Programmation Fonctionnelle
Compilation

Adrien Durier

5 octobre 2023

Unités de compilation

Unités de compilation : motivations

Un principe de base du génie logiciel est le découpage d'une application en plusieurs parties indépendantes appelées unités de compilation.

Pour :

- maîtriser la complexité de logiciels de grande taille, horizontalement (divide and conquer) et verticalement (niveaux d'abstractions).
- développer en équipe.
- modifier un morceau indépendamment des autres.
- recompiler uniquement le nécessaire.

Unités de compilation : Fichiers

En OCaml :
1 unité de compilation =
1 fichier interface + 1 fichier implémentation

Fichiers de même nom, avec différences extensions :

- Le fichier interface (`.mli`) définit les types (abstraits ou concrets) et les signatures des valeurs visibles depuis l'extérieur.
- Le fichier implémentation (`.ml`) contient les définitions (concrètes) de tous les types et de toutes les valeurs (visibles ou non) de l'unité de compilation.

Abstrait = sans implémentation dans le fichier.
Concret = avec son implémentation dans le fichier.

Unités de compilation : Espaces de noms

- 1 unité de compilation `toto.mli` + `toto.ml` est désignée par `Toto`. La valeur v de `Toto` est désignée par `Toto.v`.
- La directive `open Toto` permet de désigner la valeur v de `Toto` par v.
- Si deux unités `M` et `N` contiennent v, et que l'on effectue open `M` puis open `N`, alors v désigne `N.v`.

Unités de compilation : Interfaces

Le fichier d'interface spécifie quels types ou valeurs sont accessibles depuis l'extérieur. Pour :

- cacher certains types ou valeurs car ils sont propres à une implémentation spécifique, ou pour faciliter la lecture.
- restreindre le type de certains composants exportés, ou la manière d'y accéder, de façon à ce qu'ils soient utilisés correctement.
- rendre abstraits certains types, pour les mêmes raisons.

Syntaxe :

- Mot-clé `val` pour les valeurs :
  - `val f : int -> 'a -> a list`
- Mot-clé `type` pour les types :
  - `type t = A | B`
  - `type t` (Ceci est un type abstrait)

Unités de compilation : Interfaces et leurs compilations

- Les fichiers d'inferface doivent être compilés via

  `ocamlc -c fichier.mli`

- Le fichier compilé porte l'extension `.cmi`
- Seul le fichier .cmi est finalement nécessaire pour les étapes ultérieures de compilation.

Lors de la compilation d'un fichier d'implémentation `.ml`

- s'il n'y a pas d'interface, un fichier `.cmi` est généré automatiquement : tous les types et valeurs sont exportés
- sinon, le compilateur vérifie que les types inférés depuis le `.ml` sont compatibles avec les types déclarés dans le `.cmi`.

Unités de compilation : dépendances

Pour concevoir une unité de compilation il est seulement nécessaire de connaître les interfaces des autres unités.

- Lorsqu'une unité `M1` fait référence à une unité `M2`, on dit que `M1` dépend de `M2`.
- L'unité `M1` peut faire référence à `M2` dans son interface `M1.mli`, soit dans son implémenation `M1.ml`.
- Dans un programme avec plusieurs unités de compilation, la relation « dépend de » forme un graphe de dépendances.

Le graphe de dépendances définit une ordre partiel de compilation
(Pas de dépendances cycliques)

Unités de compilation : compilation vs édition de liens

- La phase de compilation effectue le typage et la production de codes à trous (on parle de fichiers objets)

  L'option `-c` des compilateurs
  (`ocamlc` ou `ocamlopt` pour le natif)
  permet de compiler sans faire d'édition de liens

  Les fichiers objets portent l'extension `.cmo` (en *bytecode*) ou `.cmx` (en natif).
- La phase d'édition de liens construit un exécutable en associant les trous à des implémentations—selon l'ordre des fichiers donnés en arguments.

Exemple

Exemple : interface

Voici `tri.mli` :

```
val tri_liste : 'a list -> 'a list
```

On compile :

```
ocamlc -c tri.mli
```

Cela produit `tri.cmi`

Exemple : référence à l'interface

Un code qui utilise le module `Tri` dans le fichier `test.ml` :

```
let l1 = [4;1;5;3;2]

let l2 = Tri.tri_liste l1
```

Pour compiler, il suffit de disposer de l'interface du module `Tri` (par exemple dans le répertoire courant) :

```
ocamlc -c test.ml
```

Cela produit un fichier objet test.cmo.

