Programmation TP 6 : Paresse

{ jrobert, fbouchez, apardon } @ens-lyon.fr
http://perso.ens-lyon.fr/florent.bouchez/

23 octobre 2007

1 Récursion ouverte

On peut écrire la fonction de Fibbonacci comme suit :

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

let rec fib n = fibo fib n;;
```

La récursion est ouverte dans la première définition par l'appel récursif sur l'argument `fibo'` et on la ferme dans la seconde définition.

Q 1.1 Observez le type de la fonction `fibo`.

Q 1.2 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`.

Q 1.3 Essayez votre fonction `recursive` sur `fibo`.

Considérons cette autre définition de `recursive` :

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

Le type de `recursive` est plus général : `('a -> 'a) -> 'a`. En fait, cette fonction calcule le point fixe de n'importe quelle fonction.

Q 1.4 Appliquez cette fonction sur le `fibo` initial, et évaluez `(recursive fibo) 17`. Que se passe-t-il ? Pourquoi ça marchait avant ?

Il faut en fait évaluer `recursive f` de façon paresseuse. Nous allons donc utiliser le possibilités d'OCaml en matière de paresse :

- `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 utiliseront le résultat mémorisé du premier appel.

Q 1.5 Modifiez les fonctions `fibo` et `recursive` en utilisant l'évaluation paresseuse. Votre nouvelle fonction `recursive` devra avoir pour type `('a lazy_t -> 'a) -> 'a`.

2 Mémo(r)isation

2.1 Fibonnacci

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 =
  Printf.printf ("Calling␣fibo␣%d...\n") n;
  if n <= 1 then n
  else (fibo' (n-1)) + (fibo' (n-2));;
```

Remarque : `#trace fibo` fonctionnerait aussi.

Q 2.1 Évaluez `(recursive fibo) 17`. Commentez.

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

Q 2.2 Écrivez une fonction `memoize` qui fait comme `recursive`, mais qui en plus mémorise les résultats des appels dans une table de hachage (cf. `Hashtbl`).

Q 2.3 Vérifiez que ça marche, en regardant cette fois-ci les appels effectués lors de l'évaluation de `(memoize fibo) 17`.

Définissez une fonction ainsi :

```
let fibo_memo = memoize fibo
```

Q 2.4 Testez `fibo_memo 8`, puis `fibo_memo 9`. Comparez avec `(memoize fibo) 8/9`. Quels appels recalculent tous les résultats intermédiaires ?

Q 2.5 Si `fibo_memo` recalcule ces résultats, modifiez votre fonction `memoize` pour ne pas avoir à les recalculer. Si non, modifiez aussi votre fonction `memoize` pour obtenir l'effet inverse...[1]

Maintenant, vous pouvez choisir de revenir au TD précédent sur la génération de code. C'est l'option conseillée par les TD-men qui pensent que vous apprendrez plus de choses utiles comme ça. Gardez plutôt la suite pour les soirs d'hiver où vous vous ennuyerez.

2.2 La fonction d'Ackermann

La fonction d'Ackermann $A(m, n)$ est définie comme suit :

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

Q 2.6 Écrivez la fonction d'Ackermann `ack : int -> int -> int`.

Q 2.7 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.)

Q 2.8 Transformez cette fonction pour lui permettre de mémoriser.

Q 2.9 Reprenez les exemples précédents. Tentez $A(4,2)$.

[1] Le but de la question étant que vous compreniez bien la différence entre les deux version. (Et pas de « c'est magique » svp.)