Programmation Fonctionnelle
Cours 9
Compilation et Modules

November 20, 2012

Table de matières

Compilation d'un programme monolithique

Accéder aux options et arguments d'un programme

Modules en OCaml

Compilation d'un programme découpé en plusieurs modules

Découpage en modules

Compilation

- Compilation : traduction du code source (écrit dans un langage source, ici OCaml) en code exécutable écrit dans un langage cible.
- Le compilateur met (normalement) le code exécutable dans un fichier à part, qui puis peut être exécuté à partir de la ligne de commande.
- Avantages de la compilation :
  - On obtient un exécutable autonome qu'on peut exécuter sans avoir besoin de tout l'environnement de programmation.
  - Le code exécutable sera plus efficace à exécuter.

Étapes de compilation (simplifié)

1. Analyse syntaxique : peut détecter une erreur de syntaxe.
2. Analyse sémantique : peut détecter une erreur de typage, filtrages non exhaustives, etc.
3. Génération d'un code intermédiaire.
4. Éventuellement optimisations diverses du code.
5. Production du code cible, peut nécessiter la liaison avec des bibliothèques.

Pour en savoir plus : cours informatique du *Analyse syntaxique* (2nd semestre), *Compilation* du M1 et M2

Langages cibles

OCaml supporte deux langages cible différents :

- **Code-octet** (angl.: *byte-code*) : Code destiné à être exécuté sur une « machine virtuelle » : une espèce d'interpréteur destinée exclusivement à exécuter ce type de code, sans boucle d'interaction avec le programmeur. (Pareil que pour Java).
- **Code native** : Code machine qui est spécifique au type de micro-processeur; le code est donc complètement autonome. (Pareil que pour C ou C++).

Avantages du code-octet : portabilité

- Le code exécutable peut être exécuté sur n'importe quel ordinateur, pourvu que la machine virtuelle est installée.
- Les compilateurs vers code natif n'existent que pour quelques types de micro-processeur (Intel 32bit, Intel 64bit, ...).

Avantages du code octet : efficacité (éventuellement)

- Efficacité du code engendré :
  - Le fichier contenant du code-octet est plus petit que dans le cas de la compilation en code natif (car il y a des fonctionnalités qui sont réalisées dans la machine virtuelle).
  - Éventuellement gain de vitesse par le fait que le code-octet est chargé plus rapidement en mémoire (car plus petit).
- Le compilateur vers code octet est plus rapide que le compilateur vers code natif.

Avantages du code natif

- L'exécution du code natif est plus rapide que du code-octet. Avantage surtout pour des applications très intensives en calcul.
- L'exécutable est entièrement autonome.

Conclusion : normalement on préfère la compilation vers code-octet, sauf quand on a vraiment besoin d'un plus de vitesse.

Comment utiliser le compilateur code-octet

- Mettre le code source dans un fichier dont le nom se termine sur `.ml`, par exemple `prog.ml`.
- Exécuter (ligne de commande) `oc mlc prog.ml`.
- S'il n'y a pas d'erreur, le code exécutable est mis dans le fichier `a.out`.
- Exécuter, on lançant (ligne de commande) `./a.out`.
- On peut spécifier directement le nom du fichier exécutable, par exemple :

  ```
  oc mlc -o prog prog.ml
  ```

  pour obtenir un exécutable du nom `prog`.

Exemples (fact1.ml)

```
let rec fact n = if n <=1 then 1 else n*fact(n-1);;
print_int (fact 10);;
print_newline ();;
```


Plus sur l'utilisation du compilateur

- Lancée sur le fichier `prog.ml`, le compilateur produit également des fichiers `prog.cmi` et `prog.cmo` (voir plus tard pour des explications).
- Pour appeler le compilateur vers code native : `oc mlopt prog.ml`.
- Il y a aussi des commandes `oc mlc.opt` et `oc mlopt.opt` : font la même chose que `oc mlc` et `oc mlopt`, mais sont elles même du code natif.
- Plein d'options : voir le manuel OCaml.

