Sommaire

Le language Ocaml......1
Introduction......1
L'implémentation d'Ocaml......1
quelques exemples de code Ocaml......1
Poursuivre sur Ocaml......4

Copyright......5

Le language Ocaml

Introduction

Le language Ocaml est développé en France par une équipe de chercheurs de l'INRIA, dont la plupart utilisent (et contribuent activement) à Linux et au logiciel libre. Il est utilisé par une communauté croissante, aussi bien dans le monde académique (de nombreux chercheurs prototypent leurs idées en Ocaml) ou universitaire (son prédécesseur Caml est enseigné en classe prépa, premier cycle) que dans le monde industriel (dans des petites startup comme dans des grosses multinationales). Ocaml est language fonctionnel et à objets, avec un typage statique (et inférence de types). À cet égard il est assez différent des languages habituels sous Linux (comme C, C++, Perl, Python). Il existe plusieurs grosses applications développées en Ocaml −aussi bien libres que propriétaires−. Une application libre potentiellement utile à beaucoup est **Unison** pour synchroniser (càd copier) des fichiers entres machines différentes, disponible ici. Il y aussi **Advi** (visualiseur pour fichiers dvi produits par LaTeX) qui permet notamment de faire de beaux transparents (y compris avec des formules de mathématiques, de l'animation, des fenêtres de démo de logiciels) disponible ici, le paquet Whizzytex (utilisant advi) pour visualiser immédiatement en temps quasi réel un document LaTeX pendant son édition et HeVeA, un convertisseur de LaTeX vers le html.

L'implémentation d'Ocaml

Le language Ocaml a sa page ici. Son implémentation est libre. Il y a en réalité plusieurs variantes d'implementation (partageant du code en commun):

l'interprète interactif dit *toplevel*, par la commande **ocaml**; le principal intérêt d' **ocaml** est de tester **interactivement** une (ou quelques) nouvelles fonctions (éventuellement rajoutées à un gros programme existant); l'interprète interactif compile chaque phrase tapée en du code octet, puis l'execute. Il y a aussi un débogueur **ocamldebug**

le compilateur vers du code−octet *bytecode compiler*, par la commande **ocamlc** (le code−octet produit a sa machine virtuelle propre **ocamlrun**; ce code octet est différent de ceux de Java, Lua ou Parrot).

le compilateur natif (sur AMD64, x86, PowerPC, Sparc etc...) *native compiler*, par la commande **ocamlopt**. Les programmes compilés avec ocamlopt s'exécutent à peu près aussi rapidement que leur équivalent C ou C++.

Il existe une communauté grandissante de développeurs Ocaml. Le **Ocaml Hump** ici contient une collection croissante de logiciels et bibliothèques libres codé(e)s en Ocaml; on peut développer des applications Web (avec Wdialog par exemple), interfacer des SGBDs, invoquer des appels systèmes etc... Ocaml est très puissant (et originellement développé) pour le traitement symbolique de l'information: compilateurs, démonstrateurs de théorèmes, etc..

quelques exemples de code Ocaml

Bien sûr, le hello world:

```
(* hello world en Ocaml *)
print_string "hello world\n";;
```

Les commentaires sont à la Pascal entre (* et *) et peuvent être imbriqués. Ce fichier peut–être édité (par exemple avec Emacs ou Xemacs en mode tuareg) puis sauvegardé dans hello.ml

Les arguments d'une fonction (ici print_string) suivent le nom de la fonction, sans parenthèses. Un certain nombre de fonctions (en fait de valeurs fonctionnelles) sont prédéfinies, de sorte qu'on n'a pas besoin (au contraire de C) de requérir explicitement l'accès à la bibliothèque standard d'Ocaml.

On peut sauvegarder ce fichier de 2 lignes dans hello.ml puis le compiler par ocamlc hello.ml –o hello qui produit un binaire de code octet (portable tel quel sur beaucoup d'architectures). On aurait pu le compiler avec le compilateur natif: ocamlopt hello.ml –o hello

on peut aussi utiliser l'interprète interactif (toplevel); on peu taper les phrases sur plusieurs lignes, mais le double point–virgule est indispensable pour terminer la phrase. Les entrées de l'utilisateur sont ici en italiques; tout le reste (y compris le % et le #) sont sortis par la machine.

```
% <i>ocaml</i>
  Objective Caml version 3.09.1

#  2 + 3;;
- : int = 5
```

Ici apparait un trait essentiel du langage Ocaml: l'**inférence de types** (ou synthèse de type). Chaque expression entrée (ou compilée) est analysée par le compilateur qui devine (infère) son type. Le compilateur donne le type inféré (ici le type entier *int* sur 31 bits –un bit de moins que la taille du mot machine), calcule la valeur de l'expression, puis l'imprime (ici *5*). L'inférence de types prend tout son intérêt sur des programmes non triviaux (plus d'une centaine de lignes) dont elle permet la concision (en évitant la plupart des déclarations de type, comme on le fait en C, C++, Java...).