Exemple : implémentation de l'interface

L'implémentation du module, dans `tri.ml`

```
let rec insertion x l = ...

let rec tri_liste l = ...
```

On compile :

```
ocamlc -c tri.ml
```

Cela produit un fichier objet `tri.cmo`

Exemple : édition de liens

On produit un exécutable `test` en liant les fichiers objets `test.cmo` et ceux du module `Tri`.

```
ocamlc -o test tri.cmo test.cmo
```

On peut désormais exécuter `./test`

```
> ./test
4 1 5 3 2
1 2 3 4 5
```

Exemple : recompiler juste le nécessaire

Si on souhaite changer l'implémentation du module `Tri`, par exemple avec un tri rapide, il suffit de fournir une nouvelle implémentation dans un nouveau fichier `tri.ml`

```
let rec partage e l = ...

let rec tri_liste l = ...
```

On recompile juste ce nouveau fichier avec `ocamlc -c tri.ml`

Inutile de recompiler `test.ml`.

Finalement, on refait l'édition de liens :

```
ocamlc -o test tri.cmo test.cmo
```

Langage de modules

Modules

Les notions d'implémentations et interfaces peuvent être rendues plus fines par des constructions du langage.

Définition d'une interface (signature) `I` dans un programme :

```
module type I = sig
    val a : int
    val f : int -> int
end
```

Définition d'une implémentation (module) `M` ayant cette interface :

```
module M : I = struct
    let a = 42
    let b = 3
    let f x = a * x + b
end
```

Le compilateur fait alors les même opérations que si `I` était un fichier `M.mli` et `M` un fichier `M.ml`

Modules paramétrés

- Comme les fonctions, les modules peuvent attendre des paramètres. Ces modules paramétrés sont appelés foncteurs.
- Le langage impose que ces paramètres soient des modules.

Définition d'un foncteur `M` qui attent un module `S` de signature `T` en paramètre :

```
module M ( S : T ) = struct
    ...
end
```

Pour créer une instance de `M`, il suffit de l'appliquer à un module ayant la signature `T`. Par exemple, si `B` a pour signature `T`, alors on crée un `M` en faisant :

```
module A = M(B)
```

Exemple

Exemple : des éléments comparables

Les éléments à trier sont rendus abstraits à l'aide d'une signature qui indique just le fait qu'ils ont une fonction de comparaison :

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

Le type `t` est abstrait.

Exemple : interface d'un trieur d'éléments comparables

Le module de tri à développer aura la signature S suivante :

```
module type S =
  sig
    type t
    val insertion : t -> t list -> t list
    val tri : t list -> t list
  end
```

Exemple : foncteur d'un trieur d'éléments comparables

```
module T(E : ELT) : S with type t = E.t
 =
 struct

  type t = E.t
  type tl = t list

  let rec insertion x l =
    match l with
    | [] -> [x]
    | y :: s ->
       if E.compare x y <= 0 then
          x :: l
       else
           y :: (insertion x s)

  let rec tri l =
    match l with
    | [] -> []
    | x :: s -> insertion x (tri s)
end
```

Exemple : modules trieurs d'éléments comparables

Il ne reste plus qu'à définir des instances du foncteur T et à les utiliser :

```
module T1 = T(E1)
let l1 = [4;1;5;3;2]
let l2 = T1.tri l1
let l3 = T1.insertion 6 l2
```

```
module E2 = struct
    type t = int
    let compare x y = y*y - x*x
  end

module T2 = T(E2)
let l4 = T2.tri l1
```

Exemple : préconditions...

`T1.insertion` prend une liste triée selon `T1` comme argument, et renvoie une liste triée selon `T1`.

Or `T1.insertion` accepte toute liste de type entier. On peut toujours insérer un élément dans une liste non triée, ou mélanger les fonctions de ces deux instances :

```
let l5 = T1.insertion 4 [3;1;2]    (* Bug *)
let l6 = T1.insertion 6 l4         (* Bug *)
```

Il faudrait un type liste spécifique à chaque module (à chaque instance du foncteur T).

