Programmation – TD4 : Paresse, flots, continuations

{ Jeremie.Detrey, Emmanuel.Jeandel } @ens-lyon.fr

20/21 octobre 2004

1 Paresse

1.1 Mise en jambes

Lors du précédent cours de programmation, Daniel a défini une fonction calculant les termes de la suite de Fibonacci de la manière suivante :

```
let fibo fibo' n =
  if n <= 1 then n
  else (fibo' (n-1)) + (fibo' (n-2));;
```

C'est exprès que cette fonction n'a pas été définie récursivement. Elle attend donc en guise de premier argument la fonction `fibo'` qu'elle doit appeler pour effectuer ses deux appels récursifs `fibo' (n-1)` et `fibo' (n-2)`. Le but de cet exercice est de rendre cette fonction récursive (sans faire de `let rec fibo` bien entendu).

1. Définissez-donc `fibo` comme précédemment. Observez et interprétez son type. Quel est le type de `fibo'` ?
2. Écrivez une fonction `recursive : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b` qui rend récursive une fonction `f` définie de la même manière que `fibo`.
   Le type de `f` est donc `('a -> 'b) -> 'a -> 'b` et `recursive f` doit avoir pour type `'a -> 'b`.
3. Essayez votre fonction `recursive` sur `fibo`.

C'est bien joli tout ça, mais notre fonction ne nous permet que de rendre récursives des fonctions à 1 argument, car `recursive` renvoie une fonction de type `'a -> 'b`.

Un élève a alors une idée, et écrit :

```
let rec recursive f = f (recursive f);;
```

C'est une bonne idée, car maintenant, `recursive` est de type `('a -> 'a) -> 'a`, ce qui va nous permettre de récursifier des fonctions de n'importe quel type.

4. Ce qu'a écrit l'élève vous choque-t-il ? Pourquoi ?
5. Appliquez cette fonction sur notre `fibo` du départ, et évaluez `(recursive fibo) 17`. Que se passe-t-il ? Pourquoi ça marchait avant ?
   Indiquestion : l'évaluation en Caml c'est quoi déjà ? du *innermost* ou du *outermost* ?

Il faut donc pouvoir dire à Caml de ne pas évaluer `recursive f` si c'est inutile. Pour cela, l'évaluation paresseuse est notre amie.

Rappel : en Caml on peut faire du *lazy* :
- écrire `lazy` *expr* permet de geler l'évaluation de l'expression *expr* ;
- une expression de type `'a` dont l'évaluation a été gelée par un `lazy` est de type `Lazy.t` (ou `lazy_t`) ;
- la fonction `Lazy.force` permet d'évaluer une expression qui avait été gelée ; on peut faire plusieurs appels à `Lazy.force` sur une même expression gelée, mais seul le premier appel va l'évaluer effectivement, les appels suivants utilisant le résultat mémorisé du premier appel.

6. Modifiez les fonctions `fibo` et `recursive` en utilisant l'évaluation paresseuse. Votre nouvelle fonction `recursive` devra avoir pour type `('a lazy_t -> 'a) -> 'a`.

1.2 Mémoizons z'un brin

1.2.1 Fibonacci, encore et toujours

Reprenons la fonction `fibo` définie plus haut (pas besoin du *lazy* ici), et ajoutons-lui une ligne au début, permettant de voir les appels récursifs :

```
let fibo fibo' n =
  print_string ("Calling fibo "^(string_of_int n)^"...\n");
  if n <= 1 then n
  else (fibo' (n-1)) + (fibo' (n-2));;
```

Remarque : on pourrait aussi faire `#trace fibo` pour faire plus sérieux.

On va aussi reprendre la fonction `recursive` écrite à la question 2.

1. Évaluez `(recursive fibo) 17`. Commentez.

Pour éviter de faire des appels récursifs inutiles, nous allons mémoriser les résultats déjà évalués.

2. Écrivez une fonction `memoize` qui fait comme `recursive`, mais en plus mémorise les résultats des appels dans une table de hashage.
Indication : allez lire votre cours si vous ne vous souvenez plus.

3. Vérifiez que ça marche, en regardant cette fois-ci les appels effectués lors de l'évaluation de `(memoize fibo) 17`.

1.2.2 Nombres premiers

4. Écrivez une fonction `is_prime : int -> bool` qui teste (de manière naïve) si un nombre est premier ou pas.