Changement dans le programme : entrées

- Le programme n'est plus interactif, on ne peut pas simplement entrer des expressions OCaml à interpréter.
- Un programme compilé peut seulement prendre des entrées par des canaux d'entrée (standard input, fichiers), interface graphique, ou éventuellement l'interface système.
- Par conséquent, un programme compilé doit contenir du code pour lire explicitement les entrées et pour les traiter.

Exemples (fact2.ml)

```
let rec fact n = if n <=1 then 1 else n*fact(n-1);;
let input = int_of_string (read_line ());;
print_int (fact input);;
print_newline ();;
```


Changements dans le programme : sorties

- L'interpréteur sait afficher des données, même dans des types définis par l'utilisateur (sauf fonctions et autres types abstraits).
- Dans un programme compilé on doit se servir des canaux de sortie (stdout, fichiers, etc), et des fonctions appropriées.

Exemples (sortie1.ml)

```
type mytype = Entier of int | Flottant of float;;

let x = Entier 42;;

x;;

(* n'affiche rien si compile *)
```


Exemples (sortie2.ml)

```
type mytype = Entier of int | Flottant of float;;

let x = Entier 42;;

let string_of_mytype = function
  | Entier i -> "Entier("^(string_of_int i)^")"
  | Flottant f -> "Flottant("^(string_of_float f)^")"

print_string (string_of_mytype x);;
print_newline();;
```

Accéder aux options et arguments d'un programme

Les outils doivent souvent pouvoir traiter

- des options : `ls -l -r`
- des arguments : `lpr fichier1 fichier2`
- des options avec des arguments `m ke -f mym kefile`

Comment faire en OCaml qu'un programme compilé puisse « voir » ses options ?

Solution 1 : accéder directement la ligne de commande

- Module `Sys` : Interface système, mais indépendant du système d'exploitation (ne va donc pas très loin).
- Valeur `Sys. rgv` du type `string rr y` : tableau qui contient les éléments de la ligne de commande (la commande à la position 0, puis les arguments éventuels).
- `Arr y.length Sys. rgv` : nombre d'éléments (la commande elle-même incluse).
- `Sys. rgv.(i)` : élément numéro *i* de la ligne de commande (attention, commence sur 0)
- Pour en savoir plus : voir les Modules `Sys` et `Arr y`.

Exemples (args.ml)

```
let number_args = Array.length Sys.argv;;

for i=0 to number_args-1 do (* attention, decalage *)
  print_string "Element numero ";
  print_int i;
  print_string " : ";
  print_string Sys.argv.(i);
  print_newline ();
done;;
```


Solution 2 : utiliser le module `Arg`

- On donne une spécification des options possibles, avec leurs arguments éventuels, et l'action à exécuter.
- Puis, la fonction `p rse_ rgv` s'occupe d'interpréter les éléments de la ligne de commande.
- Avantage : on n'a pas besoin de s'occuper des types des arguments des options, de leur conversion, l'ordre des arguments, etc.
- Affichage d'un message d'aide.
- Voir le manuel : Module `Arg`

```
open Arg;;

let switch = ref false;; (* mettre les valeurs de defaut
let temperature = ref 0;;
let nom = ref None;;
let arguments = ref [];;

let opt_spec = [ (* specification des options *)
  ("-sw", Unit (function () -> switch := not !switch), "
  ("-t", Set_int temperature, "mettre une temperature")
  ("-n", String (function s -> nom := Some s), "mettre
];;

let arg_action = (* quoi faire avec des autres arguments
  (function s -> arguments := !arguments @ [s]);;

let usage = (* Entete du message d'aide *)
  "Demonstration du module Arg";;
```



```
(* interpreter les options et arguments *)
parse opt_spec arg_action usage;;

(* detecter des problemes *)
if !nom=None then begin
  prerr_endline "Erreur : pas de nom donne";
  exit 0
end;;

(* action normale du programme *)
print_string "Le switch ";;
print_string (if !switch then "est" else "n'est pas");;
print_string " mis.";;
print_newline();;
```


Modules comme unités de compilation

- Un module est une unité de programme qui regroupe des définitions (définitions de quoi ? ça dépend du langage).
  - OCaml: types, valeurs (aussi de type fonctionnel), exceptions, ...
  - Pascal: constantes, variables, fonctions, procédures, ...
- Pour nous : Module = Unité de compilation.
- Relation entre modules : Dépendance. Module A dépend du module B ssi A utilise un nom défini par B.
- Le module B exporte quelque chose (par ex. une fonction), le module A l'importe.

