Programmation TP 8 : Continuations

{ jrobert, fbouchez, apardon } @ens-lyon.fr
http://perso.ens-lyon.fr/florent.bouchez/
6 novembre 2006

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 -> unit` (on fait quelque chose avec le résultat de `f`). La nouvelle fonction est donc de type `'a -> ('b -> unit) -> unit`. La fonction `f` se transforme ainsi en :

```
let fc x k = k (x + 1);;
```

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

Q 1.1 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` ∈ `l`.

Q 1.2 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`.

Q 1.3 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`.

2 Programmation parallèle : simulation à l'aide de continuations

Le but ici 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. De cette manière, l'ordonnancement entre les threads reviendra à manipuler des continuations, vues comme le « futur » de chaque calcul destiné à être effecuté.

2.1 Vers un monde parallèle

Les threads sont des continuations de type `unit -> unit` qui donnent régulièrement la main à l'environnement en appelant la fonction `yield` (voir ci-dessous). À un instant donné, un seul thread s'exécute ; les autres sont gelés et stockés par l'environnement dans la file `fifo`. La fonction `run` s'occupe de gérer l'ensemble des threads, les exécutant un à un jusqu'à ce que la file soit vide.

```
let fifo = Queue.create ();;
let run p =
  Queue.clear fifo;
  Queue.push p fifo;
  while not (Queue.is_empty fifo) do Queue.pop fifo () done;;
```

L'environnement fournit deux fonctions aux programmes pour manipuler des threads.

- La fonction `fork` lance un nouveau thread en parallèle au thread courant. Plus précisément, si `f` est une fonction et `k` est la continuation du thread courant, `fork t k` met `k` en attente dans `fifo` puis exécute `t`.
- La fonction `yield` rend temporairement la main à l'environnement. Plus précisément, étant donnée une continuation `k`, `yield k` met `k` en attente dans `fifo`.

Remarque : comme les threads travaillent ici par continuation, quand les fonctions `fork` et `yield` rendent la main, c'est la boucle de `run` qui la récupère puisque ces fonctions n'ont pas exécuté la continuation qui leur était passée en argument.

2.2 La base

Q 2.1 Écrivez la fonctions `fork` puis exécutez le code suivant :

```
let _ =
  let thread i () = print_int i; print_newline () in
  let rec prog i =
    fork
      (thread i)
      (fun () → if i > 0 then prog (i - 1))
  in run (fun () → prog 100);;
```

Q 2.2 Dans la vraie vie, il est rare que `fork` exécute immédiatement le nouveau thread. Il peut arriver que le nouveau thread soit mis en attente tandis que le thread courant continue son exécution. Ajoutez un peu de hasard pour que `fork` hésite entre ces deux comportements.

Q 2.3 Écrivez la fonction `yield`. De même que pour `fork`, le comportement de `yield` n'est en réalité pas aussi déterministe que présenté : il peut arriver que `yield` continue l'exécution du thread courant plutôt que de passer la main à un autre thread. Là encore, ajoutez une touche de hasard pour simuler ce comportement.

2.3 Une mémoire partagée

Définissons un tableau d'entiers global `memory`.

Q 2.4 Écrivez par passage de continuations les deux fonctions `read` et `write`. La fonction `read` attend un indice `i` et une continuation `k` de type `int -> unit`. Elle appelle `k` sur l'entier `memory.(i)`. La fonction `write` écrit quant à elle dans `memory`, avant d'exécuter une continuation de type `unit -> unit`.

Q 2.5 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.

2.4 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 :

- `sem_p : semaphore -> 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 locale au sémaphore `s`.
- `sem_v : semaphore -> continuation -> unit` libère un sémaphore occupé. Si `s` est un sémaphore libre, lancez une exception. Dans le cas contraire, le comportement dépend de la présence ou non de threads en attente dans la file du sémaphore. S'il n'y en a pas, il suffit de marquer le sémaphore comme étant libre puis de continuer à exécuter le thread courant. S'il y en a, il faut en sortir un de la file et le placer en attente dans l'environnement, pour que `run` ait un jour l'occasion de l'exécuter. Là encore, on finit en continuant à exécuter le thread courant.

Q 2.6 Réécrivez votre programme avec des sémaphores pour que chaque thread ait un accès exclusif à la mémoire.

3 Simuler les exceptions

On s'intéresse ici à un MiniML comprenant des constructions pour lever et rattraper des exceptions.

```
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
  | TTryWith of tree * symbol * tree
  | TRaise of tree

type value =
  | VInt of int
  | VFun of (symbol * tree)
type context = (symbol * value) list
type continuation = value → unit
```

L'objectif est d'écrire une fonction `eval : tree -> context -> continuation -> continuation -> unit` telle que `eval t c ke k` représente l'évaluation de `t` dans le contexte `c` suivi de l'appel à la continuation `k` avec le résultat de l'évaluation. Si une exception est levée à l'extérieur de tout bloc `TTryWith`, c'est la continuation `ke` qui est appelée avec le résultat de l'évaluation de l'expression contenue dans le `TRaise`.

L'idée est que le code suivant affiche `6` :

```
let print v = match v with
  | VInt n → print_int n
  | VFun _ → print_string "<fun>";;
let terminate = fun _ → print_string "Unhandled exception";;

(* 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 terminate print
```

Q 3.1 Écrivez le code de la fonction `eval`.