The Module Language

Part I – Modules, Compilation, Applying Functors

OCaml PRO – Benjamin Canou

March 9-13 2015

Outline

Toplevel Modules

Source units as modules
Separate compilation
Interfaces
Interface compilation
Documentation

Source units as modules

Basics:

- Each file `momo.ml` defines a module named `Momo` .
- Its names can be accessed form other modules prefixed by `Momo.`
- The module relationship is a partial order (cycles are forbidden).

File `momo.ml` :

```
type t = { x : int ; y : int }
let origin = { x = 0 ; y = 0 }
let above pa pb = pa.x >= pb.x
```

File `main.ml` :

```
let o : Momo.t = Momo.origin
let p : Momo.t = { Momo.x = 2 ; Momo.y = 4 }
let () = assert (Momo.above p o)
```

Prefixes can be omitted by (very cautiously) using

- `open Momo` at toplevel
- `let open Momo in` in expressions
- `Momo.(...)` in expressions

Separate compilation

Multi-modular programs can be simply compiled in one invocation:

```
ocamlopt momo.ml main.ml -o momotest
```

Order is important !

Or modules can be compiled separately, and linked together:

```
ocamlopt -c momo.ml
ocamlopt -c main.ml
ocamlopt momo.cmx main.cmx -o momotest
```

A simple `Makefile` :

```
momotest: momo.cmx main.cmx
	ocamlopt $^ -o $@
%.cmx: %.ml
	ocamlopt -c $<
clean:
	-$(RM) -f *.cm* momotest
```

Interfaces

Each `.ml` source can be accompanied with a `.mli` interface.

Interfaces define:

- The types present in the source, with the same syntax.
- Each value (function or not) present in the source in the form:
  `val name : type`

File `momo.mli`:

```
type t = { x : int ; y : int }
val origin : t
val above : t -> t -> bool
```

Interface compilation

Interfaces are compiled to `cmi` files.

- When checking a reference to an external module, `ocamlopt` will look for its `cmi` first.
- When an interface is present, `ocamlopt` will require it to be compiled before the source.

The `Makefile` updated:

```
momotest: momo.cmx main.cmx
        ocamlopt $^ -o $@
%.cmx: %.ml
        ocamlopt -c $<
%.cmx: %.mli
        ocamlopt -c $<
momo.cmx: momo.cmi
clean:
        -rm -f *.cm* momotest
```

Com more complex dependencies, there is `ocamldep` .

Documentation

The `ocamldoc` tool defines a special syntax for comments:

- Documentation comments are `(** enclosed like this *)`.
- Associated to the nearest element (before or after, without linebreak).
- The first of the file describes the module.

File `momo.mli` documented, `ocamldoc -html momo.mli` to see the result.

```
(** The world of Momo *)

(** The type of points in the world of Momo. *)
type t =
  { x : int (** How far on the horizon *) ;
    y : int (** How high in the sky *) }

(** The location of the mother of Momo. *)
val origin : t

(** To tell who is taller. *)
val above : t -> t -> bool
```

Abstraction

Structure rewriting & hiding
Abstract types
Private types

The interface must not follow the implementation completely.

- To hide internal auxiliary elements.
- To hide functions that may break internal invariants.
- To help providing a more structured documentation.

For this, it is possible to:

- Hide types and values.
- Reorder everything.
- Use type aliases instead of primitive types.

Rewriting is allowe as long as

- The interface is still consistent with the implementation.
  (An integer in the implementation cannot be exported with type float.)
- The interface is self consistent.
  (types are used after their definition, etc.)

We speak of structural inclusion between:

- The internal structure, infered by OCaml.
- The external structure, written by the programmer.

This inclusion is checked when compiling the implementation.

The definition of a type can be hidden, making it abstract.

- The syntax is `type t` (cut everything from the `=`).
- Abstract types cannot be constructed or destructed except in their own module.
- The structure is completely unknown of the outside.
- Don't forget to provide builders and accessors !

What you cannot do:

- Hide only some fields of a record.
- Hide only some constructors of a sum.

Why type abstraction ?

- Usual point: to control the implementation.
- To clarify the API documentation (not always the best solution).
- To preserve internal invariants.
- For program architecture: write abstract interfaces first.
- For quick prototyping (bosket oriented programming).

