λ

Programmation Fonctionnelle: Le premier ordre

Yann Régis-Gianas
(IRIF, Univ. Paris Diderot, Inria) – yrg@irif.fr

2019-09-20

Rappel du cours précédent

Programmation fonctionnelle = pureté + induction + fonction

Plan

Développer en OCaml

Les types de données que vous connaissez déjà

OCaml pour développer un logiciel

- `OCaml` structure ses programmes en **modules**.
- La bibliothèque standard d'`OCaml` est minimaliste.
- Elle est documentée dans le manuel officiel : `http//www.ocaml.org/`.
- Des bibliothèques externes sont disponibles grâce à `findlib` et `dune`.
- Des outils de développement sont disponibles : `alcotest`, `quickcheck`, `odoc`, `fuzzer`, ...

Les modules `OCaml` définis par des fichiers

Fichiers d'implémentation et d'interface

Le fichier `sayHello.ml` définit le module `SayHello`.
Le fichier `sayHello.mli` définit l'interface du module `SayHello`.

Observons le code de :

`https://gaufre.informatique.univ-paris-diderot.fr/yrg/pf5/cours/cours-02/code/simple/`.

Le module `Hello` utilise le module `SayHello`.

Compilation en code-octet (bytecode)

Pour compiler ce petit programme, il faut :

```
$ ocamlc -c sayHello.ml  # Compiler l'implémentation de SayHello.
# Le fichier sayHello.cmo est produit.
$ ocamlc -c sayHello.mli # Compiler l'interface de SayHello.
# Le fichier sayHello.cmi est produit.
$ ocamlc -c hello.ml     # Compiler l'implémentation de Hello.
# Les fichiers hello.cmo et hello.cmi sont produits.
# (Quand le fichier .mli est absent, une interface est
# automatiquement produite et elle dévoile tout!)
$ ocamlc -o hello sayHello.cmo hello.cmo
$ # Pour lier les modules en un exécutable.
```

Extensions

- `.cmo` : implémentation de module compilé en code-octet.
- `.cmi` : interface de module compilé.

Compilation en code natif

Le code natif est environ 8 fois plus rapide que le code-octet mais non portable.

Pour compiler ce petit programme, il faut :

```
$ ocamlopt -c sayHello.ml  # Compiler l'implémentation de SayHello.
# Le fichier sayHello.cmx est produit.
$ ocamlopt -c sayHello.mli # Compiler l'interface de SayHello.
# Le fichier sayHello.cmi est produit.
$ ocamlopt -c hello.ml     # Compiler l'implémentation de Hello.
# Les fichiers hello.cmx et hello.cmi sont produits.
# (Quand le fichier .mli est absent, une interface est
# automatiquement produite et elle dévoile tout!)
$ ocamlopt -o hello sayHello.cmx hello.cmx
$ # Pour lier les modules en un exécutable.
```

Extensions

- `.cmx` : implémentation de module compilé en natif.
- `.cmi` : interface de module compilé.

Comment utiliser un module?

- `SayHello.say_it ()` fait appel à la fonction `say_it` du module `SayHello`.
- Pour donner accès aux définitions d'un module sans avoir besoin de préciser le nom du module à chaque accès :

```
1  open SayHello
2  let main = say_it ()
```

Bibliothèque standard d'`OCaml` (1/2)

- Un module nommé **`Pervasives`** est accessible automatiquement par tout programme `OCaml`. Il fournit des exceptions standards, des opérateurs de comparaisons et booléens, de compositions, de calculs arithmétiques, de conversions, d'entrées/sorties, ...

  `https://caml.inria.fr/pub/docs/manual-ocaml-4.08/core.html`

- La bibliothèque standard fournit les utilitaires et les structures de données classiques: manipulation de chaînes de caractères ; dictionnaires ; ensembles ; structures de séquence, liste, file et de pile ; interactions via le terminal ; prise en charge des arguments de la ligne de commande ; primitives cryptographiques ...

  `https://caml.inria.fr/pub/docs/manual-ocaml-4.08/stdlib.html`

Bibliothèque standard d'`OCaml` (2/2)

- Cette bibliothèque standard est complétée par une bibliothèque pour la programmation système, nommé **`Unix`**. On y trouve les appels systèmes standards de **`POSIX`**. Ces appels sont en grande partie émulée sous Windows.

  `https://caml.inria.fr/pub/docs/manual-ocaml-4.09/libunix.html`

- D'autres bibliothèques importantes sont inclues dans la distribution:
  - **`Num`** : pour faire du calcul arithmétique en précision arbitraire.
  - **`Str`** : pour utiliser des expressions régulières.
  - **`Threads`** : pour utiliser des fils d'exécution (système).
  - **`Bigarray`** : pour travailler sur de grandes quantités de données.

