Programmation fonctionnelle avancée

Notes de cours

Cours 9

25 novembre 2015

Sylvain Conchon

sylvain.conchon@lri.fr

Système de modules

Unités de compilation

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

Cela permet notamment :

- de mieux maîtriser la complexité de logiciels de grandes tailles
- de réaliser un développement en équipe
- de recompiler rapidemment un programme en ne recompilant que ce qui est nécessaire après une modification

Unités de compilation en Ocaml

En OCaml, chaque unité de compilation est un couple de deux fichiers : le fichier interface et le fichier implémentation

Ces fichiers portent le même préfixe, seules les extensions diffèrent :

- le fichier interface (`.mli`) définit les types (abstraits ou concrets) et les signatures des valeurs visibles à 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.

Désignation d'une unité de compilation

- Si `module.mli` et `module.ml` sont les fichiers d'interface et d'implémentation d'une unité de compilation, on utilisera le nom `Module` pour désigner cette unité. On utilisera également la notation `Module.v` pour faire référence à la valeur `v` de `Module`.
- La directive `open Module` évite d'utiliser la notation pointée pour faire référence aux valeurs de `Module`.

Attention : si deux unités `M` et `N` contiennent la même valeur `v`, alors seule la déclaration de `N` est visible après les deux directives consécutives `open M` et `open N`.

Interfaces

On utilise un fichier d'interface pour spécifier quels types ou valeurs d'une implémentation sont accessibles de « l'extérieur »

Cela permet notamment :

- de cacher certains composants (type ou valeur) ;
- de restreindre le type de certains composants exportés ;
- de rendre abstraits certains types

La syntaxe utilisée dans les fichiers d'interface est la suivante :

- les valeurs sont déclarées en utilisant le mot-clé `val`.

```
val f : int -> 'a -> a list
```

- les types sont déclarés à l'aide du mot-clé `type`

```
type t = A | B
```

- les types abstraits sont des déclarations sans définitions

```
type t
```


Extensions de fichiers

- les fichiers d'inferface doivent être compilés

  `ocamlc -c fichier.mli`

- le fichier compilé porte l'extension `.cmi`
- seul le fichier .cmi est nécessaire pour la compilation séparée

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

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

Graphe de dépendances

L'idée principale du découpage est que 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` soit dans son interface, soit dans son implémenation.
- 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

Compilation et 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`) 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 ≪ remplissant ≫ les trous, selon l'ordre des fichiers donnés en arguments

`Makefile`

Un fichier `Makefile` est constitué de règles de la forme suivante :

```
cible:   liste de dépendances
         actions
```

- Le programme `make <cible>` cherche à exécuter la règle `cible` du fichier `Makefile` du répertoire courant.
- Si aucune règle n'est donnée en argument, c'est la première règle de ce fichier qui est exécutée.
- Les dépendances sont soit des noms d'autres règles, soit des noms de fichiers.
- Les actions sont des commandes shell.

Attention : les lignes contenant les actions doivent commencer par une tabulation.

`Makefile` : exécution d'une règle

- Les dépendances d'une règle sont analysées, si une dépendance est la cible d'une autre règle du Makefile, cette régle est à son tour évaluée.
- Lorsque l'ensemble des dépendances est analysé et si la cible ne correspond pas à un fichier existant ou si un fichier dépendance est plus récent que la régle, les différentes commandes sont exécutées.

De cette manière, un `Makefile` ne recompile que ce qui est nécessaire pour construire une cible

`Makefile` : exemple

- La commande `make foo` pour le `Makefile` ci-dessous permet de compiler le programme `foo`.

```
foo : a.ml b.ml
      ocamlc -c a.ml
      ocamlc -c b.ml
      ocamlc -o foo a.cmo b.cmo
```

- Le `Makefile` ci-dessous permet la compilation séparée.

```
foo : a.cmo b.cmo
      ocamlc -o foo a.cmo b.cmo
a.cmo : a.ml
      ocamlc -c a.ml
b.cmo : b.ml
      ocamlc -c b.ml
```

Makefile : macros

L'outil make permet de définir des macros de la manière suivante :

```
NOM = valeur
```

- NOM est le nom de la macro
- valeur une chaîne de caractères quelconque

On peut ensuite écrire $(NOM) n'importe où dans le fichier comme raccourci pour valeur.

Quelques macros prédéfinies :

$@	Nom de la cible à reconstruire
$<	Nom de la dépendance à partir de laquelle la cible est reconstruite
$^	Liste de toutes les dépendances
$*	Le nom de la cible sans suffixe (cf. règles génériques)

Makefile : exemple (suite)

Le Makefile précédent peut se récrire de la manière suivante :

```
OCAMLC = ocamlc -c
CMO = a.cmo b.cmo

foo : $(CMO)
	ocamlc -o $@ $^

a.cmo : a.ml
	$(OCAMLC) $<

b.cmo : b.ml
	$(OCAMLC) $<
```


Makefile : règles d'inférences

- On peut voir dans l'exemple précédent que les règles a.cmo et b.cmo sont très similaires.
- Afin de les factoriser, make autorise la définition de règles génériques de la forme suivante :

```
%.xxx: %.yyy
	actions
```

Ces règles permettent de construire, à partir d'un fichier quelconque foo.yyy, un fichier foo.xxx.

Makefile : exemple (suite)

On peut encore simplifier le Makefile précédent de la manière suivante :

```
OCAMLC = ocamlc -c
CMO = a.cmo b.cmo

foo : $(CMO)
	ocamlc -o $@ $^

%.cmo : %.ml
	$(OCAMLC) $<
```

Calcul automatique des dépendances

- Afin d'appliquer les règles génériques dans le bon ordre, il faut fournir les dépendances mutuelles des fichiers `.cmo`.
- Cela est calculé automatiquement pour `OCaml` avec l'outil `ocamldep` dont le résultat peut être inclus directement dans le `Makefile` avec l'instruction `include`.

```
.PHONY: depend
OCAMLC = ocamlc -c
CMO = a.cmo b.cmo
foo : depend $(CMO)
     ocamlc -o $@ $^
%.cmo : %.ml
     $(OCAMLC) $<
depend:
     ocamldep *.ml > .depend
include .depend
```


Modules

Les notions de modules et interfaces sont en réalité plus fines que les fichiers et correspondent à des constructions du langage

On définit une interface `I` dans un programme comme ceci :

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

on définit un module `M` ayant cette interface comme ceci :

```
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 `.mli` et `M` un fichier `.ml`

Modules paramétrés

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

Voici par exemple la déclaration d'un module paramétré `M` ayant 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 on suppose que `B` est un module ayant la signature `T`, alors on crée une instance de `M` de la manière suivante :

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


Interface d'un dictionnaire

```
module type DICO = sig
    type key
    type 'a dico
    val empty : 'a dico
    val add : key -> 'a -> 'a dico -> 'a dico
    val mem : key -> 'a dico -> bool
    val find : key -> 'a dico -> 'a
    val remove : key -> 'a dico -> 'a dico
end
```

Dictionnaires paramétrés

Un dictionnaire doit être indépendant du type des clés, mais il est important pour l'implémentation de pouvoir les comparer

On définit donc un dictionnaire comme un module paramétré par un module `Key` ayant la signature `ORDERED` suivante :

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

module MakeDico ( Key : ORDERED ) : DICO = struct
    ...
end
```