module, you need to unpack it into an ordinary module. This can be done using the `val` keyword, using this syntax:



```
(val <first_class_module> : <Module_type>)
```

Syntax ∗ fcm/unpack.syntax ∗ all code

And here's an example:

```
# module New_three = (val three : X_int) ;;
 module New_three : X_int
# New_three.x;;
 - : int = 3
```

OCaml Utop ∗ fcm/main.topscript , continued (part 5) ∗ all code

Equality of First-Class Module Types

The type of the first-class module, e.g., `(module X_int)`, is based on the fully qualified name of the signature that was used to construct it. A first-class module based on a signature with a different name, even if it is substantively the same signature, will result in a distinct type:

```
# module type Y_int = X_int;;
 module type Y_int = X_int
# let five = (module struct let x = 5 end : Y_int);;
 val five : (module Y_int) = <module>
# [three; five];;
 Characters 8-12:
 Error: This expression has type (module Y_int)
        but an expression was expected of type (module X_int)
```

OCaml Utop ∗ fcm/main.topscript , continued (part 6) ∗ all code

Even though their types as first-class modules are distinct, the underlying module types are compatible (indeed, identical), so we can unify the types by unpacking and repacking the module:

```
# [three; (module (val five))];;
 - : (module X_int) list = [<module>; <module>]
```

OCaml Utop ∗ fcm/main.topscript , continued (part 7) ∗ all code

The way in which type equality for first-class modules is determined can be confusing. One common and problematic case is that of creating an alias of a module type defined elsewhere. This is often done to improve readability and can happen both through an explicit declaration of a module type or implicitly through an `include` declaration. In both cases, this has the unintended side effect of making first-class modules built off the alias incompatible with those built off the original module type. To deal with this, we should be disciplined in how we refer to signatures when constructing first-class modules.

We can also write ordinary functions which consume and create first-class modules. The following shows the definition of two functions: `to_int`, which converts a `(module X_int)` into an `int`; and `plus`, which returns the sum of two `(module X_int)`:

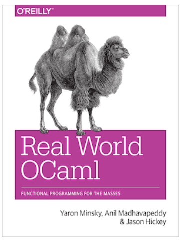
```
# let to_int m =
    let module M = (val m : X_int) in
    M.x
  ;;
 val to_int : (module X_int) -> int = <fun>
# let plus m1 m2 =
    (module struct
       let x = to_int m1 + to_int m2
     end : X_int)
  ;;
 val plus : (module X_int) -> (module X_int) -> (module X_int) = <fun>
```

OCaml Utop ∗ fcm/main.topscript , continued (part 8) ∗ all code

With these functions in hand, we can now work with values of type `(module X_int)` in a more natural style, taking advantage of the concision and simplicity of the core language:

```
# let six = plus three three;;
 val six : (module X_int) = <module>
# to_int (List.fold ~init:six ~f:plus [three;three]);;
 - : int = 12
```

OCaml Utop ∗ fcm/main.topscript , continued (part 9) ∗ all code



There are some useful syntactic shortcuts when dealing with first-class modules. One notable one is that you can do the conversion to an ordinary module within a pattern match. Thus, we can rewrite the `to_int` function as follows:

```
# let to_int (module M : X_int) = M.x ;;
 val to_int : (module X_int) -> int = <fun>
```

OCaml Utop ∗ fcm/main.topscript , continued (part 10) ∗ all code

First-class modules can contain types and functions in addition to simple values like `int`. Here's an interface that contains a type and a corresponding `bump` operation that takes a value of the type and produces a new one:

```
# module type Bumpable = sig
    type t
    val bump : t -> t
  end;;
 module type Bumpable = sig type t val bump : t -> t end
```

