Separate Compilation in OCaml

P. Cousot

July 29, 2003

1 Modules

A module is a named structure which fields define types, data and operations on data:

```
module B_implementation =
struct
  type entier = int
  let c = ref 1
  let i () = c := !c + !c ; !c
  let p v = print_int v; print_newline ()
end;;
```

The type of a module is a signature indicating the types of the fields of the structure:

```
% ocaml
                  Objective Caml version 3.06

# module B_implementation =
struct
  type entier = int
  let c = ref 1
  let i () = c := !c + !c ; !c
                  let p v = print_int v; print_newline ()
end;;
  module B_implementation :
  sig
         type entier = int
         val c : int ref
         val i : unit -> int
         val p : int -> unit
  end
# #quit;;
```

The fields of the structure are accessed by its name, followed by a dot and the name of the field:

```
# B_implementation.c;;
- : int ref = {contents = 1}
# i ();;
Unbound value i
# B_implementation.i ();;
- : int = 2
```

The scope of a structure can be opened to access its fields directly:

```
# open B_implementation;;
# i ();;
- : int = 4
# c;;
- : int ref = {contents = 4}
# i ();;
- : int = 8
# p (i ());;
16
- : unit = ()
# p (i ());;
32
- : unit = ()
```

2 Signatures

A signature specifies which components of a structure are accessible from the outside, and with which type. It can be used to hide some components of a structure (e.g. local function definitions) or export some components with a restricted type:

```
# module type A_signature =
  sig
         type entier
         val i : unit -> entier
         val p : entier -> unit
  end;;
module type A_signature =
  sig type entier val i : unit -> entier val p : entier -> unit end
```

A signature can be used to make a module “abstract” in the sense that some of its fields and its actual representation as a “concrete” type and can b is hidden:

```
# module B = (B_implementation:A_signature);;
module B : A_signature
# B.c;;
Unbound value B.c
```

```
# B.i ();;
- : B.entier = <abstr>
# B.p (B.i ());;
128
- : unit = ()
# open B;;
# p (i ());;
256
- : unit = ()
```

One can write either:

```
module B = (B_implementation:A_signature);;
```

or equivalently:

```
module B : A_signature = B_implementation;;
```

3 Functors

A functor maps structures to structures. More precisely, a functor F parameterized with a formal parameter A (along with the expected signature for A) maps any actual implementation B (B', B'', ...) of the expected signature for A to a structure $F(B)$ ($F(B')$, $F(B'')$, ...).

```
# module type C_signature =
  functor (A: A_signatur  e) ->
                        sig
                                              val i : unit -> A.entier
                                              val p : A      .entier -> unit
                        end;;
  module type C_signature =
  functor (A : A_signature) ->
         sig val i : unit -> A.entier val p : A.entier -> unit end
# module C_implementation =
  functor (A: A_signatur  e) ->
  struct
                        type ent = A.entier
                        let f i = (      )
                        let i () = f (A.i ()); A.i ()
                        let p = A.p
                   end;;
module C_implementation :
  functor (A : A_signature) ->
         sig
                type ent = A.entier
                val f : 'a -> unit
                val i : unit -> A.entier
```

```
                val p : A.entier -> unit
        end
# module C = (C_implementation:C_signature);;
module C : C_signature
# module D = C(B);;
module D : sig val i : unit -> B.entier val p : B.entier -> unit end
# D.p (D.i ());;
1024
- : unit = ()
# open D;;
# p (i ());;
4096
- : unit = ()
```

4 Example modular program

In summary, the modular program is:

```

module type A_signature =
  sig
    type entier
    val i : unit -> entier
    val p : entier -> unit
  end;;

module B_implementation =
struct
  type entier = int
  let c = ref 1
  let i () = c := !c + !c ; !c
  let p v = print_int v; print_newline ()
end;;
module B = (B_implementation:A_signature);;

module type C_signature =
  functor (A: A_signature) ->
        sig
                val i : unit -> A.entier
                val p : A.entier -> unit
        end;;

module C_implementation =
  functor (A: A_signature) ->
  struct
        type ent = A.entier
        let f i = ()
        let i () = f (A.i ()); A.i ()
        let p = A.p
  end;;

```

```
module C = (C_implementation:C_signature);;

module D = C(B);;

D.p (D.i ());;

```

The program `p.ml` can be interpreted as follows:

```
ocaml p.ml
4
```

The program `p.ml` can also be compiled and executed as follows:

```
% ocamlc p.ml
% ./a.out
4
```

5 Separate Compilation

The modules `Name` of the program can be organized into separate files:

- `name.mli`, analogous to the inside of a `sig` ... `end` construct, for signatures specifying the module interface;
- `name.ml`, analogous to the inside of a `struct` ... `end` construct, specifying the module interface.

