Premiers pas avec OCaml

Samuel Mimram

4 février 2006

1 Généralités

Vous pourrez trouver les énoncés et les corrigés des TDs sur la page web `http://perso.ens-lyon.fr/samuel.mimram/tdcaml.html`. N'hésitez pas à m'envoyer un mail en cas de besoin ou si quelque chose ne vous parait pas clair ou si vous avez du mal à installer caml chez vous : `samuel.mimram@ens-lyon.org`.

Il faut se logger sur les machines sous l'utilisateur `guest`, il n'y a pas de mot de passe. Lancez XEmacs par le menu (attention c'est long à démarrer !). Les fichiers doivent être enregistrés dans le répertoire `network/05-HX` dans un fichier `votre_nom.ml`.

Petits rappels sur Emacs

- première chose à faire, ouvrir un fichier : `C-x C-f` (`C` = ctrl)
- sauvez régulièrement : `C-x C-s`
- évaluez votre code caml en appuyant sur `F12`
- on rappelle que pour faire du copier-coller, il suffit de sélectionner avec le bouton gauche de la souris puis de coller en utilisant le bouton du milieu

Ce TD est plutôt long. N'attaquez la 3^{e} partie que si vous avez bien compris les deux premières.

2 Premières expressions

2.1 Jouons avec les types

▶ Entrez votre première expression :

```
22 + 47;;
```

Quelle est la réponse de caml ? Que cela signifie-t-il ?

▶ Entrez maintenant l'expression suivante (notez bien qu'il y a des points) :

```
1111. +. 553.;;
```

Quelle est maintenant la réponse de caml ? En quoi les points ont-ils changé quelque chose ?

Testez aussi les expressions suivantes :

```
false;;
'x';;
"toto";;
```

▶ En quoi le code suivant pose-t-il problème ? À quoi servent ces vérifications de type ?

```
let x = 5.;;
x + 2;;
```

Existe-t-il des fonctions pour convertir des flottants en entiers ?

▶ Définissez la fonction `double` qui prend en argument un entier et renvoie le double. Quel est son type ? Pourquoi caml affiche-t-il `<fun>` et non la fonction ? Calculez

```
double 2;;
double 2.;;
double "toto";;
```

En quoi les erreurs que vous obtenez sont-elles en rapport avec le type de `double` ?

▶ Définissez la fonction identité (la fonction `id` telle que $\forall x, \texttt{id}(x) = x$). Quel est son type ? Quel est à votre avis le résultat de `id id` ?

▶ Quel est le type de la fonction suivante ? En quoi est-elle différente de la précédente ?

```
let f x = if true then x else 0;;
```

(★) Si `f` et `g` sont deux fonctions caml quelconques, savez vous s'il est possible de savoir si $\forall x, \texttt{f}\ x = \texttt{g}\ x$? La question cache une subtilité, en effet, si `f` est une fonction caml, `f` x renvoie-t-elle toujours un résultat ? Avez-vous déjà entendu parler du « problème de l'arrêt » ?

▶ Si g est une fonction à un argument et f est une fonction à deux arguments, on peut calculer $f(gx)(gx)$. Par exemple si f est la fonction telle que $fxy = x + y$ et g est la fonction `double` alors pour un entier x, $f(gx)(gx)$ vaudra $(2 * x) + (2 * x)$. On peut définir une fonction `same_app` qui prend en argument f, g et x et qui renvoie $f(gx)(gx)$ par

```
let same_app f g x = f (g x) (g x)
```

L'inconvénient de cette fonction est qu'elle calcule deux fois `g x` (donc une fois de trop). Pourquoi est-ce un problème ? Comment y remédier ?

2.2 Arithmétique de Peano

On peut redéfinir les entiers de la façon suivante

```
type nat =
  | O
  | S of nat;;
```

(attention « `O` » est la lettre O et pas le chiffre 0). `O` signifie 0 et `S` n signifie $n + 1$. Ainsi, 2 s'écrira par exemple `S (S O)`. On appellera cette notation d'un entier la notation de Peano. Cette définition a en effet été employée par monsieur Peano pour axiomatiser l'arithmétique.