OCaml Utop ∗ fcm/main.topscript , continued (part 11) ∗ all code

We can create multiple instances of this module with different underlying types:

```
# module Int_bumper = struct
    type t = int
    let bump n = n + 1
  end;;
 module Int_bumper : sig type t = int val bump : t -> t end
# module Float_bumper = struct
     type t = float
     let bump n = n +. 1.
  end;;
 module Float_bumper : sig type t = float val bump : t -> t end
```

OCaml Utop ∗ fcm/main.topscript , continued (part 12) ∗ all code

And we can convert these to first-class modules:

```
# let int_bumper = (module Int_bumper : Bumpable);;
 val int_bumper : (module Bumpable) = <module>
```

OCaml Utop ∗ fcm/main.topscript , continued (part 13) ∗ all code

But you can't do much with `int_bumper`, since `int_bumper` is fully abstract, so that we can no longer recover the fact that the type in question is `int`.

```
# let (module Bumpable) = int_bumper in Bumpable.bump 3;;
 Characters 52-53:
 Error: This expression has type int but an expression was expected of type
          Bumpable.t
```

OCaml Utop ∗ fcm/main.topscript , continued (part 14) ∗ all code

To make `int_bumper` usable, we need to expose the type, which we can do as follows:

```
# let int_bumper = (module Int_bumper : Bumpable with type t = int);;
 val int_bumper : (module Bumpable with type t = int) = <module>
# let float_bumper = (module Float_bumper : Bumpable with type t = float);;
 val float_bumper : (module Bumpable with type t = float) = <module>
```

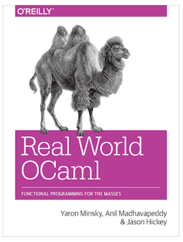
OCaml Utop ∗ fcm/main.topscript , continued (part 15) ∗ all code

The sharing constraints we've added above make the resulting first-class modules polymorphic in the type `t`. As a result, we can now use these values on values of the matching type:

```
# let (module Bumpable) = int_bumper in Bumpable.bump 3;;
 - : int = 4
# let (module Bumpable) = float_bumper in Bumpable.bump 3.5;;
 - : float = 4.5
```

OCaml Utop ∗ fcm/main.topscript , continued (part 16) ∗ all code

We can also write functions that use such first-class modules polymorphically. The following function takes two arguments: a `Bumpable` module and a list of elements of the same type as the type `t` of the module:



Buy in print and eBook.

Login with GitHub to view and add comments

```
# let bump_list
      (type a)
      (module B : Bumpable with type t = a)
      (l: a list)
    =
    List.map ~f:B.bump l
  ;;
 val bump_list : (module Bumpable with type t = 'a) -> 'a list -> 'a list =
   <fun>
```

OCaml Utop ∗ fcm/main.topscript , continued (part 17) ∗ all code

Here, we used a feature of OCaml that hasn't come up before: a *locally abstract type*. For any function, you can declare a pseudoparameter of the form `(type a)` for any type name `a` which introduces a fresh type. This type acts like an abstract type within the context of the function. In the example above, the locally abstract type was used as part of a sharing constraint that ties the type `B.t` with the type of the elements of the list passed in.

The resulting function is polymorphic in both the type of the list element and the type `Bumpable.t`. We can see this function in action:

```
# bump_list int_bumper [1;2;3];;
 - : int list = [2; 3; 4]
# bump_list float_bumper [1.5;2.5;3.5];;
 - : float list = [2.5; 3.5; 4.5]
```

OCaml Utop ∗ fcm/main.topscript , continued (part 18) ∗ all code

Polymorphic first-class modules are important because they allow you to connect the types associated with a first-class module to the types of other values you're working with.

More on Locally Abstract Types

One of the key properties of locally abstract types is that they're dealt with as abstract types in the function they're defined within, but are polymorphic from the outside. Consider the following example:

```
# let wrap_in_list (type a) (x:a) = [x];;
 val wrap_in_list : 'a -> 'a list = <fun>
```

OCaml Utop ∗ fcm/main.topscript , continued (part 19) ∗ all code

This compiles successfully because the type `a` is used in a way that is compatible with it being abstract, but the type of the function that is inferred is polymorphic.