Exemple : préconditions. . .garanties par les types

Nouvelle signature pour le foncteur `T` avec type abstrait `tl`.

```
module type SL =
  sig
    type t
    type tl
    val export : tl -> t list
    val insertion : t -> tl -> tl
    val tri : t list -> tl
  end
```

Dans notre implémentation, `tl` sera bien une `t list`, mais au niveau de la déclaration de type du foncteur, rien ne le garantira : ce type `tl` sera donc interne à chaque instance du foncteur `T`.

Exemple : préconditions. . .garanties par les types

De cette manière, impossible de mélanger les listes triées par les différentes instances de T. Impossible également d'insérer un élément dans une liste non triée.

```
  let l5 = T1.insertion 4 [3;1;2]

Error: This expression has type 'a list
 but an expression was expected of type T1.tl = Sort.T(E1).tl
```

```
  let l6 = T1.insertion 6 l4

Error: This expression has type T2.tl = Sort.T(E2).tl
  but an expression was expected
  of type T1.tl = Sort.T(E1).tl
```

Exemple : exportation type interne vers type externe

```
module T(E : ELT) : SL with type t = E.t
 =
  struct

   type t = E.t
   type tl = t list

   let export l = l


   ...
 end
```

Pour pouvoir faire :

```
List.iter (Printf.printf "%d ") (T1.export l2);
```

Entrées - Sorties

Entrées-sorties : Exemple

```
let copy_file f1 f2 =
  let c1 = open_in f1 in
  let c2 = open_out f2 in
  try
    while true do
      let v = input_char c1 in
      output_char c2 v
    done
  with End_of_file ->
    close_in c1; close_out c2

let () = copy_file Sys.argv.(1) Sys.argv.(2)
```

Entrées-sorties : canaux

- Les canaux d'entrée sont de type `in_channel`

```
# stdin;;
- : in_channel = <abstr>
```

- Les canaux de sortie sont de type `out_channel`

```
# stdout;;
- : out_channel = <abstr>
# stderr;;
- : out_channel = <abstr>
```

Entrées-sorties : lecture

- depuis l'entrée standard

  ```
  read_line: unit -> string
  read_int: unit -> int
  ```

- depuis n'importe quel canal d'entrée

  ```
  input_char: in_channel -> char
  input_line: in_channel -> string
  input_byte: in_channel -> int
  ```

```
# let i = read_int ();;
10
val i : int = 10
# let c = input_char stdin;;
a
val c : char = 'a'
```

Entrées-sorties : écriture

- sur la sortie standard

  ```
  print_char: string -> unit
  print_int: int -> unit
  print_string: string -> unit
  ```

- sur n'importe quel canal de sortie

  ```
  output_char: out_channel -> char -> unit
  output_string: out_channel -> string -> unit
  ```

```
# print_char 'a';;
a- : unit = ()
# output_string stdout "bonjour";;
bonjour
- : unit = ()
```

Entrées-sorties : Fichiers comme canaux

Les fichiers sont manipulés comme des canaux.

Ils doivent être ouverts en lecture ou en écriture :

```
open_in : string -> in_channel
```

```
open_out : string -> out_channel
```

```
# let cout = open_out "file.txt";;
val cout : out_channel = <abstr>
# output_string cout "bonjour";;
- : unit = ()
```

Ne pas oublier de fermer les canaux pour être sûr que tout soit bien écrit :

```
# close_out cout;;
- : unit = ()
```

Entrées-sorties : Fichiers comme canaux

```
# let cin = open_in "file.txt";;
val cin : in_channel = <abstr>
# input_char cin;;
- : char = 'b'
# close_in cin;;
- : unit = ()
```

Entrées-sorties : Sérialisation

On peut lire ou écrire des valeurs arbitraires dans des canaux

```
# let c = open_out "foo" in
  output_value c (1, 3.14, true);
  close_out c;;
- : unit = ()
```

Le format d'écriture est spécifique au langage OCaml

Il est préférable d'indiquer le type de la valeur lue

```
# let c = open_in "foo";;
val c : in_channel = <abstr>
# let v : int * float * bool = input_value c;;
val v : int * float * bool = (1, 3.14, true)
```

Entrées-sorties : Sérialisation

La lecture est non sûre :

```
# let c = open_in "foo";;
val c : in_channel = <abstr>
# let v = input_value c in fst (fst v)

Process caml-toplevel segmentation fault
```