Modules et compilation séparée

- Si module A dépend du module B, alors la compilation de A a besoin de connaître les définitions effectuées par B :
  - soit textuellement (le compilateur regarde dans le code source de B)
  - soit en forme compilée (cas de OCaml) $\Rightarrow$ compiler B avant de compiler A.
- Dans le deuxième cas, la compilation d'un module A engendre
  - un « résumé » des définitions effectuées par A
  - du code, avec des adresses symboliques pour les identificateurs définis dans des autres modules.
- À la fin : assemblage des morceaux de code et résolution des symboles (édition des liens, angl.: linking).
- La relation de dépendance doit être acyclique.

Interface et corps d'un module

- Interface: Résumé des définitions d'un module
- Corps (Implantation): Code réalisant les définitions

Dans le cas de OCaml :

- Un fichier interface `.mli`
- Un fichier corps `.ml`.

Avantage: Si A dépend de B, alors on fixe d'abord l'interface de B. Puis, on peut rédiger (et même compiler) les corps de A et B indépendamment.

Exemples (exfact.mli)

```
(* (fact n) renvoie la factorielle de n quand *)
(* n est non negatif; leve l'exception        *)
(* Argument_negatif quand n est negatif.      *)

exception Argument_negatif;
val fact: int -> int
```


Exemples (exfact.ml)

```
exception Argument_negatif;;

let rec fact_nn n =
  assert (n >=0);
  if n <=1 then 1 else n*fact_nn(n-1);;

let fact n =
  if n<0
  then raise Argument_negatif
  else fact_nn n;;
```


Importation dans les modules de OCaml

Deux constructions:

1. Directive `open B` au début du module importeur A : fait accessible en `A` toutes les définitions exportées par `B`.
   Avantage: plus court.
2. Préfixer les noms par le nom du module: `B.f` dénote l'identificateur f exportée par le module B.
   Avantage: plus explicite, et pas d'ambiguïté (plusieurs modules peuvent exporter le même identificateur).

Dépendance d'une interface d'un module d'un autre module

L'interface d'un module A peut dépendre d'un module B, c'est précisément le cas quand A exporte un type concret dont la définition utilise un type exporté par B.
Exemple: Le module `Expressions` exporte un type `expr`, et l'interface du module `Instruction` contient

```
type instr =
  Print of Expression.expr
| Affect of string * Expression.expr
| While of Expression.expr * instr list
```


Compilation des modules en OCaml

- `oc mlc module.mli` produit `module.cmi` à partir de `module.mli`
- `oc mlc -c module.ml` produit `module.cmo` à partir de `module.ml`
- `oc mlc -o progr m module_1.cmo module_2.cmo ... module_n.cmo` fait l'édition des liens et crée l'exécutable `progr m`.
  Il ne faut pas que `module_i` dépend de `module_j` pour $i < j$.
- `oc mlc -i prog.ml` donne sur stdout une interface du module `prog.ml` qui exporte tout. Peut à la rigueur servir comme base d'une vraie interface, mais ne remplace pas une interface bien pensée !

Pourquoi ne serait ce pas une bonne interface ?

- Pas de commentaire (l'interface doit contenir une spécification des valeurs etc. définies sous forme de commentaire).
- Souvent on ne veut pas exporter toutes les définitions d'un module. Il n'y a pas d'encapsulation.

Le principe d'encapsulation

Exporter aussi peu de définitions que possible. On fait, l'interface d'un module peut être plus abstraite que son corps. Cacher les fonctions, types, exceptions auxiliaires. En OCaml :

- Une interface peut exporter un type abstrait : L'interface ne contient que la déclaration `type t`, et le corps contient sa définition complète :
  `type t = ...`
- Si un type concret est exporté par l'interface, alors le corps doit contenir la même définition.
- Tout identificateur (ou exception) exporté doit être défini par le corps, et cela avec un type égal ou plus général, éventuellement en utilisant les définitions des types abstraits.