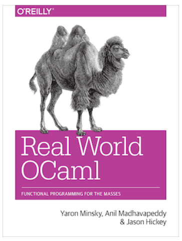
If, on the other hand, we try to use the type `a` as equivalent to some concrete type, say, `int`, then the compiler will complain:

```
# let wrap_int_in_list (type a) (x:a) = x + x;;
 Characters 38-39:
 Error: This expression has type a but an expression was expected of type
```

OCaml Utop ∗ fcm/main.topscript , continued (part 20) ∗ all code

One common use of locally abstract types is to create a new type that can be used in constructing a module. Here's an example of doing this to create a new first-class module:

```
# module type Comparable = sig
    type t
    val compare : t -> t -> int
  end ;;
 module type Comparable = sig type t val compare : t -> t -> int end
# let create_comparable (type a) compare =
    (module struct
       type t = a
       let compare = compare
     end : Comparable with type t = a)
  ;;
 val create_comparable :
   ('a -> 'a -> int) -> (module Comparable with type t = 'a) = <fun>
# create_comparable Int.compare;;
 - : (module Comparable with type t = int) = <module>
```



Buy in print and eBook.

Table of Contents

Login with GitHub to view and add comments

```
# create_comparable Float.compare;;
 - : (module Comparable with type t = float) = <module>
```

OCaml Utop ∗ fcm/main.topscript , continued (part 21) ∗ all code

Here, what we effectively do is capture a polymorphic type and export it as a concrete type within a module.

This technique is useful beyond first-class modules. For example, we can use the same approach to construct a local module to be fed to a functor.

EXAMPLE: A QUERY-HANDLING FRAMEWORK

Now let's look at first-class modules in the context of a more complete and realistic example. In particular, consider the following signature for a module that implements a system for responding to user-generated queries.

```
# module type Query_handler = sig

    (** Configuration for a query handler.  Note that this can be
        converted to and from an s-expression *)
    type config with sexp

    (** The name of the query-handling service *)
    val name : string

    (** The state of the query handler *)
    type t

    (** Creates a new query handler from a config *)
    val create : config -> t

    (** Evaluate a given query, where both input and output are
        s-expressions *)
    val eval : t -> Sexp.t -> Sexp.t Or_error.t
  end;;
 module type Query_handler =
   sig
     type config
     val name : string
     type t
     val create : config -> t
     val eval : t -> Sexp.t -> Sexp.t Or_error.t
     val config_of_sexp : Sexp.t -> config
     val sexp_of_config : config -> Sexp.t
   end
```

OCaml Utop ∗ fcm/query_handler.topscript ∗ all code

Here, we used s-expressions as the format for queries and responses, as well as the configuration for the query handler. S-expressions are a simple, flexible, and human-readable serialization format commonly used in Core. For now, it's enough to think of them as balanced parenthetical expressions whose atomic values are strings, e.g., `(this (is an) (s expression))`.

In addition, we use the Sexplib syntax extension which extends OCaml by adding the `with sexp` declaration. When attached to a type in a signature, `with sexp` adds declarations of s-expression converters, for example:

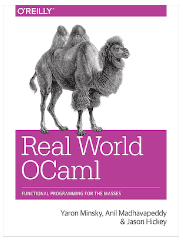
```
# module type M = sig type t with sexp end;;
 module type M =
   sig type t val t_of_sexp : Sexp.t -> t val sexp_of_t : t -> Sexp.t end
```

OCaml Utop ∗ fcm/query_handler.topscript , continued (part 1) ∗ all code

In a module, `with sexp` adds the implementation of those functions. Thus, we can write:

```
# type u = { a: int; b: float } with sexp;;
 type u = { a : int; b : float; }
 val u_of_sexp : Sexp.t -> u = <fun>
 val sexp_of_u : u -> Sexp.t = <fun>
# sexp_of_u {a=3;b=7.};;
 - : Sexp.t = ((a 3) (b 7))
# u_of_sexp (Sexp.of_string "((a 43) (b 3.4))");;
 - : u = {a = 43; b = 3.4}
```

OCaml Utop ∗ fcm/query_handler.topscript , continued (part 2) ∗ all code



Buy in print and eBook.