5. Utilisez la mémorisation pour retenir la primalité des nombres déjà testés et éviter de les tester à nouveau. Faites attention à ce que votre table de hashage se soit pas accessible depuis l'extérieur de la fonction.

1.3 La fabuleuse fonction de Wilhelm Ackermann

Voici un exemple (presque) concret, où l'on sent bien pourquoi il est utile de mémoriser lorsqu'on fait de la récursion violente.

La fonction d'Ackermann $A(m,n)$ (avec $m,n \in \mathbb{N}$) est définie comme suit :

$$\begin{cases} A(0,n) &= n+1 \\ A(m,0) &= A(m-1,1) \\ A(m,n) &= A(m-1,A(m,n-1)) \end{cases}.$$

Appétissant, non ? Les plus téméraires d'entre vous peuvent s'amuser à calculer la forme générale des termes $A(3,n)$ ou $A(4,n)$ pour rigoler un peu (ou pas).

1. Écrivez la fonction d'Ackermann `ack : int -> int -> int`.
2. Testez la sur quelques exemples. En particulier $A(3,10)$ et $A(4,1)$. (N'oubliez pas Ctrl-C Tab pour interrompre le top-level sous Emacs.)

Vous pouviez vous douter que les calculs allaient être immondes à la tête de la fonction, en fait.

3. Passez donc cette fonction à la moulinette pour la mémoizer.
4. Retentez les exemples précédents. Ô joie !
   Tentez $A(4,2)$? Fallait pas rêver non plus (mais là c'est pas notre problème, pour aujourd'hui tout du moins).

2 Flots paresseux

2.1 Des listes infinies ? et pourquoi pas des objets récursifs ? !

Un flot est une liste (potentiellement infinie) d'objets.

Si votre première réaction en voyant le terme *liste infinie* a été du genre «ha ha ! même pas cap'», donnez donc ceci à Caml :

```
let rec l = 0::l;;
```

Esbaudissez-vous une paire de fois.

En fait, il y a une astuce (quelle surprise) : Caml ne stocke pas le nombre `0` une infinité (ni même un nombre fini mais très très grand) de fois. Il sait juste que la liste commence par `0` et que sa queue est elle-même.

On va s'inspirer d'une remarque un peu plus générale pour représenter nos flots : une liste infinie peut être donnée par un nombre fini (typiquement 1) d'éléments de tête, ainsi qu'une expression permettant de construire la queue de la liste. Bien-sûr, si cette expression est évaluée, elle va renvoyer une liste elle-même potentiellement infinie. On va donc bien se garder de faire cette évaluation en mettant des `lazy` partout.

Le type d'un flot (*stream*) est donc le suivant :

```
type 'a stream = Nil | Cons of 'a * ('a stream Lazy.t);;
```

où `Nil` désigne le flot vide.

On peut alors écrire la liste `l` précédente comme un flot :

```
let rec l = Cons (0, lazy l);;
```

2.2 Fonctions de base

À titre d'exemple, voici une fonction qui convertit une liste en un flot :

```
let rec stream_of_list = function
  | [] -> Nil
  | x::q -> Cons (x, lazy (stream_of_list q));;
```

Cette fonction évite donc tout appel récursif inutile car il est protégé par un `lazy`. Cela veut dire que la queue de la liste ne devra être évaluée que lorsque l'on tentera d'y accéder.

Si vous n'avez pas compris, hurlez à la mort, le TDman viendra vous expliquer. Sinon, on se jette à l'eau[1] :

1. Écrivez une fonction `hd : 'a stream -> 'a` qui renvoie le premier élément d'un flot (ou une erreur s'il est vide).
2. Écrivez une fonction `tl : 'a stream -> 'a stream` qui renvoie la queue d'un flot (ou une erreur).
3. Écrivez une fonction `iter : ('a -> unit) -> 'a stream -> unit` qui itère une fonction sur tous les éléments d'un flot. Faites attention aux flots infinis : mettez une borne sur le nombre d'itérations pour éviter de tuer votre top-level toutes les 30 secondes.
   Remarque : cette fonction va vous permettre d'afficher simplement le contenu d'un flot, faites des tests.

Rappel : le type `option` est défini dans Caml par :

```
type 'a option = None | Some of 'a;;
```

Il permet de renvoyer une valeur *optionnelle* de type `'a`.