So a pair of files `name.mli` and `name.ml` is equivalent to the definition of a module `Name` (same name as the base name of the two files, with the first letter capitalized) that would be defined as follows:

```
module Name: sig (* contents of file name.mli *) end
           = struct (* contents of file name.ml *) end;;
```

A single file `name.ml` (without corresponding `name.mli`) is equivalent to the following definition:

```
module Name = struct (* contents of file name.ml *) end;;
```

The program `p.ml` can be decomposed into the following separate files:

- `a.mli`

```
module type A_signature =
  sig
         type entier
         val i : unit -> entier
         val p : entier -> unit
  end;;
```

- b.mli

```
open A;;
module B : A_signature;;
```

- b.ml

```
open A;;
module B_implementation =
struct
  type entier = int
  let c = ref 1
  let i () = c := !c + !c ; !c
  let p v = print_int v; print_newline ()
end;;
module B = (B_implementation:A_signature);;
```

- c.mli

```
open A;;
module type C_signature =
  functor (A: A_signature) ->
         sig
                val i : unit -> A.entier
                val p : A.entier -> unit
         end;;
module C : C_signature;;
```

- c.ml

```
open A;;
module type C_signature =
  functor (A: A_signature) ->
         sig
                val i : unit -> A.entier
                val p : A.entier -> unit
         end;;
module C_implementation =
  functor (A: A_signature) ->
  struct
```

```
          type ent = A.entier
          let f i = ()
          let i () = f (A.i ()); A.i ()
          let p = A.p
  end;;
module C = (C_implementation:C_signature)
```

- `d.ml`

```
open B;;
open C;;
module D = C(B);;
```

- `e.ml`

```
open D;;
D.p (D.i ());;
```

The files defining the compilation units can be compiled separately using the `ocamlc -c` command (the `-c` option means "compile only, do not try to link"). The compilation of a `.mli` produces a compiled module interface file `.cmi` while that of a `.mlo` produces a compiled module object file `.cmi`. When all units have been compiled, their `.mlo` files must be linked together using the `ocaml -o exec` command to produce an executable file `exec` (`a.out` by default):

```
ocamlc -c a.mli
ocamlc -c b.mli
ocamlc -c b.ml
ocamlc -c c.mli
ocamlc -c c.ml
ocamlc -c d.ml
ocamlc -c e.ml
ocamlc b.cmo c.cmo d.cmo e.cmo
./a.out
4
```

Notice that only top-level structures can be mapped to separately-compiled files, but not functors nor module types. So the functors or module type signatures have to be included inside a top-level structure defined in a separate file.

For example, the separately compiled files `a.mli`, `b.mli`, `b.ml`, `c.mli`, `c.ml`, `d.ml` and `e.ml` are equivalent to the following single file:

```

module A = struct
module type A_signature =
  sig
    type entier
```

```
    val i : unit -> entier
    val p : entier -> unit
  end;;
end;;
module B : sig
open A;;
module B : A_signature;;
end = struct
open A;;
module B_implementation =
struct
  type entier = int
  let c = ref 1
  let i () = c := !c + !c ; !c
  let p v = print_int v; print_newline ()
end;;
module B = (B_implementation:A_signature);;
end;;
module C : sig
open A;;
module type C_signature =
  functor (A: A_signature) ->
         sig
                val i : unit -> A.entier
                val p : A.entier -> unit
         end;;
module C : C_signature;;
end = struct
open A;;
module type C_signature =
  functor (A: A_signature) ->
         sig
                val i : unit -> A.entier
                val p : A.entier -> unit
         end;;
module C_implementation =
  functor (A: A_signature) ->
  struct
         type ent = A.entier
         let f i = ()
         let i () = f (A.i ()); A.i ()
         let p = A.p
  end;;
module C = (C_implementation:C_signature)
end;;
module D = struct
open B;;
open C;;
module D = C(B);;
end;;
open D;;
D.p (D.i ());;
```