Table of Contents

Login with GitHub to view and add comments

This is all described in more detail in Chapter 17, *Data Serialization with S-Expressions*.

Implementing a Query Handler

Let's look at some examples of query handlers that satisfy the `Query_handler` interface. The first example is a handler that produces unique integer IDs. It works by keeping an internal counter which it bumps every time it produces a new value. The input to the query in this case is just the trivial s-expression `()`, otherwise known as `Sexp.unit`:

```
# module Unique = struct
    type config = int with sexp
    type t = { mutable next_id: int }

    let name = "unique"
    let create start_at = { next_id = start_at }

    let eval t sexp =
      match Or_error.try_with (fun () -> unit_of_sexp sexp) with
      | Error _ as err -> err
      | Ok () ->
        let response = Ok (Int.sexp_of_t t.next_id) in
        t.next_id <- t.next_id + 1;
        response
  end;;
 module Unique :
   sig
     type config = int
     val config_of_sexp : Sexp.t -> config
     val sexp_of_config : config -> Sexp.t
     type t = { mutable next_id : config; }
     val name : string
     val create : config -> t
     val eval : t -> Sexp.t -> (Sexp.t, Error.t) Result.t
   end
```

OCaml Utop ∗ fcm/query_handler.topscript , continued (part 3) ∗ all code

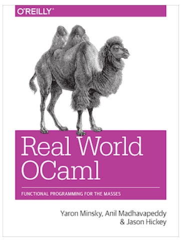
We can use this module to create an instance of the `Unique` query handler and interact with it directly:

```
# let unique = Unique.create 0;;
 val unique : Unique.t = {Unique.next_id = 0}
# Unique.eval unique Sexp.unit;;
 - : (Sexp.t, Error.t) Result.t = Ok 0
# Unique.eval unique Sexp.unit;;
 - : (Sexp.t, Error.t) Result.t = Ok 1
```

OCaml Utop ∗ fcm/query_handler.topscript , continued (part 4) ∗ all code

Here's another example: a query handler that does directory listings. Here, the config is the default directory that relative paths are interpreted within:

```
# module List_dir = struct
    type config = string with sexp
    type t = { cwd: string }

    (** [is_abs p] Returns true if [p] is an absolute path  *)
    let is_abs p =
      String.length p > 0 && p.[0] = '/'

    let name = "ls"
    let create cwd = { cwd }

    let eval t sexp =
      match Or_error.try_with (fun () -> string_of_sexp sexp) with
      | Error _ as err -> err
      | Ok dir ->
        let dir =
          if is_abs dir then dir
          else Filename.concat t.cwd dir
        in
        Ok (Array.sexp_of_t String.sexp_of_t (Sys.readdir dir))
  end;;
 module List_dir :
   sig
     type config = string
     val config_of_sexp : Sexp.t -> config
     val sexp_of_config : config -> Sexp.t
     type t = { cwd : config; }
```



Buy in print and eBook.

Login with GitHub to view and add comments

```
      val is_abs : config -> bool
      val name : config
      val create : config -> t
      val eval : t -> Sexp.t -> (Sexp.t, Error.t) Result.t
    end
```

OCaml Utop ∗ fcm/query_handler.topscript , continued (part 5) ∗ all code

Again, we can create an instance of this query handler and interact with it directly:

```
# let list_dir = List_dir.create "/var";;
 val list_dir : List_dir.t = {List_dir.cwd = "/var"}
# List_dir.eval list_dir (sexp_of_string ".");;
 - : (Sexp.t, Error.t) Result.t =
 Ok
  (agentx at audit backups db empty folders jabberd lib log mail msgs netboot
   networkd root rpc run rwho spool tmp vm yp)
# List_dir.eval list_dir (sexp_of_string "yp");;
 - : (Sexp.t, Error.t) Result.t = Ok (binding binding~orig)
```