File `momo.mli`, its type t abstracted.

```
type t
val origin : t
val above : t -> t -> bool
```

Note the regression: one cannot use pattern matching on `Momo.t`.

The previous `main.ml` is not valid anymore:

```
let o : Momo.t = Momo.origin
let p : Momo.t = { Momo.x = 2 ; Momo.y = 4 }
let () = assert (Momo.above p o)
```

We have to rework `momo.mli`, to provide the necessary builders and accessors.

```
type t
val origin : t
val translate : t -> int -> int -> t
val above : t -> t -> bool
```

And update `main.ml`:

```
let o : Momo.t = Momo.origin
let p : Momo.t = Momo.(translate x 4 origin)
let () = assert (Momo.above p o)
```

A less brutal alternative to type abstraction.

- Syntax: `type t = private ... .`
- The definition is kept public and appears in the API.
- Values cannot be constructed outside of the module.
- Values can be destructed outside of the module.

Differences with type abstraction:

⊖ Less possibilities of implementation rewrite.
⊜ Can preserve internal invariants.
⊕ Can make the documentation clearer.
⊕ Makes values destructible for the outside.
⊕ No need for accessors, just builders.

A same interface can contain a mix of public, abstract and private types.

Private types 2/2

File `momo.ml` remains unchanged:

```
type t = { x : int ; y : int }
let origin = { x = 0 ; y = 0 }
let translate { x ; y } dx dy = { x = x + dx ; y = y + dy }
let above pa pb = pa.x >= pb.x
```

We add the `private` modifier to `momo.mli`.

```
type t = private { x : int ; y : int }
val origin : t
val translate : t -> int -> int -> t
val above : t -> t -> bool
```

And update `main.ml`:

```
let o : Momo.t = Momo.origin
let p : Momo.t = Momo.(translate x 4 origin)
let () = assert (Momo.above p o)
```

The module Language

Local modules
Local signatures
Example: variations on a (key x value) table
Abstraction and signature rewriting
Composition

Local modules

A module can define child modules:

- Syntax: `module Name = struct ... end`.
- They support the same features as toplevel modules, including modules.
- Access Syntax: `Module.Child.Child. ...`

That can be used for:

- Architecturing the API (by topic, hierarchically, etc.)
- Grouping local utilities to be hidden.
- Locally extending / patching extermal modules.
- Defining functor parameters (to be continued).

Local signatures

Child modules can also be restricted using signatures.

In the parent module signature:

- Syntax: `module Name : sig ... end`.
- Elements inside the `sig` are the same than in `mli` s.
- This way, the module is restricted only for the outside world.
- It is also possible to hide a child module completely.

Inside the module.

- A child module can also be restricted for the rest of the module.
- Syntax: `module Name : sig ... end = struct ... end`

Signatures can be named:

- Syntax: `module type NAME = sig ... end`
- For use in interfaces: `module Name : NAME`
- Or implementation: `module Name : NAME = struct ... end`

We will:

1. Build a quick and dirty on disk (key x value) storage.
2. Make the value type polymorph by requiring serializers..
3. Write an interface for it.
4. Then we will write a simple in memory storage.
5. And combine them into a cached disk storage.
6. We will make them available as three modules with the same interface.

A simple (key x value) database:

- One file for each key.
- String data stored as is.

```
let init dir =
  Unix.mkdir dir 0o750
let put n v dir =
  let fp = open_out (Filename.concat dir n) in
  output_string fp v ;
  close_out fp
let get n dir =
  let fn = Filename.concat dir n in
  if not (Sys.file_exists fn) then raise Not_found ;
  let fp = open_in fn in
  let len = in_channel_length fp in
  let buf = Bytes.create len in
  really_input fp buf 0 len ;
  close_in fp ;
  Bytes.to_string buf
```

We implement value polymorphism by providing converters:

```
let init dir =
  Unix.mkdir dir 0o750
let put to_string n v dir =
  let fp = open_out (Filename.concat dir n) in
  output_string fp (to_string v) ;
  close_out fp
let get of_string n dir =
  let fn = Filename.concat dir n in
  if not (Sys.file_exists fn) then raise Not_found ;
  let fp = open_in fn in
  let len = in_channel_length fp in
  let buf = Bytes.create len in
  really_input fp buf 0 len ;
  close_in fp ;
  of_string (Bytes.to_string buf)
```