Le corps peut contenir des types, fonctions, exceptions privés (pas exportés).

Intérêt de l'encapsulation

- L'interface comme « contrat » entre programmeur d'un module et son utilisateur : liberté de réalisation au programmeur.
  L'interface contient toutes les informations qui sont nécessaires pour utiliser le module (avec des commentaires !!!).
- Le codage d'un module peut être changé sans que cette modification ne soit visible de l'extérieur.
- Types abstrait: maintenance d'un invariant puisque modifications que par des fonctions exportées.

Encapsulation dans les langages de programmation

- Portée locales des définition, par exemple une fonction auxiliaire qui est définie à l'intérieur d'une autre fonction.
- Modules, avec exportation sélective.
- Objets avec des méthodes privées etc.
- C'est un outil essentiel pour maîtriser la complexité d'un programme (rédaction, relecture, et maintenance).

Exemple

- Un module pour des tours d'entier (des piles d'entiers, où la valeur décroit vers le sommet, comme pour les Tours de Hanoï).
- On ne veut permettre que la construction des tours qui satisfont l'invariant : Les valeurs décroissent vers le sommet.
- Solution : Déclarer un type abstrait, et ne permettre la modification d'un tour que par une fonction exportée par le module.


```
(* interface of the module for towers of integers. A tow
   integers which are strictly decreasing from bottom to

type tower
exception Operation_illegal
  (* the empty tower *)
val empty: tower
(* (push i t) returns a new tower consisting of t with a
   the top, provided that result is still a tower. Other
   Operation_illegal.*)
val push: int -> tower -> tower
(* (pop t) returns a tower which consists of t without i
   raises Operation_illegal when t is empty. *)
val pop: tower -> tower
(* (top t) returns the top element of t, raises Operatio
   t is empty. *)
val top: tower -> int
```

```
(* implementation of module tour *)
type tower = int list
exception Operation_illegal

let empty = []
let top = function
    h::r -> h
  | [] -> raise Operation_illegal
(* (can_be_pushed i t) is true iff i can be pushed on t
let can_be_pushed i = function
    [] -> true
  | h::r -> i h
let push i t =
  if can_be_pushed i t then i::t else raise Operation_i
let pop = function
    h::r -> r
  | [] -> raise Operation_illegal
```

Programmation Fonctionnelle Cours 9 Compilation et Modules
└Compilation d'un programme découpé en plusieurs modules

```
open Tour

let a = empty;;
let b = pop (push 17 (push 42 a));;
print_int (top b);;
print_newline ();;
```

Programmation Fonctionnelle Cours 9 Compilation et Modules
└Compilation d'un programme découpé en plusieurs modules

```
let a = Tour.empty;;
let b = Tour.pop (Tour.push 17 (Tour.push 42a));;
print_int (Tour.top b);;
print_newline ();;
```

Programmation Fonctionnelle Cours 9 Compilation et Modules
└Compilation d'un programme découpé en plusieurs modules

Ordre de compilation sur l'exemple

`oc mlc tour.mli`	compiler l'interface `tour.mli`
`oc mlc -c tour.ml`	compiler le corps `tour.ml`
`oc mlc -c m in.ml`	compiler le corps `m in.ml`
`oc mlc -o m in tour.cmo m in.cmo`	édition de liens

Cas particuliers

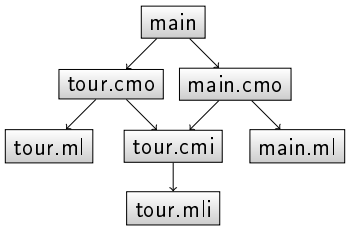
- Attention, l'utilisation de certains modules de la bibliothèques nécessitent des options supplémentaires lors de l'édition de liens.
- C'est en particulier le cas pour les modules `Gr phics`, `Str`, `Unix`.
- Voir le manuel pour connaître les options nécessaires.
- Dans le cas du module `Gr phics` :

```
oc mlc gr phics.cm  module_1.cmo ... module_n.cmo
```


Comment automatiser le processus de compilation

- Problème : quand on change le code source d'un module il faut recompiler (dans le bon ordre) ce module et les modules qui en dépendent, plus refaire l'exécutable.
- Première solution : un script shell (un programme) qui compile tout.
- Inconvénient : Solution ad-hoc, pas de ré-compilation partielle.
- Mieux : Utiliser des outils plus intelligents.