OCaml Utop ∗ fcm/query_handler.topscript , continued (part 6) ∗ all code

Dispatching to Multiple Query Handlers

Now, what if we want to dispatch queries to any of an arbitrary collection of handlers? Ideally, we'd just like to pass in the handlers as a simple data structure like a list. This is awkward to do with modules and functors alone, but it's quite natural with first-class modules. The first thing we'll need to do is create a signature that combines a `Query_handler` module with an instantiated query handler:

```
# module type Query_handler_instance = sig
    module Query_handler : Query_handler
    val this : Query_handler.t
  end;;
 module type Query_handler_instance =
   sig module Query_handler : Query_handler val this : Query_handler.t end
```

OCaml Utop ∗ fcm/query_handler.topscript , continued (part 7) ∗ all code

With this signature, we can create a first-class module that encompasses both an instance of the query and the matching operations for working with that query.

We can create an instance as follows:

```
# let unique_instance =
    (module struct
       module Query_handler = Unique
       let this = Unique.create 0
     end : Query_handler_instance);;
 val unique_instance : (module Query_handler_instance) = <module>
```

OCaml Utop ∗ fcm/query_handler.topscript , continued (part 8) ∗ all code

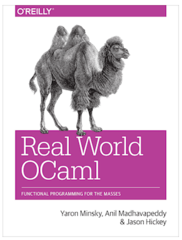
Constructing instances in this way is a little verbose, but we can write a function that eliminates most of this boilerplate. Note that we are again making use of a locally abstract type:

```
# let build_instance
        (type a)
        (module Q : Query_handler with type config = a)
        config
    =
    (module struct
       module Query_handler = Q
       let this = Q.create config
     end : Query_handler_instance)
  ;;
 val build_instance :
   (module Query_handler with type config = 'a) ->
   'a -> (module Query_handler_instance) = <fun>
```

OCaml Utop ∗ fcm/query_handler.topscript , continued (part 9) ∗ all code

Using `build_instance`, constructing a new instance becomes a one-liner:

```
# let unique_instance = build_instance (module Unique) 0;;
 val unique_instance : (module Query_handler_instance) = <module>
# let list_dir_instance = build_instance (module List_dir)  "/var";;
 val list_dir_instance : (module Query_handler_instance) = <module>
```


Buy in print and eBook.

Login with GitHub to view and add comments

OCaml Utop ∗ fcm/query_handler.topscript , continued (part 10) ∗ all code

We can now write code that lets you dispatch queries to one of a list of query handler instances. We assume that the shape of the query is as follows:

```
(query-name query)
```

Scheme ∗ fcm/query-syntax.scm ∗ all code

where *query-name* is the name used to determine which query handler to dispatch the query to, and *query* is the body of the query.

The first thing we'll need is a function that takes a list of query handler instances and constructs a dispatch table from it:

```
# let build_dispatch_table handlers =
    let table = String.Table.create () in
    List.iter handlers
      ~f:(fun ((module I : Query_handler_instance) as instance) ->
        Hashtbl.replace table ~key:I.Query_handler.name ~data:instance);
    table
  ;;
 val build_dispatch_table :
   (module Query_handler_instance) list ->
   (module Query_handler_instance) String.Table.t = <fun>
```

OCaml Utop ∗ fcm/query_handler.topscript , continued (part 11) ∗ all code

Now, we need a function that dispatches to a handler using a dispatch table:

```
# let dispatch dispatch_table name_and_query =
    match name_and_query with
    | Sexp.List [Sexp.Atom name; query] ->
      begin match Hashtbl.find dispatch_table name with
      | None ->
        Or_error.error "Could not find matching handler"
          name String.sexp_of_t
      | Some (module I : Query_handler_instance) ->
        I.Query_handler.eval I.this query
      end
    | _ ->
      Or_error.error_string "malformed query"
  ;;
 val dispatch :
   (string, (module Query_handler_instance)) Hashtbl.t ->
   Sexp.t -> Sexp.t Or_error.t = <fun>
```

