

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:

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

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

```
1 % ocaml
2           Objective Caml version 3.06
3
4 # module B_implementation =
5   struct
6     type entier = int
7     let c = ref 1
8     let i () = c := !c + !c ; !c
9           let p v = print_int v; print_newline ()
10  end;;
11   module B_implementation :
12     sig
13       type entier = int
14       val c : int ref
15       val i : unit -> int
16       val p : int -> unit
17     end
18 # #quit;;
```

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

```

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

```

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

```

1 # open B_implementation;;
2 # i ();;
3 - : int = 4
4 # c;;
5 - : int ref = {contents = 4}
6 # i ();;
7 - : int = 8
8 # p (i ());;
9 16
10 - : unit = ()
11 # p (i ());;
12 32
13 - : 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:

```

1 # module type A_signature =
2   sig
3     type entier
4     val i : unit -> entier
5     val p : entier -> unit
6   end;;
7 module type A_signature =
8   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:

```

1 # module B = (B_implementation:A_signature);;
2 module B : A_signature
3 # B.c;;
4 Unbound value B.c

```

```

5 # B.i ();;
6 - : B.entier = <abstr>
7 # B.p (B.i ());;
8 128
9 - : unit = ()
10 # open B;;
11 # p (i ());;
12 256
13 - : unit = ()

```

One can write either:

```

1 module B = (B_implementation:A_signature);;

```

or equivalently:

```

1 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 $F(B)$ ($F(B')$, $F(B'')$, ...).

```

1 # module type C_signature =
2   functor (A: A_signatur e) ->
3     sig
4       val i : unit -> A.entier
5       val p : A.entier -> unit
6     end;;
7 module type C_signature =
8   functor (A : A_signature) ->
9     sig val i : unit -> A.entier val p : A.entier -> unit end
10 # module C_implementation =
11   functor (A: A_signatur e) ->
12     struct
13       type ent = A.entier
14       let f i = ( )
15       let i () = f (A.i ()); A.i ()
16       let p = A.p
17     end;;
18 module C_implementation :
19   functor (A : A_signature) ->
20     sig
21       type ent = A.entier
22       val f : 'a -> unit
23       val i : unit -> A.entier

```

```

24         val p : A.entier -> unit
25     end
26 # module C = (C_implementation:C_signature);;
27 module C : C_signature
28 # module D = C(B);;
29 module D : sig val i : unit -> B.entier val p : B.entier -> unit end
30 # D.p (D.i ());;
31 1024
32 - : unit = ()
33 # open D;;
34 # p (i ());;
35 4096
36 - : unit = ()

```

4 Example modular program

In summary, the modular program is:

```

1
2 module type A_signature =
3     sig
4         type entier
5         val i : unit -> entier
6         val p : entier -> unit
7     end;;
8
9 module B_implementation =
10    struct
11        type entier = int
12        let c = ref 1
13        let i () = c := !c + !c ; !c
14        let p v = print_int v; print_newline ()
15    end;;
16 module B = (B_implementation:A_signature);;
17
18 module type C_signature =
19    functor (A: A_signature) ->
20        sig
21            val i : unit -> A.entier
22            val p : A.entier -> unit
23        end;;
24
25 module C_implementation =
26    functor (A: A_signature) ->
27        struct
28            type ent = A.entier
29            let f i = ()
30            let i () = f (A.i ()); A.i ()
31            let p = A.p
32        end;;
33

```

```

34 module C = (C_implementation:C_signature);;
35
36 module D = C(B);;
37
38 D.p (D.i ());;
39

```

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

```

1  ocaml p.ml
2  4

```

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

```

1  % ocamlc p.ml
2  % ./a.out
3  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:

```

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

```

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

```

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

```

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

- `a.mli`

```

1 module type A_signature =
2   sig
3     type entier
4     val i : unit -> entier
5     val p : entier -> unit
6   end;;

```

- b.mli

```

1 open A;;
2 module B : A_signature;;

```

- b.ml

```

1 open A;;
2 module B_implementation =
3   struct
4     type entier = int
5     let c = ref 1
6     let i () = c := !c + !c ; !c
7     let p v = print_int v; print_newline ()
8   end;;
9 module B = (B_implementation:A_signature);;

```

- c.mli

```

1 open A;;
2 module type C_signature =
3   functor (A: A_signature) ->
4     sig
5       val i : unit -> A.entier
6       val p : A.entier -> unit
7     end;;
8 module C : C_signature;;

```

- c.ml

```

1 open A;;
2 module type C_signature =
3   functor (A: A_signature) ->
4     sig
5       val i : unit -> A.entier
6       val p : A.entier -> unit
7     end;;
8 module C_implementation =
9   functor (A: A_signature) ->
10  struct

```

```

11         type ent = A.entier
12         let f i = ()
13         let i () = f (A.i ()); A.i ()
14         let p = A.p
15     end;;
16 module C = (C_implementation:C_signature)

```

- d.ml

```

1   open B;;
2   open C;;
3   module D = C(B);;

```

- e.ml

```

1   open D;;
2   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 `.cmo`. 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):

```

1   ocamlc -c a.mli
2   ocamlc -c b.mli
3   ocamlc -c b.ml
4   ocamlc -c c.mli
5   ocamlc -c c.ml
6   ocamlc -c d.ml
7   ocamlc -c e.ml
8   ocamlc b.cmo c.cmo d.cmo e.cmo
9   ./a.out
10  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:

```

1
2   module A = struct
3   module type A_signature =
4       sig
5           type entier

