Programmation – TD5 : Continuations (suite*)

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

27/28 octobre 2004

Rappelons le principe des continuations. Au lieu d'écrire une bonne vieille fonction `f` :

```
let f x = x + 1;;
```

on ajoute un argument à `f` (la kontinuation `k`) qui représente son futur, c'est-à-dire intuitivement ce qu'on doit effectuer une fois que `f` a fini son calcul. Si `f` était du type `'a -> 'b` (soit ici `int -> int`), la continuation sera du type `'b -> 'c` (on fait quelque chose avec le résultat de `f`).

On écrira donc :

```
let fc x k = let y = x + 1 in k y;;
```

Au lieu d'écrire :

```
print_int (f (1+(g x)));;
```

on écrit :

```
gc x (fun rg -> fc (1+rg) (fun rf -> print_int rf));;
```

1 Un peu d'exercice

1. Écrivez une version continuation `mapc` de la fonction `map` de sorte que `mapc f l k` renvoie le résultat de `k` appliqué à la liste des `(f i)` pour `i` $\in$ `l`.
2. Écrivez une version continuation `appendc` de la fonction `append` de sorte que `appendc l1 l2 k` renvoie le résultat de `k` appliqué à la concaténation de `l1` et `l2`.
3. Reprenez `mapc` mais en supposant maintenant que le premier argument `f` est également un argument type continuation, c'est-à-dire `f i k` représente intuitivement `k` appliqué au résultat d'un calcul effectué à partir de `i`.

*Si vous vous souvenez bien, il y en avait un peu dans le TD précédent. Tournez la page. Encore. Regardez, là, en bas ! Et puis bon, continuation, suite, ca fait jeu de mots. Continuation. Suite. Suite. Continuation. Non ?

2 Arbres de calcul

2.1 Introduction

Dans cette partie, on va enrichir successivement un exemple. On va partir d'un type :

```
type tree =
  | TInt of int
  | TAdd of tree * tree
  | TMul of tree * tree
```

qu'on va transformer en :

```
type tree =
  | TInt of int
  | TAdd of tree * tree
  | TMul of tree * tree
  | TVar of symbol
  | TFun of symbol * tree
  | TApp of tree * tree
```

et qu'on va continuer à enrichir.

Le but est de représenter un mini-langage de programmation qui peut manipuler des fonctions, des variables, qui peut lancer des exceptions, et que sais-je encore.

S'il n'y avait pas d'exceptions, programmer tout ceci est d'une simplicité enfantine et aurait pu être fait dès le premier TD de Caml. Mais les exceptions vous entraînent dans un monde plus adulte, dans lequel les continuations vont jouer un rôle essentiel.

2.2 Préliminaires[1]

On représente un langage de programmation rudimentaire par des arbres d'expressions :

```
type tree =
  | TInt of int
  | TAdd of tree * tree
  | TMul of tree * tree
```

Par exemple :

```
TAdd (TMul (TInt 32, TInt 51), TInt 32)
```

représente `(32*51) + 32`.

Il est temps de donner le coup d'envoi :

1. Écrivez une fonction `eval : tree -> (int -> 'a) -> 'a` telle que `eval t k` représente l'évaluation de `t` suivi de l'appel à la continuation `k` avec le résultat de l'évaluation. Ainsi :

   ```
   eval (TAdd (TMul (TInt 32, TInt 51), TInt 32)) print_int;;
   ```

   affiche `1664`.

[1] On en déduit que l'ordre des paragraphes n'est pas représentatif.

2.3 Variables

En avant ! Expliquons comment ajouter variables et fonctions :

```
type symbol = string
type tree =
  | TInt of int
  | TAdd of tree * tree
  | TMul of tree * tree
  | TVar of symbol
  | TFun of symbol * tree
  | TApp of tree * tree
```

On pourra maintenant écrire :

```
TApp (TFun ("x", TMul (TVar "x", TInt 32)), TInt 52)
```

pour représenter `(fun x -> x*32) 52`.

Le résultat de l'évaluation d'un arbre peut maintenant être autre chose qu'un entier, par exemple une fonction. Une valeur a donc le type suivant :

```
type value = VInt of int | VFun of symbol * tree
```

On a maintenant besoin, pour évaluer un arbre, d'un contexte d'évaluation, c'est-à-dire une liste de paires (symbole, valeur) :

```
type context = (symbol * value) list
```

2. Écrivez une nouvelle fonction `eval : tree -> context -> (value -> 'a) -> 'a` telle que `eval t c k` représente l'évaluation de `t` dans le contexte `c` suivi de l'appel à la continuation `k` avec le résultat de l'évaluation.
Ainsi, le code suivant :

```
let print v = match v with
  | VInt n -> print_int n
  | VFun _ -> print_string "<fun>";;

let context = [ ("x", VInt 52) ; ("y", VInt 32) ];;

eval (TMul (TVar "x", TInt 32)) context print;;
eval (TFun ("x", TMul (TVar "x", TInt 32))) context print;;
```

écrit successivement `1664` puis `<fun>`.

2.4 Exceptions : simulation

On explique ici comment on traite les exceptions, c'est à dire comment lancer des exceptions[2] et comment les rattraper[3].

On ajoute pour cela directement les continuations au langage de la façon suivante :

[2] On dit aussi botter.
[3] Arrêts de volée.

```
type 'a value =
  | VInt of int
  | VFun of (symbol * tree)
  | VCont of 'a continuation
and 'a continuation = 'a value -> 'a
```

Avec la sémantique suivante : `TApp (x,y)` où `x` s'évalue en une continuation `k` revient à faire `eval y context k`.