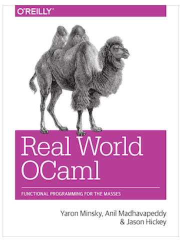
OCaml Utop ∗ fcm/query_handler.topscript , continued (part 12) ∗ all code

This function interacts with an instance by unpacking it into a module `I` and then using the query handler instance (`I.this`) in concert with the associated module (`I.Query_handler`).

The bundling together of the module and the value is in many ways reminiscent of object-oriented languages. One key difference, is that first-class modules allow you to package up more than just functions or methods. As we've seen, you can also include types and even modules. We've only used it in a small way here, but this extra power allows you to build more sophisticated components that involve multiple interdependent types and values.

Now let's turn this into a complete, running example by adding a command-line interface:

```
# let rec cli dispatch_table =
    printf ">>> %!";
    let result =
      match In_channel.input_line stdin with
      | None -> `Stop
      | Some line ->
        match Or_error.try_with (fun () -> Sexp.of_string line) with
        | Error e -> `Continue (Error.to_string_hum e)
        | Ok (Sexp.Atom "quit") -> `Stop
        | Ok query ->
          begin match dispatch dispatch_table query with
          | Error e -> `Continue (Error.to_string_hum e)
          | Ok s    -> `Continue (Sexp.to_string_hum s)
          end;
    in
    match result with
    | `Stop -> ()
    | `Continue msg ->
```



```
      printf "%s\n%!" msg;
      cli dispatch_table
  ;;
val cli : (string, (module Query_handler_instance)) Hashtbl.t -> unit = <fun>
```
OCaml Utop ∗ fcm/query_handler.topscript , continued (part 13) ∗ all code

We can most effectively run this command-line interface from a standalone program, which we can do by putting the above code in a file along with following command to launch the interface:

```
let () =
  cli (build_dispatch_table [unique_instance; list_dir_instance])
```
OCaml ∗ fcm/query_handler.ml , continued (part 1) ∗ all code

Here's an example of a session with this program:

```
$ ./query_handler.byte
>>> (unique ())
0
>>> (unique ())
1
>>> (ls .)
(agentx at audit backups db empty folders jabberd lib log mail msgs named
 netboot pgsql_socket_alt root rpc run rwho spool tmp vm yp)
>>> (ls vm)
(sleepimage swapfile0 swapfile1 swapfile2 swapfile3 swapfile4 swapfile5
 swapfile6)
```
OCaml Utop ∗ fcm/query_example.rawscript ∗ all code

Loading and Unloading Query Handlers

One of the advantages of first-class modules is that they afford a great deal of dynamism and flexibility. For example, it's a fairly simple matter to change our design to allow query handlers to be loaded and unloaded at runtime.

We'll do this by creating a query handler whose job is to control the set of active query handlers. The module in question will be called `Loader`, and its configuration is a list of known `Query_handler` modules. Here are the basic types:

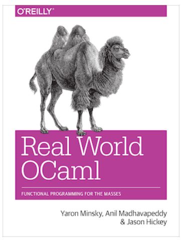
```
module Loader = struct
  type config = (module Query_handler) list sexp_opaque
  with sexp

  type t = { known  : (module Query_handler)          String.Table.t
           ; active : (module Query_handler_instance) String.Table.t
           }

  let name = "loader"
```
OCaml ∗ fcm/query_handler_core.ml , continued (part 1) ∗ all code

Note that a `Loader.t` has two tables: one containing the known query handler modules, and one containing the active query handler instances. The `Loader.t` will be responsible for creating new instances and adding them to the table, as well as for removing instances, all in response to user queries.

