Programmation 1 – TP n° 3
Continuations

Rappels - Continuations

Au lieu d'écrire une fonction `f` sous la forme : `let f x = x + 1`, on passe en argument à `f` la continuation `k` qui représente son futur (intuitivement, ce qu'on doit effectuer après le calcul de `f`).

Si `f` était du type `'a -> 'b`, la continuation sera du type `'b -> 'c` et la nouvelle fonction `f` sera donc du type `'a -> ('b -> 'c) -> 'c` :

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

Ainsi, si avant on écrivait

```
let g x = ... in
  print_int (f (1 + g x) );;
```

on utilise maintenant :

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

(alors que, initialement, `print_int` utilisait `f` qui appelait `g`, dans la version par continuation c'est `g` qui se continue avec `f` qui se continue avec `print_int`).

Q 0.1 Ecrire `mapc : ('a -> 'b) -> 'a list -> ('b list -> 'c) -> 'c`, version par continuation (et récursive terminale) de la fonction `map`. Noter que la fonction prise en argument par `mapc` n'est pas écrite par continuation.

Q 0.2 Ecrire `mapc2` qui fait la même chose mais dont la fonction prise en argument est écrite par continuation.

1 Traductions CPS

1.1 Sur Papier

On rappelle l'encodage CPS vu en cours (avec des couples de continuations) :


```
⟦e1 + e2⟧ = fun (k,kE) ->
              ⟦e1⟧ (fun v1 -> ⟦v2⟧ (fun v2 -> k (v1 + v2), kE),kE)
⟦e1 e2⟧ = fun (k,kE) ->
              ⟦e1⟧ (fun v1 -> ⟦e2⟧ (fun v2 -> v1 v2 (k,kE), kE))
⟦fun x -> e⟧ = fun (k,kE) -> k (fun x -> ⟦e⟧)
⟦let x = e1 in e2⟧ = fun (k,kE) -> ⟦e1⟧ (fun x -> ⟦e2⟧ (k,kE))
⟦raise (E e)⟧ = fun (k,kE) -> ⟦e⟧ (fun v -> kE v, kE)
⟦try e1 with | E x -> e2⟧ = fun (k,kE) -> ⟦e1⟧ (k,fun x -> ⟦e2⟧ (k,kE))
```

Q 1.1 Evaluer :

$$[\![\texttt{(fun x -> x + 12) 3}]\!] \ \texttt{(id,id)}$$

Q 1.2 Donner un encodage CPS avec une seule continuation.

1.2 Sur Machine

On considère un encodage CPS "pur" (sans gestion des exceptions) à une seule continuation.

Utiliser les fichiers fournis dans `tp3.tgz` :

- `langage.ml` contient un mini-langage de programmation, sans exceptions.
- `run.ml` contient l'interpréteur de ce langage.
- `cps.ml` contient le squelette de la traduction CPS du langage dans lui-même.
- `tests.ml` contient des tests pour vérifier l'encodage

Q 1.3 Lire `langage.ml`. Ne pas lire le code des fonctions de `run.ml`.

Q 1.4 Réaliser l'encodage CPS et le vérifier.

2 Des `break` dans les `while`

Le fonctionnement de l'opérateur `while` est bien connu.
A titre indicatif et formel, on donne sa sémantique opérationnelle à grand pas :

$$\frac{(C,\sigma) \Rightarrow \sigma' \qquad (\texttt{while}\ b\ \texttt{do}\ C,\sigma') \Rightarrow \sigma'' \qquad B[b]_\sigma = \texttt{true}}{(\texttt{while}\ b\ \texttt{do}\ C,\sigma) \Rightarrow \sigma''}$$

$$\frac{B[b]_\sigma = \texttt{false}}{(\texttt{while}\ b\ \texttt{do}\ C,\sigma) \Rightarrow \sigma}$$

où :

- $\frac{A}{B}$ est la règle de déduction qui permet de déduire B de A
- σ dénote un contexte d'évaluation, C dénote une commande.
- $(C,\sigma) \Rightarrow \sigma'$ signifie que l'exécution de la commande C dans le contexte σ aboutit au nouveau contexte σ'.
- $B[b]_\sigma$ est l'évaluation du booléen b dans le contexte σ

Q 2.1 Programmer le `while` par continuation : le type de la fonction `do_while` doit être `(unit -> bool) -> ((unit -> 'a) -> 'a) -> (unit -> 'a) -> 'a`.

Q 2.2 Utiliser cette fonction pour calculer la somme des éléments d'un tableau d'entiers.

Q 2.3 Ecrire une fonction `do_whileB` qui permet d'utiliser un `break` lors de la definition du corps de la boucle.

Q 2.4 Ecrire une fonction maline qui calcule le produit des éléments d'un tableau d'entiers (ne pas oublier que 0 est absorbant).

FIG. 1 – un exemple de `break`.

3 Programmation parallèle

L'objectif de cette partie est l'écriture de routines permettant la simulation d'un environnement multi-thread. Chaque thread est simulé par une fonction écrite en programmation par continuation. Ordonnancer les threads c'est donc manipulers les différents futurs.

3.1 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). A 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 la fonction que l'on veut exécuter en parallèle, et `k` la continuation du thread courant (ce qu'on veut faire après le fork), `fork f k` met `k` en attente dans `fifo` puis exécute `f`.
- La fonction `yield` rend temporairement la main à l'environnement. Plus précisément, étant donnée une continuation `k` (ce qu'il nous reste à faire dans notre thread), `yield k` met `k` en attente dans `fifo` et rend la main.

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.

3.2 La base

Q 3.1 Écrivez la fonction `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 3.2 En programmation, 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 `forkR` hésite entre ces deux comportements.

Q 3.3 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.

3.3 Une mémoire partagée

Définissons un tableau d'entiers global `memory`.

Q 3.4 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 3.5 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.

3.4 Cohérence et sémaphores

L'objectif ici de faire la même chose que précèdemment, 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` (pour **p***rolaag* ou **p***robeer te verlagen*) 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` (pour **v***erhoog*) 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 3.6 Réécrivez votre programme avec des sémaphores pour que chaque thread ait un accès exclusif à la mémoire.