3. Sans rien programmer, donnez le résultat du code suivant :

```
let expr    = TAdd (TInt 3, TMul (TApp (TVar "raise", TInt 19), TInt 9))
and context = [ ("raise", VCont (fun _ -> print_int 1664)) ;
                ("x", VInt 51) ]
in eval expr context print;;
```

On ajoute deux constructions syntaxiques `TryWith` et `Raise` :

```
type tree =
  | TInt of int
  | TAdd of tree * tree
  | TMul of tree * tree
  | TVar of symbol
  | TFun of symbol * tree
  | TApp of tree * tree
  | TTryWith of tree * symbol * tree
  | TRaise of tree
```

L'idée étant que le code suivant :

```
  (* try (raise 3) + 9 with x -> x*2 *)
let expr = TTryWith (TAdd (TRaise (TInt 3), TInt 9),
                     "x", TMul (TVar "x", TInt 2))
and context = []
in eval expr context print;;
```

écrit 6.

4. Écrivez le code de la fonction `eval` correspondante.

2.5 Prolongations

Si jamais vous vous sentez bien ici et que vous ne voulez pas passer à la prochaine partie, vous pouvez botter en touche et ajouter les constructions `IfThenElse`, et toute autre de votre choix.

3 Programmation parallèle, ou comment faire à plusieurs[4]

Le but est d'écrire un petit ensemble de routines simulant un environnement parallèle (multi-threads). Chaque thread (représenté par une fonction Caml) devra être écrit dans le style de programmation par passage de continuations.

[4]On remercie les TDs précédents desquels cette partie a été habilement recopiée

3.1 Vers un monde parallèle

Quelques définitions :
- Les *continuations* sont des fonctions de type `unit -> 'a`. On les déclenche en les appliquant à `()`.
- Les *processus* (ou *threads*) sont des fonctions de type `(unit -> 'a) -> 'a`. Ils prennent en argument une continuation, qu'ils déclenchent quand ils ont fini.

Notre environnement est prévu pour des processus :
- écrits par passage de continuations ;
- et qui donnent régulièrement la main à l'environnement (voir ci-dessous ce que ça veut dire).

Pendant l'exécution, les processus en attente sont gelés sous forme de continuations : le processus `p` devient `fun () -> p cc`, où `cc` désigne la continuation courante (voir ci-dessous aussi).

Techniquement, notre environnement maintient une file (FIFO) *globale* de processus en attente (c'est-à-dire de continuations). On appelle continuation courante à un instant donné la continuation en tête de cette file.

On appelle `fifo` notre file globale et on définit la continuation `cc` par :

```
let cc () = try Queue.pop fifo () with Queue.Empty -> ();;
```

L'environnement offre deux fonctions aux programmes l'utilisant :
- La fonction `fork` lance un nouveau processus en parallèle au processus courant. Plus précisément, si `p` est un processus et `k` est une continuation, `fork p k`, de façon non-déterministe, soit met `p` en attente et déclenche `k`, soit met `k` en attente et exécute `p` avec la continuation courante.
- La fonction `yield` rend la main à l'environnement. Lorsqu'un programme rend la main à l'environnement, en gros, il dit : «Voilà, moi il me reste `k` à faire, mais je peux attendre cinq minutes». Plus précisément, étant donnée une continuation `k`, `yield k`, de façon non-déterministe, soit met `k` en attente et déclenche la continuation courante, soit déclenche `k` et ensuite déclenche la continuation courante.

3.2 La base

1. Écrivez les fonctions `fork` et `yield`, puis exécutez le code suivant :

```
let thread i k' = print_int i; print_newline (); k' () in
let rec prog i k =
  fork
    (thread i)
    (fun () -> if i > 0 then prog (i - 1) k else k ())
in prog 100 (fun () -> ());;
```

3.3 Une mémoire partagée

Définissons un tableau d'entiers global `memory`.

2. Écrivez par passage de continuations les deux fonctions `read` et `write` suivantes. La fonction `read` attend un indice `i` et une continuation `k` (au sens général, de type `int -> 'a`).

Elle appelle `k` sur l'entier `memory.(i)`. La fonction `write` fait la même chose, mais pour écrire dans `memory`.

3. Écrivez un programme dans lequel plusieurs (mettons 100) threads lisent `memory.(0)` avec `read`, font un `yield`, puis incrémentent de 1 la valeur obtenue, mettent le résultat dans `memory.(0)` et l'affichent. Observez le résultat.

3.4 Cohérence et sémaphores

On veut maintenant refaire le programme de l'exercice précédent, mais de telle sorte que les entiers soient lus et écrits dans l'ordre. Pour éviter le désordre de l'exercice précédent, on introduit des sémaphores. Un sémaphore est une file de continuations, avec un booléen indiquant s'il est libre. On définit les deux opérations suivantes sur les sémaphores :

- La fonction `sem_p : semaphore -> unit continuation -> unit` essaie de prendre le sémaphore. Si `s` est un sémaphore libre, `sem_p s k` le marque comme occupé et déclenche `k`. Si `s` est occupé, `sem_p s k` met `k` en attente dans la file de `s` et appelle la continuation courante.
- La fonction `sem_v : semaphore -> 'a continuation -> unit` libère un sémaphore occupé. Si `s` est un sémaphore occupé, `sem_v s k` le rend libre, remet sur la file globale (`fifo`) la tête de la file de `s` (en prenant soin de le protéger à nouveau par un `sem_p`), puis déclenche successivement `k` et `cc`. Le comportement de `sem_v` sur les sémaphores libres est non-spécifié.

4. Réécrivez donc votre programme avec des sémaphores pour que chaque thread ait un accès exclusif à la mémoire.