We want to give it a proper abstract type interface:

```
type 'a table

val init
  : ('a -> string) -> (string -> 'a) -> string -> 'a table
val put
  : string -> 'a -> 'a table -> unit
val get
  : string -> 'a table -> 'a
```

So we make a real `'a table` type, in which we embed the converters:

```
type 'a table =
  { to_string : 'a -> string;
    of_string : string -> 'a ;
    dir : string }
let init to_string of_string dir =
  Unix.mkdir dir 0o750 ;
  { to_string ; of_string ; dir }
let put n v { to_string ; dir } =
  (* unchanged *)
let get n { of_string ; dir } =
  (* unchanged *)
```

Now let's write the memory storage, using OCaml hash tables.
We write it so that it fits in the same interface.

```
type 'a table =
  { to_string : 'a -> string;
    of_string : string -> 'a ;
    table : (string, string) Hashtbl.t }
let init to_string of_string _ =
  let open Hashtbl in
  let table : (string, string) t = create 100 in
  { to_string ; of_string ; table }
let put n v { to_string ; table } =
  Hashtbl.replace table n (to_string v)
let get n { of_string ; table } =
  of_string (Hashtbl.find table n)
```

Notice how the database name is unused.
We'll correct that later on.

We group them inside a single compilation unit:

```
module In_memory_table = struct
  type 'a table = { (* ... *) } (* ... *)
end
module On_disk_table = struct
  type 'a table = { (* ... *) } (* ... *)
end
```

Whose interface is:

```
module In_memory_table : sig
  type 'a table  (* ... *)
end
module On_disk_table : sig
  type 'a table (* ... *)
end
```

Of course, we want to make clear that the API is the same for both.
So we name it and the interface becomes:

```
module type TABLE = sig
  type 'a table
  val init
    : ('a -> string) -> (string -> 'a) -> string -> 'a table
  val put : string -> 'a -> 'a table -> unit
  val get : string -> 'a table -> 'a
end

module In_memory_table : TABLE
module On_disk_table : TABLE
```

TABLE appears in the interface, it must appear in the implementation.

```
module type TABLE = sig (* ... *) end
module In_memory_table = struct (* ... *) end
module On_disk_table = struct (* ... *) end
```

Now we can add implementations of the API, for instance this cached one.

```
module Cached_table = struct
  type 'a table =
    'a On_disk_table.table * 'a In_memory_table.table
  let init to_string of_string dir =
    let ds = On_disk_table.init to_string of_string dir in
    let ms = In_memory_table.init to_string of_string dir in
    (ds, ms)
  let put n v (ds, ms) =
    On_disk_table.put n v ds ;
    In_memory_table.put n v ms
  let get n (ds, ms) =
    try
      In_memory_table.get n ms
    with Not_found ->
      let res = On_disk_table.get n ds in
      In_memory_table.put n res ms ;
      res
end
```

In the previous example, the database name was:

- used by the filesystem implementation as a relative directory ;
- dropped by the memory implementation.

Both cases are suboptimal.

Ideally, we would like to:

- Require all modules to provide a `param` type.
- While leaving implementations free to use whatever they need (here `unit` for in-memory, a path for on-disk).
- And making these types public. (their values have to be forged externally to be passed to `init`).

That is what interface rewriting is for.

The abstract types of a named interface can be refined in two ways.

- Variant 1: `NAME with type t = def`
  will build the same interface with `t` publicly specified to be `def`.
- Variant 2: `NAME with type t := def`
  will remove the declaration of `t` and replace all its occurences with `def`

Suppose we have the interface:

```
module type OF_STRING = sig
  type t
  val of_string : string -> t
end
```

We can rewrite it as follows:

```
module Float_of_string : OF_STRING with type t = float
```

Equivalent to writing

```
module Float_of_string : sig
  type t = float
  val of_string : string -> t
end
```

Or alternatively, we can make `t` disappear:

```
module Float_of_string : OF_STRING with type t := float
```

Equivalent to writing

```
module Float_of_string : sig
  val of_string : string -> float
end
```

Example: contextual table initialization

In our example, we could update `TABLE` to be:

```
module type TABLE = sig
  type 'a table
  type param
  val init
    : ('a -> string) -> (string -> 'a) -> param -> 'a table
  val put : string -> 'a -> 'a table -> unit
  val get : string -> 'a table -> 'a
end
```

Then the modules could be given refined signatures:

```
module In_memory_table
  : TABLE with type param := unit
module On_disk_table
  : TABLE with type param := string
```

An other module language primitive is `include`.