A Example script

The following script summarizes the OCaml module system.

```
Script started on Tue Jul 29 11:30:48 2003
% ls

a.mli           c.ml            e.ml            p.ml
b.ml            c.mli           makefile        scriptfile.ml
b.mli           d.ml            makefile.depend typescript
% make

rm: No match.
make: [clean] Error 1 (ignored)
*** execution of the original program p.ml:
ocaml p.ml
4
*** generic separate compilation of the modules:
ocamlc -c a.mli
ocamlc -c b.mli
ocamlc -c b.ml
ocamlc -c c.mli
ocamlc -c c.ml
ocamlc -c d.ml
ocamlc -c e.ml
ocamlc a.mli b.cmo b.mli c.cmo c.mli d.cmo e.cmo
*** execution of the compiled code:
./a.out
4
*** customed separate compilation of the modules:
ocamlc -c a.mli
ocamlc -c b.mli
ocamlc -c b.ml
ocamlc -c c.mli
ocamlc -c c.ml
ocamlc -c d.ml
ocamlc -c e.ml
ocamlc b.cmo c.cmo d.cmo e.cmo
*** execution of the compiled code:
./a.out
4
*** creation of a script file:
*** ocaml in script mode:
ocaml scriptfile.ml
4
*** compilation of the scriptfile:
ocamlc scriptfile.ml
*** execution of the compiled code:
./a.out
4
*** original program p.ml:


module type A_signature =
  sig
```

```
52          type entier
53          val i : unit -> entier
54          val p : entier -> unit
55    end;;
56
57  module B_implementation =
58  struct
59    type entier = int
60    let c = ref 1
61    let i () = c := !c + !c ; !c
62    let p v = print_int v; print_newline ()
63  end;;
64  module B = (B_implementation:A_signature);;
65
66  module type C_signature =
67    functor (A: A_signature) ->
68          sig
69                val i : unit -> A.entier
70                val p : A.entier -> unit
71          end;;
72
73  module type C_signature =
74    functor (A: A_signature) ->
75          sig
76                val i : unit -> A.entier
77                val p : A.entier -> unit
78          end;;
79
80  module C_implementation =
81    functor (A: A_signature) ->
82    struct
83          type ent = A.entier
84          let f i = ()
85          let i () = f (A.i ()); A.i ()
86          let p = A.p
87    end;;
88  module C = (C_implementation:C_signature)
89
90  module D = C(B);;
91
92  D.p (D.i ());;
93
94  *** modules:
95
96  ** a.mli
97
98  module type A_signature =
99    sig
100         type entier
101         val i : unit -> entier
102         val p : entier -> unit
103   end;;
104
105 ** b.mli
```

```
106
107 open A;;
108 module B : A_signature;;
109
110 ** b.ml
111
112 open A;;
113 module B_implementation =
114 struct
115   type entier = int
116   let c = ref 1
117   let i () = c := !c + !c ; !c
118   let p v = print_int v; print_newline ()
119 end;;
120 module B = (B_implementation:A_signature);;
121
122 ** c.mli
123
124 open A;;
125 module type C_signature =
126   functor (A: A_signature) ->
127          sig
128                 val i : unit -> A.entier
129                 val p : A.entier -> unit
130          end;;
131 module C : C_signature;;
132
133 ** c.ml
134
135 open A;;
136 module type C_signature =
137   functor (A: A_signature) ->
138          sig
139                 val i : unit -> A.entier
140                 val p : A.entier -> unit
141          end;;
142 module C_implementation =
143   functor (A: A_signature) ->
144   struct
145          type ent = A.entier
146          let f i = ()
147          let i () = f (A.i ()); A.i ()
148          let p = A.p
149   end;;
150 module C = (C_implementation:C_signature)
151
152 ** d.ml
153
154 open B;;
155 open C;;
156 module D = C(B);;
157
158 ** e.ml
159
```

```
open D;;
D.p (D.i ());;

*** script file:

module A = struct
module type A_signature =
  sig
         type entier
         val i : unit -> entier
         val p : entier -> unit
  end;;
end;;
module B : sig
open A;;
module B : A_signature;;
end = struct
open A;;
module B_implementation =
struct
  type entier = int
  let c = ref 1
  let i () = c := !c + !c ; !c
  let p v = print_int v; print_newline ()
end;;
module B = (B_implementation:A_signature);;
end;;
module C : sig
open A;;
module type C_signature =
  functor (A: A_signature) ->
         sig
                val i : unit -> A.entier
                val p : A.entier -> unit
         end;;
module C : C_signature;;
end = struct
open A;;
module type C_signature =
  functor (A: A_signature) ->
         sig
                val i : unit -> A.entier
                val p : A.entier -> unit
         end;;
module C_implementation =
  functor (A: A_signature) ->
  struct
         type ent = A.entier
         let f i = ()
         let i () = f (A.i ()); A.i ()
         let p = A.p
  end;;
module C = (C_implementation:C_signature)
end;;
```

```
module D = struct
open B;;
open C;;
module D = C(B);;
end;;
open D;;
D.p (D.i ());;

*** interactive mode, type:
  ocaml
and then:
  #use "scriptfile.ml";;
  #quit;;
to use "ocaml" interactively with "scriptfile.ml"
% ocaml

                  Objective Caml version 3.06

# #use "scriptfile.ml";;
module A :
  sig
         module type A_signature =
                sig type entier val i : unit -> entier val p : entier -> unit end
  end
module B : sig module B : A.A_signature end
module C :
  sig
         module type C_signature =
                functor (A : A.A_signature) ->
                  sig val i : unit -> A.entier val p : A.entier -> unit end
         module C : C_signature
  end
module D :
  sig
         module D : sig val i : unit -> B.B.entier val p : B.B.entier -> unit end
  end
4
- : unit = ()
# D.p (D.i ());;
16
- : unit = ()
# D.p (D.i ());;
64
- : unit = ()
# D.p (D.i ());;
256
- : unit = ()
# D.p (D.i ());;
1024
- : unit = ()
# #quit;;
% make clean

% ^D
```

```
Script done on Tue Jul 29 11:33:35 2003
```

B reference

http://caml.inria.fr/ocaml/htmlman/manual004.html