Next, we'll need a function for creating a `Loader.t`. This function requires the list of known query handler modules. Note that the table of active modules starts out as empty:

```
let create known_list =
    let active = String.Table.create () in
    let known  = String.Table.create () in
    List.iter known_list
      ~f:(fun ((module Q : Query_handler) as q) ->
        Hashtbl.replace known ~key:Q.name ~data:q);
    { known; active }
```
OCaml ∗ fcm/query_handler_core.ml , continued (part 2) ∗ all code

Now we'll start writing out the functions for manipulating the table of active query handlers. We'll start with the function for loading an instance. Note that it takes as an argument both the name of the query handler and the configuration for instantiating that handler in the form of an s-expression. These are used for creating a first-class module of type `(module Query_handler_instance)`, which is then added to the active table:



Buy in print and eBook.

Login with GitHub to view and add comments

```
let load t handler_name config =
    if Hashtbl.mem t.active handler_name then
      Or_error.error "Can't re-register an active handler"
        handler_name String.sexp_of_t
    else
      match Hashtbl.find t.known handler_name with
      | None ->
        Or_error.error "Unknown handler" handler_name String.sexp_of_t
      | Some (module Q : Query_handler) ->
        let instance =
          (module struct
             module Query_handler = Q
             let this = Q.create (Q.config_of_sexp config)
           end : Query_handler_instance)
        in
        Hashtbl.replace t.active ~key:handler_name ~data:instance;
        Ok Sexp.unit
```

OCaml ∗ fcm/query_handler_core.ml , continued (part 3) ∗ all code

Since the `load` function will refuse to `load` an already active handler, we also need the ability to unload a handler. Note that the handler explicitly refuses to unload itself:

```
let unload t handler_name =
    if not (Hashtbl.mem t.active handler_name) then
      Or_error.error "Handler not active" handler_name String.sexp_of_t
    else if handler_name = name then
      Or_error.error_string "It's unwise to unload yourself"
    else (
      Hashtbl.remove t.active handler_name;
      Ok Sexp.unit
    )
```

OCaml ∗ fcm/query_handler_core.ml , continued (part 4) ∗ all code

Finally, we need to implement the `eval` function, which will determine the query interface presented to the user. We'll do this by creating a variant type, and using the s-expression converter generated for that type to parse the query from the user:

```
type request =
    | Load of string * Sexp.t
    | Unload of string
    | Known_services
    | Active_services
  with sexp
```

OCaml ∗ fcm/query_handler_core.ml , continued (part 5) ∗ all code

The `eval` function itself is fairly straightforward, dispatching to the appropriate functions to respond to each type of query. Note that we write `<:sexp_of<string list>>` to autogenerate a function for converting a list of strings to an s-expression, as described in Chapter 17, *Data Serialization with S-Expressions*.

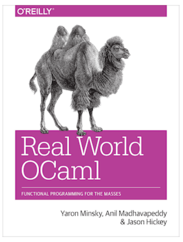
This function ends the definition of the `Loader` module:

```
let eval t sexp =
    match Or_error.try_with (fun () -> request_of_sexp sexp) with
    | Error _ as err -> err
    | Ok resp ->
      match resp with
      | Load (name,config) -> load   t name config
      | Unload name        -> unload t name
      | Known_services ->
        Ok (<:sexp_of<string list>> (Hashtbl.keys t.known))
      | Active_services ->
        Ok (<:sexp_of<string list>> (Hashtbl.keys t.active))
end
```

OCaml ∗ fcm/query_handler_core.ml , continued (part 6) ∗ all code

Finally, we can put this all together with the command-line interface. We first create an instance of the loader query handler and then add that instance to the loader's active table. We can then just launch the command-line interface, passing it the active table:

```
let () =
  let loader = Loader.create [(module Unique); (module List_dir)] in
  let loader_instance =
    (module struct
       module Query_handler = Loader
       let this = loader
```



Buy in print and eBook.