In signatures

- To type module level inheritance.
- With signature rewriting, to type module level traits.
- Syntax: `include NAME`
- With rewriting: `include NAME with type ...`

In implementation,

- To extend or patch existing modules.
- Syntax: `include Name`
- With signature: `include (Name : NAME)`
- With rewritten signature: `include (Name : NAME with type ...)`

We want to provide pre-instanciated cached tables for primitive types.
Their interface is simpler, for instance `Int_table`'s should be:

```
module Int_table : sig
  type t
  val init : string -> t
  val put : string -> int -> t -> unit
  val get : string -> t -> int
end
```

First, let's define the interface.

We define a generic interface:

```
module type TYPED_TABLE = sig
  type t
  type value
  val init : string -> t
  val put : string -> value -> t -> unit
  val get : string -> t -> value
end
```

And instanciate it by rewriting.

```
module Int_table
  : TYPED_TABLE with type value := int
module Float_table
  : TYPED_TABLE with type value := float
module String_table
  : TYPED_TABLE with type value := string
```

Then, we implement the modules, by patching the polymorphic implementation.

```
module Int_table = struct
  include In_memory_table
  type t = int table
  let init dir = init string_of_int int_of_string dir
end
module Float_table = struct
  include In_memory_table
  type t = float table
  let init dir = init string_of_float float_of_string dir
end
module String_table = struct
  include In_memory_table
  type t = string table
  let init dir = init (fun s -> s) (fun s -> s) dir
end
```

We want an alternative version of tables, with a `clear` operation.
We extend the signature using `include`, and instantiate it:

```
module type CLEARABLE_TABLE = sig
  include TABLE
  val clear : 'a table -> unit
end

module Clearable_in_memory_table
  : CLEARABLE_TABLE with type param := unit
module Clearable_on_disk_table
  : CLEARABLE_TABLE with type param := string
module Clearable_cached_table
  : CLEARABLE_TABLE with type param := string
```

Example: clearable tables 2 / 2

And we extend the implementations using `include`:

```
module Clearable_in_memory_table = struct
  include In_memory_table
  let clear { table } = Hashtbl.clear table
end
module Clearable_on_disk_table = struct
  include On_disk_table
  let clear { dir } =
    Array.iter
      (fun f -> Unix.unlink (Filename.concat dir f))
      (Sys.readdir dir)
end
module Clearable_cached_table = struct
  include Cached_table
  let clear (ds, ms) =
    Clearable_in_memory_table.clear ms ;
    Clearable_on_disk_table.clear ds
end
```

Using Standard Functors

Functor Application
Set and Map

Functor Application

What are functors ?

- Parametric modules.
- Module level functions.

The code of a functor:

- Supposes that some module of a given signature exists.
- Is linked to such a module by a functor application at runtime.

Syntax:

- Functor signatures:
  `module Name : functor (Arg : ARG) -> RESULT.`
- Functior application:
  `module Instance = Name (Arg).`

The standard library defines:

- `Set` , functional sets of elements of a given type ;
- `Map` functional polymorphic maps of keys of a given type.

For constructing maps, the `map.mli` exports the functor:

```
module Make (Ord : OrderedType) : S with type key = Ord.t
```

- `OrderedType` is the signature that the parameter must respect.
  It defines an abstract type `t` , the type of keys.
- `S` is the signature of the result.
  It is linked to the parameter by the `with type` syntax.

In more details, the `OrderedType` signature is:

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

And an extract of `S`:

```
module type S = sig
  type key
  type (+'a) t
  val empty: 'a t
  val is_empty: 'a t -> bool
  (* ... *)
end
```

Let's instanciate Map to build maps of strings.

```
module StringOrderedType = struct
  type t = string
  let compare = Pervasives.compare
end
module StringMap = Map.Make (String)
```

StringMap is then usable as any other module.

More concisely:

```
module StringMap = Map.Make (struct
  type t = string
  let compare = Pervasives.compare
end)
```

Actually, the `String` module already defines `t` and `compare` :

```
module StringMap = Map.Make (String)
```

If a map is only needed for a local computation, one can write:

```
let module StringMap = Map.Make (String) in (* expr *)
```

Outline