Pour l'instant, mis à part les listes comme `l` de la partie précédente, vous n'avez pas manipulé de liste infinie. Remédions vite à cela :

4. Écrivez une fonction `stream_from_fun : (int -> 'a option) -> 'a stream` qui construit, à partir d'une fonction `f`, le flot $[\texttt{f}_0; \texttt{f}_1; \texttt{f}_2; \ldots; \texttt{f}_{n-1}]$ tel que pour tout i dans $[0; n-1]$, `f` $i =$ `Some` $\texttt{f}_i$, et où n est le plus petit entier tel que `f` $n =$ `None`. Si un tel n n'existe pas, le flot est infini.

À partir de la fonction précédente, on peut définir le flot des entiers naturels par :

```
let nat = stream_from_fun (fun n -> Some n);;
```

2.3 Transformation, filtrage

1. Écrivez une fonction `map : ('a -> 'b) -> 'a stream -> 'b stream` qui applique une fonction `f` à tous les éléments d'un flot $[x_0; x_1; x_2; \ldots]$ et renvoie le flot $[\texttt{f}\ x_0; \texttt{f}\ x_1; \texttt{f}\ x_2; \ldots]$.
   Attention : ici on ne se défile pas, et on ne tente pas de borner la taille du flot renvoyé. On fait confiance à `lazy`.
2. Écrivez une fonction `filter : ('a -> bool) -> 'a stream -> 'a stream` qui prend un prédicat `p` et un flot `s`, et renvoie le flot des éléments de `s` qui vérifient `p`.

[1] Normal, on fait des flots...

3. Définissez le flot des nombres premiers par le crible d'Ératosthène, en faisant des filtrages à partir du flot des entiers.
   Indication : vous allez écrire une fonction récursive `sieve : int stream -> int stream` qui prend un flot d'entiers, considère le premier élément de ce flot comme premier, et élimine par filtrage tous les multiples de cet élément. Il vous suffira ensuite d'appeler `sieve` sur le flot des naturels supérieurs à 2.

2.4 Quick-sort

Puisque les flots sont des listes, à quelques légers détails près, on peut s'amuser à recoder les algos classiques sur les listes.

1. Écrivez une fonction `append : 'a stream -> 'a stream -> 'a stream` qui concatène deux flots. Et si le premier flot est infini, on fait quoi ?
2. Écrivez une fonction `quick_sort : 'a stream -> 'a stream` qui trie un flot selon l'algorithme *quick-sort* (on sélectionne un pivot dans le flot, on partage le flot en deux en comparant avec le pivot, on trie récursivement puis on concatène dans le bon ordre). Ça marche sur les flots infinis ça ?

2.5 Lecture d'un fichier

Les flots sont très utiles lorsque l'on *parse* un fichier. En effet, on peut voir un fichier comme un flot de caractère, potentiellement infini (l'entrée standard par exemple).

Voici une fonction `stream_from_file` qui ouvre le fichier indiqué par `filename` et renvoie le flot de caractères correspondant à ce fichier :

```
let stream_of_file filename =
  let f = open_in filename in
    stream_of_fun (fun _ -> try Some (input_char f) with End_of_file -> None);;
```

1. Écrivez une fonction `to_lines : char stream -> string stream` qui prend un flot de caractères et renvoie le flot formé en groupant les caractères d'une même ligne (dans le flot de caractères, les lignes sont séparées par des `'\n'`).
2. Le TD dont vous êtes le héros : vos TDmen sont à court de questions sympas à poser sur la lecture des fichiers. Réfléchissez à un truc marrant à faire, puis faites le.

3 Le commencement des continuations

Les continuations, c'est surtout pour la prochaine fois. Juste un petit avant-goût pour vous faire languir (ou pas) le TD prochain.

1. Adaptez la fonction `is_prime` d'avant pour qu'elle prenne sa continuation en argument. (Pas besoin d'utiliser la version avec mémorisation, la fonction `is_prime` de base va suffire.)
   Votre nouvelle fonction doit avoir pour type : `int -> (bool -> 'a) -> 'a`.
2. Écrivez une fonction `prod_prime : int -> (int -> 'a) -> 'a` qui prend comme argument un entier `n` et calcule le produit des nombres premiers inférieurs à `n`, en faisant des continuations dans tous les sens. C'est beau, non[2] ?

[2] Just kidding.