Bibliothèques

- Les bibliothèques sont des ensembles de modules.
- Elles viennent sous trois formats distincts:
  - Bibliothèques statiques code-octets : `.cma`
  - Bibliothèques statiques natives : `.cmxa`
  - Bibliothèques dynamiques natives : `.cmxs`
- Pour utiliser une bibliothèque, il suffit de préciser le fichier de la bibliothèque lors de l'édition des liens.

```
1 $ ocamlopt -o mywebserver unix.cma webserver.cmo
```

- Le fichier `unix.cma` est situé dans le répertoire standard `lib/ocaml` installé par la distribution. Pour connaître la configuration de votre compilateur `OCaml`:

```
1 $ ocamlc -config
```

- Si la bibliothèque ne se trouve pas dans un emplacement standard, on peut spécifier cette emplacement via l'option `-I`:

```
1 $ ocamlc -o myvideogame -I mylib renderengine.cma mainlogic.cmo
```

L'outil `ocamlfind`

L'outil standard **`ocamlfind`** permet d'associer des informations à un nom de bibliothèque. Cette fonctionnalité permet de simplifier les commandes de compilation:

```
$ ocamlfind query re # Quel est l'emplacement de la bibliothèque 're'?
/home/yann/.opam/4.05.0/lib/re
```

Mieux, **`ocamlfind`** sait réécrire pour vous les lignes de commande de compilation:

```
$ ocamlfind ocamlc -o hello -package re sayHello.cmo hello.cmo
```

rajoute les bonnes options pour inclure la bibliothèque **`re`** dans l'exécutable **`hello`**.

L'outil **dune**

dune est un outil qui prend en charge la compilation de programmes **OCaml**. Il suffit de décrire ce que l'on veut faire dans un fichier **dune** à la racine de l'arbre de sources :

```
1  (executable (name hello))
```

puis d'exécuter la commande:

```
1  $ dune build hello.exe
```

pour que **dune** calcule automatiquement les dépendances du projet et lance les bonnes commandes de compilation!

Nous utiliserons cet outil pour développer le projet.

Pour plus d'informations: `http://dune.readthedocs.io`

Autres éléments utiles pour le développement logiciel

Une prochaine fois, nous verrons:

- Comment produire de la documentation à l'aide de `odoc`.
- Comment faire des tests unitaires.
- Quelles sont les bibliothèques utiles aux développements collaboratifs.
- Comment produire du code client web à partir de code `OCaml`.
- Comment écrire ses propres bibliothèques.
- Comment distribuer ses paquets logiciels.
- Comment utiliser des extensions du langage.

Plan

Retournons à l'apprentissage du langage!

Ce que vous connaissez déjà

Nous allons rapidement faire le tour des constructions permettant de définir des calculs manipulant des données avec lesquelles vous êtes déjà familiers dans le notebook suivant:

`https://sketch.sh/s/l3N96HVMsM3eGQQw9JQ6y8/`

A retenir : Les types du premier ordre

Nous venons de présenter:

- les booléens ;
- les entiers (qui sont signés et sur 63 bits) ;
- les flottants (qui respectent la norme IEEE) ;
- les caractères (qui sont sur 8 bits) ;
- les n-uplets ;
- les enregistrements ;
- les énumérations ;
- les tableaux (dont le type est “paramétrique”);
- les listes (dont le type est aussi paramétrique).

A retenir : Les types du premier ordre

Nous venons de présenter:

- les booléens ;
- les entiers (qui sont signés et sur 63 bits) ;
- les flottants (qui respectent la norme IEEE) ;
- les caractères (qui sont sur 8 bits) ;
- les n-uplets ;
- les enregistrements ;
- les énumérations ;
- les tableaux (dont le type est “paramétrique”);
- les listes (dont le type est aussi paramétrique).

Ces constructions sont dites du premier ordre car elles ne contiennent pas de **code**.

A retenir : Les types du premier ordre

Nous venons de présenter:

- les booléens ;
- les entiers (qui sont signés et sur 63 bits) ;
- les flottants (qui respectent la norme IEEE) ;
- les caractères (qui sont sur 8 bits) ;
- les n-uplets ;
- les enregistrements ;
- les énumérations ;
- les tableaux (dont le type est "paramétrique");
- les listes (dont le type est aussi paramétrique).

Ces constructions sont dites du premier ordre car elles ne contiennent pas de **code**.

D'ailleurs, on peut les **sérialiser** facilement grâce au module `Marshal`.