12 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.

12.1 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.

12.1.1 Creating First-Class Modules

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

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

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
```

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

https://doi.org/10.1017/9781009129220.014 Published online by Cambridge University Press

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

We can convert `Three` into a first-class module as follows:

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

12.1.2 Inference and Anonymous Modules

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>]
```

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>]
```

12.1.3 Unpacking First-Class Modules

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>)
```

Here's an example:

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

12.1.4 Functions for Manipulating 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>
```

You can also unpack a first-class module with a pattern match, which lets us write `to_int` more concisely:

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

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
```

12.1.5 Richer First-Class Modules

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
```

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
```

And we can convert these to first-class modules:

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

12.1.6 Exposing types

You can't do much with `int_bumper` because it's fully abstract, so we can't take advantage of the fact that the type in question is `int`, which makes it impossible to construct or really do anything with values of type `Bumper.t`.

```
# let (module Bumper) = int_bumper in
  Bumper.bump 3;;
Line 2, characters 15-16:
```

```
Error: This expression has type int but an expression was expected of
    type
         Bumper.t
```

To make `int_bumper` usable, we need to expose that the type `Bumpable.t` is actually equal to `int`. Below we'll do that for `int_bumper`, and also provide the corresponding definition for `float_bumper`.

```
# 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>
```

The addition of the sharing constraint has exposed the type `t`, which lets us actually use the values within the module.

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

We can also use these 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:

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

In this example, `a` is a *locally abstract type*. For any function, you can declare a pseudoparameter of the form `(type a)` which introduces a fresh type named `a`. 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]
```

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>
```

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 if it were equivalent to some concrete type, say, `int`, then the compiler will complain.

```
# let double_int (type a) (x:a) = x + x;;
Line 1, characters 33-34:
Error: This expression has type a but an expression was expected of
    type int
```

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

12.2 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, we're going to implement a system for responding to user-generated queries.

This system will use *s-expressions* for formatting 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 `Base` and related libraries. 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))`. S-expressions are covered in more detail in Chapter 21 (Data Serialization with S-Expressions).

The following signature for a module that implements a system for responding to user-generated queries. Here, we use Base's `Sexp` module for handling s-expressions. Note that we could just as easily have used another serialization format, like JSON, as discussed in Chapter 19 (Handling JSON Data).

```
module type Query_handler = sig

  (** Configuration for a query handler *)
  type config

  val sexp_of_config : config -> Sexp.t
  val config_of_sexp : Sexp.t -> config

  (** 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;;
```

Implementing s-expression converters by hand is tedious and error-prone, but happily, we have an alternative. `ppx_sexp_conv` is a syntax extension which can be used to automatically generate s-expression converters based on their type definition. We'll enable `ppx_sexp_conv` by enabling `ppx_jane`, which brings in a larger family of syntax extensions.

```
# #require "ppx_jane";;
```

Here's an example of the extension in action. Note that we need the annotation `[@@deriving sexp]` to kick off the generation of the converters.

```
# type u = { a: int; b: float } [@@deriving 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 (Core.Sexp.of_string "((a 43) (b 3.4))");;
- : u = {a = 43; b = 3.4}
```

The same annotations can be attached within a signature to add the appropriate type signature.

```
# module type M = sig type t [@@deriving sexp] end;;
module type M =
sig type t val t_of_sexp : Sexp.t -> t val sexp_of_t : t -> Sexp.t
  end
```

12.2.1 Implementing a Query Handler

Now we can construct an example of a query handler that satisfies the `Query_handler` interface. We'll start with a handler that produces unique integer IDs, which works by keeping an internal counter that's bumped every time a new value is requested. 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 [@@deriving 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;;
```

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.List []);;
- : (Sexp.t, Error.t) result = Ok 0
# Unique.eval unique (Sexp.List []);;
- : (Sexp.t, Error.t) result = Ok 1
```

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 [@@deriving sexp]
  type t = { cwd: string }

  (** [is_abs p] Returns true if [p] is an absolute path  *)
  let is_abs p =
    String.length p > 0 && Char.(=) 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 Core.Filename.concat t.cwd dir
        in
        Ok (Array.sexp_of_t String.sexp_of_t (Core.Sys.readdir dir))
end;;
```

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 =
Ok
 (yp networkd install empty ma mail spool jabberd vm msgs audit root
    lib db
  at log folders netboot run rpc tmp backups agentx rwho)
# List_dir.eval list_dir (sexp_of_string "yp");;
- : (Sexp.t, Error.t) result = Ok (binding)
```

12.2.2 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
```

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>
```

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>
```

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>
```

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)
```

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

Next, we'll 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.Poly.t ->
  Sexp.t -> Sexp.t Or_error.t = <fun>
```

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.

We can turn this into a complete, running example by adding a command-line interface:

```
# open Stdio;;
# 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 () ->
          Core.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.Poly.t ->
    unit =
  <fun>
```

We'll run this command-line interface from a standalone program by putting the above code in a file, and adding the following to launch the interface.

```
let () =
  cli (build_dispatch_table [unique_instance; list_dir_instance])
```

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

```
$ dune exec -- ./query_handler.exe
>>> (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)
```

12.2.3 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]
  [@@deriving sexp]

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

  let name = "loader"
```

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.set known ~key:Q.name ~data:q);
    { known; active }
```

Now we can write 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:

```
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.set t.active ~key:handler_name ~data:instance;
         Ok Sexp.unit
```

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

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
  [@@deriving sexp]
```

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 21 (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
```

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

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.

```
$ dune exec -- ./query_handler_loader.exe
>>> (loader known_services)
(ls unique)
>>> (loader active_services)
(loader)
```

Any attempt to use an inactive query handler will fail:

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

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

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
```

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 takes care of much of the workflow around setting up dynamic linking.

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

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>
```

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.