Dépendances dans la création des fichiers



- Avant de créer un fichier il faut s'assurer que toutes ses dépendances existent.
- Ce graphe doit être acyclique.

Compilation automatique avec make

Un outil universel

`m ke` est un outil de développement sous UNIX, il peut aussi être utilisé pour compiler du C, C++, Java, créer des transparents à partir d'un description texte, ...

Principe de base:

- Règles qui décrivent des dépendances entre des cibles
- un cible est un nom de fichier, ou un nom symbolique (qui doit être déclaré comme *phony*)
- Avec chaque règle : instructions (commandes shell) pour mettre à jour un cible.

Le Makefile sur l'exemple

```
# Édition des liens et création de l'exécutable
main: tour.cmo main.cmo
        ocamlc -o main tour.cmo main.cmo

# Compilation du corps du module tour
tour.cmo: tour.ml tour.cmi
        ocamlc -c tour.ml

# Compilation de l'interface du module tour
tour.cmi: tour.mli
        ocamlc tour.mli

# Compilation du corps du module main
main.cmo: main.ml tour.cmi
        ocamlc -c main.ml
```


Utilisation du Makefile

- Le cas le plus simple : `make`
- Essaye d'assurer que tous les cibles existent, et que chaque cible est plus nouveau que toutes ces dépendances.
- Quand `make` est lancé après mise-à-jour d'un fichier : re-génération seulement des cibles qui ne sont plus à jour.
- Il y a beaucoup à savoir sur `make`, voir le cours *Environnements et outils de développement*.

ocamlbuild

- Inconvénient de `make` : il faut écrire un `Makefile` avec les bonnes dépendances.
- Il y a des outils et des mécanismes de `make` qui aident à cette tâche.
- Dans le de *OCaml* : il existe un outil specialisé qui dans des cas simples gère lui seul toute la compilation, et qui est très configurable pour des cas complexes.
- Ocamlbuild fait partie de la distribution standard de OCaml.

Utilisation simple

- `ocamlbuild` *main*`.native` ou `ocamlbuild` *main*`.byte`, où *main* est à remplacer par le nom du module principal.
- Crée tous les .cmi, .cmo, etc. dans un répertoire `_build`
- Crée l'exécutable en tant que lien symbolique vers `_build`
- `ocamlbuild -clean` pour nettoyer le répertoire.

Gérer des bibliothèques externes

- Il existe un système du nom `findlib` pour gérer les bibliothèques OCaml.
- Pour l'utiliser avec `oc mlbuild` :
  - Créer un fichier du nom `_tags` qui spécifie des options supplémentaires au compilateur.
  - Appeler comme `ocamlbuild -use-ocamlfind main.native`
  - Dans le cas d'un projet avec camlimages : Le fichier `_tags` consiste en une seule ligne:

```
<*>: package(camlimages.all_formats), package(camlimages.graphics)
```


Analyse descendante

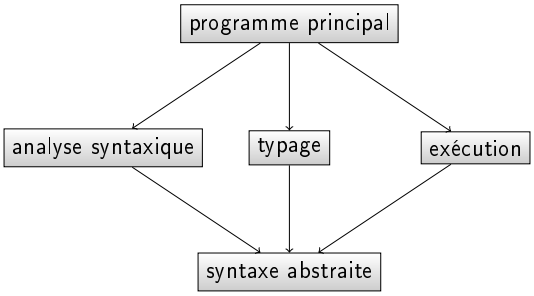
- (ou: Comment trouver le bon découpage en modules ?)
- Analyse du plus général vers le plus particulier.
- Commencer avec le programme entier : quelle est l'entrée, quelle est le résultat ?
- Puis, couper le fonctionnement du programme en sous tâches. Identifier les fonctionnalités principales de la sous tâche ($\Rightarrow$ fonctions exportées), les types de données et des fonctionnalités partagées par les sous tâches ($\Rightarrow$ modules auxiliaires utilisés par des autres modules).
- Il est utile d'avoir une idée grossière de l'implémentation des modules.