▶ Définir une fonction de type `nat -> nat` et qui renvoie `true` si son argument est nul (et `false` sinon).

▶ Écrivez une fonction `int_of_nat` de type `nat -> int` qui donne la « valeur » d'un entier écrit sous forme de Peano. Par exemple `eval (S (S O))` devra renvoyer `2`.

▶ Et hop dans l'autre sens... Écrivez `nat_of_int`.

▶ Essayez de définir la fonction `equal n m` de la façon suivante

```
let equal m n =
  match n with
    | m -> true
    | _ -> false
;;
```

Calculez `equal O (S O)`. Le résultat est-il celui auquel vous vous attendiez ? Pourquoi ? Définissez correctement cette fonction.

▶ L'addition vérifie les deux égalités suivantes

$$\begin{aligned} x+0 &= x \\ x+(y+1) &= (x+1)+y \end{aligned}$$

Utilisez-les pour définir récursivement l'addition sur les `nat`, `add n m : nat -> nat -> nat`. Testez votre fonction sur quelques exemples.

L'égalité est symmétrique. Pourtant que se serait-il passé si vous aviez « lu » les égalités dans l'autre sens ?

▶ Qu'est ce que vaut `add (S 0)` ? Quel est son type ? Comment appelle-t-on des fonctions qui peuvent être appliquées partiellement ?

▶ Redéfinissez la fonction `add` mais cette fois avec le type `nat * nat -> nat`. Comment obtenez-vous la fonction de la question précédente à partir de la nouvelle définition de `add` ?

▶ Refaites évaluer à Caml la définition du type `nat` sans lui refaire évaluer la définition de `add`, puis calculez `add (S 0) (S (S 0))`. Qu'obtenez-vous ? Caml est-il devenu fou ?

▶ Définissez une fonction récursive `pred : nat -> nat` qui a un entier n renvoie son prédécesseur (on mettra une valeur arbitraire à `pred O`, ou un `failwith`). Combien d'appels récursifs votre fonction fait-elle en fonction de son argument ?

▶ (★) Définissez la multiplication en vous inspirant de la définition l'addition et en utilisant la fonction `add`.

3 Itérée d'une fonction (si le temps le permet)

Si f est une fonction quelconque de type $\alpha \to \alpha$ et x est une valeur de type α, on définit $f^n(x)$ comme l'itérée n fois de f :

$$f^n(x) = \underbrace{f(f(\ldots f(x)\ldots))}_{n \text{ fois}}$$

Par convention $f^0(x) = x$.

▶ Si n est un entier de Peano, trouvez une définition récursive de $f^n(x)$ en distinguant les cas $n =$ `O` et $n =$ `S n'`.

▶ Définissez la fonction `iter_x` telle que `iter` n f $x = f^n(x)$.

▶ Modifiez la fonction précédente en une fonction `iter` telle que `iter` $f = f^n$.

▶ Écrivez une fonction `int_list` qui à un entier `n` de type `int` associe la liste

$$\underbrace{\texttt{[S (S (...S (O) ...))}}_{n \text{ fois}}\texttt{; ... ; O]}$$

▶ Écrivez une fonction `map` qui applique une fonction f à tous les éléments d'une liste :

```
map f [x1 ; ... ; xn]=[f x1 ; ... ; f xn]
```

Cette fonction existe en réalité déjà dans la librairie standard de Caml et s'appelle `List.map`.

▶ Si f est une fonction, x est une valeur, et n un entier de Peano, on peut calculer la liste des itérés de f par

```
let iteres n f x =
  map (fun k -> iter_x k f x) (int_list n)
```

Essayez de comprendre cette définition. Pourquoi est-elle peu efficace ? Programmez une meilleure version de cette fonction.