Login with GitHub to view and add comments

```
    end : Query_handler_instance)
  in
  Hashtbl.replace loader.Loader.active
    ~key:Loader.name ~data:loader_instance;
  cli loader.Loader.active
```

OCaml ∗ fcm/query_handler_loader.ml , continued (part 1) ∗ all code

Now build this into a command-line interface to experiment with it:

```
$ corebuild query_handler_loader.byte
```

Terminal ∗ fcm/build_query_handler_loader.out ∗ all code

The resulting command-line interface behaves much as you'd expect, starting out with no query handlers available but giving you the ability to load and unload them. Here's an example of it in action. As you can see, we start out with `loader` itself as the only active handler:

```
$ ./query_handler_loader.byte
>>> (loader known_services)
(ls unique)
>>> (loader active_services)
(loader)
```

Terminal ∗ fcm/loader_cli1.out ∗ all code

Any attempt to use an inactive query handler will fail:

```
>>> (ls .)
Could not find matching handler: ls
```

Terminal ∗ fcm/loader_cli2.out ∗ all code

But, we can load the `ls` handler with a config of our choice, at which point it will be available for use. And once we unload it, it will be unavailable yet again and could be reloaded with a different config:

```
>>> (loader (load ls /var))
()
>>> (ls /var)
(agentx at audit backups db empty folders jabberd lib log mail msgs named
 netboot pgsql_socket_alt root rpc run rwho spool tmp vm yp)
>>> (loader (unload ls))
()
>>> (ls /var)
Could not find matching handler: ls
```

Terminal ∗ fcm/loader_cli3.out ∗ all code

Notably, the loader can't be loaded (since it's not on the list of known handlers) and can't be unloaded either:

```
>>> (loader (unload loader))
It's unwise to unload yourself
```

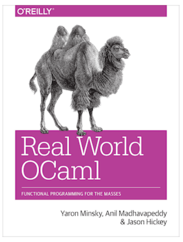
Terminal ∗ fcm/loader_cli4.out ∗ all code

Although we won't describe the details here, we can push this dynamism yet further using OCaml's dynamic linking facilities, which allow you to compile and link in new code to a running program. This can be automated using libraries like `ocaml_plugin`, which can be installed via OPAM, and which automates much of the workflow around setting up dynamic linking.

LIVING WITHOUT FIRST-CLASS MODULES

It's worth noting that most designs that can be done with first-class modules can be simulated without them, with some level of awkwardness. For example, we could rewrite our query handler example without first-class modules using the following types:

```
# type query_handler_instance = { name : string
                                 ; eval : Sexp.t -> Sexp.t Or_error.t
                                 }
  type query_handler = Sexp.t -> query_handler_instance
  ;;
 type query_handler_instance = {
   name : string;
```



Login with GitHub to view and add comments

```
   eval : Sexp.t -> Sexp.t Or_error.t;
 }
 type query_handler = Sexp.t -> query_handler_instance
```

OCaml Utop ∗ fcm/query_handler.topscript , continued (part 14) ∗ all code

The idea here is that we hide the true types of the objects in question behind the functions stored in the closure. Thus, we could put the `Unique` query handler into this framework as follows:

```
# let unique_handler config_sexp =
    let config = Unique.config_of_sexp config_sexp in
    let unique = Unique.create config in
    { name = Unique.name
    ; eval = (fun config -> Unique.eval unique config)
    }
  ;;
 val unique_handler : Sexp.t -> query_handler_instance = <fun>
```

OCaml Utop ∗ fcm/query_handler.topscript , continued (part 15) ∗ all code

For an example on this scale, the preceding approach is completely reasonable, and first-class modules are not really necessary. But the more functionality you need to hide away behind a set of closures, and the more complicated the relationships between the different types in question, the more awkward this approach becomes, and the better it is to use first-class modules.

< Previous Next >