Exemple

- Un interpréteur pour un langage de programmation avec déclaration des variables types (par ex. Pascal).
- Fonctionnalité principale: lit un programme, déclenche une erreur si le programme n'est pas correcte, sinon l'exécute et affiche le résultat.
- Découpage en sous tâches:
  1. Analyse syntaxique
  2. Vérification des types et déclarations des variables
  3. Exécution

  Les trois modules travaillent sur la syntaxe abstraite des programmes.

Graphe de dépendances (première version)



Interface du module Syntaxe

La syntaxe abstraite est un type inductif, pas de raison de cacher sa définition.

```
type typ = Entier | Réel | Bool
type expr = Var of string
          | EConst of int
          | Plus of expr * expr
          ...
type instr = Affect of string * expr
           | While of expr * instr list
           ...
type decl = string * typ
type prog = decl list * instr list
```


Interfaces

Interface du module typage :

```
v l bien_typé : Synt xe.prog -> bool
```

Interface du module exécution :

```
exception: Division_p r_zero
v l: exec: Synt xe.prog -> string
```


Fonctions privées de ces modules

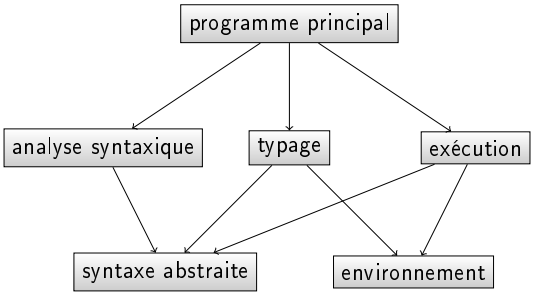
- module typage : typ_expr: typenv -> Syntaxe.expr -> Syntaxe.typ
- module exécution : eval_expr: valenv -> Syntaxe.expr -> valeur

où

- typenv : Type des environnements de types
- valenv : Type des environnements de valeurs

Le type valeur doit être un type privé du module exécution (type somme regroupant int, real, etc.)
Par contre les deux types d'environnements doivent être généralisés : un type polymorphe d'environnement, défini dans un module à part.

Graphe de dépendances (deuxième version)



Interface du module Environnements

```
(* type polymorphes des environnements *)
type 'a env
```

(plus des spécifications des exceptions et fonctions).
Est-ce que le module Environnements doit dépendre du module Syntaxe (qui défini le type `typ`) quand on veut construire des environnements de typage ?

Remarques

- Cas caricaturales à éviter:
  - Un seul module pour le programme entier
  - Un module par définition de type ou fonction
  - Découpage arbitraire: un module pour tous les types, un autre pour toutes les fonctions, etc.
- Bon découpage:
  - Correspond à la logique du programme
  - Modules d'une taille raisonnable
  - Définition auxiliaires cachées dans les modules (l'organisation en modules simplifie la structure du programme)
  - Réutilisation du code au lieu de duplication

Quelques mots sur la documentation

- Documenter la structure globale du programme (graphe de dépendance)
- Interfaces des modules: Documenter l'utilisation du module :
  - Son rôle général
  - Que représentent les types ?
  - Spécifier les fonctions : Expliquer les rôles des arguments, les hypothèses sur leurs valeurs (par ex: entier positif, liste triée, etc.), et bien sûr le résultat. N'oubliez pas les cas d'erreur.
- En général: La doc de l'interface doit contenir toutes les informations nécessaires pour l'utilisation du module.
- Corps des modules :
  - Spécifier les fonctions privées.
  - Expliquer l'algorithme utilisée quand pas évident.
  - Donner des invariants des fonctions (utiliser des construction du langage (assertions si possible)

Le langage des modules en OCaml

En vérité : Un module n'est pas toujours une unité de compilation. Il y a en OCaml des constructions pour définir des modules (à part des modules définis implicitement par unité de compilation). Cela permet par exemple de :

- définir un module avec plusieurs interfaces
- définir des modules qui sont paramétrés par des modules (par exemples des tables de hachage)

```
module Nom = struct
   <définitions des types, valeurs, et exceptions>
end
```

Pour en savoir plus : voir le manuel OCaml.