```

```

6     val i : unit -> entier
7     val p : entier -> unit
8     end;;
9 end;;
10 module B : sig
11 open A;;
12 module B : A_signature;;
13 end = struct
14 open A;;
15 module B_implementation =
16 struct
17     type entier = int
18     let c = ref 1
19     let i () = c := !c + !c ; !c
20     let p v = print_int v; print_newline ()
21 end;;
22 module B = (B_implementation:A_signature);;
23 end;;
24 module C : sig
25 open A;;
26 module type C_signature =
27     functor (A: A_signature) ->
28         sig
29             val i : unit -> A.entier
30             val p : A.entier -> unit
31         end;;
32 module C : C_signature;;
33 end = struct
34 open A;;
35 module type C_signature =
36     functor (A: A_signature) ->
37         sig
38             val i : unit -> A.entier
39             val p : A.entier -> unit
40         end;;
41 module C_implementation =
42     functor (A: A_signature) ->
43     struct
44         type ent = A.entier
45         let f i = ()
46         let i () = f (A.i ()); A.i ()
47         let p = A.p
48     end;;
49 module C = (C_implementation:C_signature)
50 end;;
51 module D = struct
52 open B;;
53 open C;;
54 module D = C(B);;
55 end;;
56 open D;;
57 D.p (D.i ());;

```


A Example script

The following script summarizes the OCaml module system.

```
1 Script started on Tue Jul 29 11:30:48 2003
2 % ls
3
4 a.mli          c.ml           e.ml           p.ml
5 b.ml          c.mli         makefile       scriptfile.ml
6 b.mli         d.ml          makefile.depend typescript
7 % make
8
9 rm: No match.
10 make: [clean] Error 1 (ignored)
11 *** execution of the original program p.ml:
12 ocaml p.ml
13 4
14 *** generic separate compilation of the modules:
15 ocamlc -c a.mli
16 ocamlc -c b.mli
17 ocamlc -c b.ml
18 ocamlc -c c.mli
19 ocamlc -c c.ml
20 ocamlc -c d.ml
21 ocamlc -c e.ml
22 ocamlc a.mli b.cmo b.mli c.cmo c.mli d.cmo e.cmo
23 *** execution of the compiled code:
24 ./a.out
25 4
26 *** customized separate compilation of the modules:
27 ocamlc -c a.mli
28 ocamlc -c b.mli
29 ocamlc -c b.ml
30 ocamlc -c c.mli
31 ocamlc -c c.ml
32 ocamlc -c d.ml
33 ocamlc -c e.ml
34 ocamlc b.cmo c.cmo d.cmo e.cmo
35 *** execution of the compiled code:
36 ./a.out
37 4
38 *** creation of a script file:
39 *** ocaml in script mode:
40 ocaml scriptfile.ml
41 4
42 *** compilation of the scriptfile:
43 ocamlc scriptfile.ml
44 *** execution of the compiled code:
45 ./a.out
46 4
47 *** original program p.ml:
48
49
50 module type A_signature =
51   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

```

```

160 open D;;
161 D.p (D.i ());;
162
163 *** script file:
164
165 module A = struct
166 module type A_signature =
167   sig
168     type entier
169     val i : unit -> entier
170     val p : entier -> unit
171   end;;
172 end;;
173 module B : sig
174 open A;;
175 module B : A_signature;;
176 end = struct
177 open A;;
178 module B_implementation =
179 struct
180   type entier = int
181   let c = ref 1
182   let i () = c := !c + !c ; !c
183   let p v = print_int v; print_newline ()
184 end;;
185 module B = (B_implementation:A_signature);;
186 end;;
187 module C : sig
188 open A;;
189 module type C_signature =
190   functor (A: A_signature) ->
191     sig
192       val i : unit -> A.entier
193       val p : A.entier -> unit
194     end;;
195 module C : C_signature;;
196 end = struct
197 open A;;
198 module type C_signature =
199   functor (A: A_signature) ->
200     sig
201       val i : unit -> A.entier
202       val p : A.entier -> unit
203     end;;
204 module C_implementation =
205   functor (A: A_signature) ->
206     struct
207       type ent = A.entier
208       let f i = ()
209       let i () = f (A.i ()); A.i ()
210       let p = A.p
211     end;;
212 module C = (C_implementation:C_signature)
213 end;;

```

```

214 module D = struct
215   open B;;
216   open C;;
217   module D = C(B);;
218   end;;
219   open D;;
220   D.p (D.i ());;
221
222   *** interactive mode, type:
223     ocaml
224   and then:
225     #use "scriptfile.ml";;
226     #quit;;
227   to use "ocaml" interactively with "scriptfile.ml"
228   % ocaml
229
230           Objective Caml version 3.06
231
232   # #use "scriptfile.ml";;
233   module A :
234     sig
235       module type A_signature =
236         sig type entier val i : unit -> entier val p : entier -> unit end
237     end
238   module B : sig module B : A.A_signature end
239   module C :
240     sig
241       module type C_signature =
242         functor (A : A.A_signature) ->
243           sig val i : unit -> A.entier val p : A.entier -> unit end
244       module C : C_signature
245     end
246   module D :
247     sig
248       module D : sig val i : unit -> B.B.entier val p : B.B.entier -> unit end
249     end
250   4
251   - : unit = ()
252   # D.p (D.i ());;
253   16
254   - : unit = ()
255   # D.p (D.i ());;
256   64
257   - : unit = ()
258   # D.p (D.i ());;
259   256
260   - : unit = ()
261   # D.p (D.i ());;
262   1024
263   - : unit = ()
264   # #quit;;
265   % make clean
266
267   % ^D

```

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

B reference

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