Sous le toplevel, on peut demander la compilation (interactive) d'un fichier, par la directive #use:

```
# #use "hello.ml";;
hello world
- : unit = ()
```

le fichier hello.ml a été compilé puis exécuté, donc le message hello word est imprimé. Toutes les expressions Ocaml ont une valeur (à l'inverse de C ou Pascal, où procedures et fonctions sont différentes), et les fonctions qui n'ont qu'un effet de bord (par exemple une impression) renvoie une valeur de type *unit* – cette valeur (la seule du type unit) est notée ()

les fonctions sont des valeurs comme les autres: Ocaml est un langage fonctionnel, et des fonctions peuvent prendre comme arguments ou renvoyer comme résultat d'autres fonctions. Pour interroger le type d'une valeur, il suffit de la taper; voici comment demander la fonction print_string:

```
# print_string;;
- : string -> unit = < fun >
```

La valeur d'une fonction n'est pas affichée (c'est techniquement une fermeture, c'est à dire la juxtaposition de code executable et des valeurs qu'il utilise); Ocaml affiche donc le type string –> unit de la fonction print_string: c'est une fonction qui prend un argument chaine et renvoie unit.

On peut définir des valeurs par le mot–clé **let**. Par exemple

```
# let dix = 5*2;;
val dix : int = 10
```

définit une valeur nommée dix de type int valant 10

On peut définir des valeurs fonctionnelles (c'est à dire des fonctions) par exemple

```
# let double x = 2*x;;
val double : int -> int = &lt;fun&gt;
```

Le compilateur a inféré le type de la fonction double: son argument x est nécessairement entier, car il est multiplié par 2 et que la multiplication est une opération sur les entiers; son résultat est entier, car c'est le résultat produit par la multiplication. Une fois compilé par ocamlopt, la fonction ci–dessus est aussi efficace que celle écrite en C!

On peut définir des fonctions récursives par la construction **let rec** – voici l'inévitable factorielle et le calcul de 10!

```
# <i>let rec fact n = if n < 1 then 1 else n * fact (n-1);;</i>
val fact : int -> int = < fun >
# ''fact 10;;''
- : int = 3628800
```

Voici l'une des plus simples des recursivités croisées (le mot–clé and lie plusieurs définitions) – les fonctions paire et impaire

```
# <i>let rec paire n = if n = 0 then true else impaire (n-1)
and impaire n = if n = 0 then false else paire (n-1) ;;</i>
val paire : int -> bool = &lt;fun&gt;
val impaire : int -> bool = &lt;fun&gt;
```

le compilateur tient compte du fait que ces recursions sont terminales; l'appel recursif à impaire dans la définition de pair (et réciproquement) est la dernière application de fonction, et il est interprété itérativement (sans grossir la pile des appels). On peut donc calculer

```
# <i>paire 123456789;;</i>
- : bool = false
```

le résultat est calculé en quelques secondes, la machine a bouclé plus d'un milliard de fois. Les compilateurs classiques (par exemple C, C++, Java) aurait fait exploser la pile d'appel.

En fait on peut réécrire la factorielle de manière récursive terminale en définissant une fonction interne à 2 arguments: le résultat partiel (produit précédemment calculé p) et l'argument n

```
#'<i>let fact n =
  let rec interne_fact p n = if n < 1 then p else interne_fact (p*n) (n-1)
  in interne_fact 1 n;;</i>
val fact : int -> int = &lt;fun&gt;
```

On peut dans une expression définir des noms **internes** par la construction let *...définitions internes...* in *....corps.....* (ou let rec ... in .... pour une recursion).

la factorielle ainsi redefinie est compilée en du code identique à la fonction C *int factc(int n) { int p=1; while(n>1) p=p*(n−−);return p; }*

La plupart des boucles s'écrivent en fait par une telle récursion terminale interne. Il faut noter que les variables dans tous les exemples ci–dessus sont des paramètres qui ne changent pas: on a rarement besoin d'affectation en Ocaml.

En fait on peut aussi coder de manière **impérative**, avec des variables qui varient (comme en C ou Java), qu'on appelle référence:

```
# <i>let fact n = let p = ref 1 and c = ref 1 in
   ''  while !c <= n do p := !p * !c  ;  incr c done; !p;; </i>
```

l'affectation à une référence se note := et la déréference est explicitée par un point d'exclamation. La fonction incr incrémente une réference vers un entier

(((à suivre)))

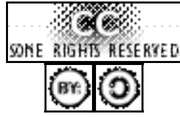
Poursuivre sur Ocaml

Convaincus ? Voici quelques liens qui pourront vous aider dans l'appentissage de ce langage :

- Pour ceux qui connaissent d'autres langages, les articles Wikipédia pour différencier programmation fonctionnelle et programmation impérative ;
- un cours de Caml à l'usage des débutants en programmation (français). Consulter également la page pour des cours plus avancés (anglais/français) ;
- le manuel de référence d'Ocaml (anglais) ;
- la version en ligne du livre Développement d'applications avec Objective Caml (français, autrement disponible aux éditions O'Reilly).

Copyright

© 9/11/2005 *Basiles*



Ce document est publié sous licence Creative Commons Attribution, Partage à l'identique 2.0 : http://creativecommons.org/licenses/by−sa/2.0/