TD9 : Les continuations

1 Notion de continuation

La notion de continuation est très utile dans de nombreuses situations dans lesquelles, très schématiquement, on a besoin de connaitre/nommer ce qui va s'exécuter **après** le code courant, *i.e.* ce qui *continue* l'exécution. Dans un langage fonctionnel, le code courant est associé à l'exécution d'une fonction f. Ce qui s'exécute après (la continuation, donc) est également une fonction, qui sera passée en paramètre (supplémentaire) de f.

On pourrait dire de façon cavalière qu'une continuation est un accumulateur de calculs (qu'il faudra effectuer plus tard), et non de données.

Supposons que l'on doive réaliser une fonction `f : a -> b` et qu'on veuille introduire une continuation dans la définition de `f`. L'application de `f` peut être suivie de n'importe quelle autre fonction `g : b -> c` produisant n'importe quel type de résultat `c`, *i.e.* on calcule `g (f x)` pour `x : a`.

On aura donc pour `f` un paramètre supplémentaire, la continuation représentant `g : b -> c`, le "futur" de `f`. Le type de retour de `f` est également changé. La version avec continuation de `f` sera appelée `kf : a -> (b -> c) -> c` (ou plus élégamment `kf : (b -> c) -> a -> c`, suivant l'ordre des arguments).

1.1 Passage par continuation

L'usage des continuations permet une transformation de programme systématique, illustrée par les exemples ci-dessous : la factorielle et la fonction de Fibonacci :

```
let rec fact n =
 if n = 0 then 1 else n * fact (n-1);;
(* kfact : int -> (int -> 'a) -> 'a *)
let rec kfact n k =
 if n = 0 then k 1 else kfact (n-1) (fun fact_n_1 -> k (n * fact_n_1));;

let fact' n = kfact n (fun fact_n -> fact_n);;

let rec fibo n =
 match n with
 | 0 | 1 -> 1
 | _ -> fibo (n-1) + fibo (n-2);;
(* kfibo : int -> (int -> 'a) -> 'a *)
let rec kfibo n k =
 match n with
 | 0 | 1 -> k 1
 | _ -> kfibo (n-2)
         (fun fib_n_2 -> kfibo (n-1)
                         (fun fib_n_1 -> k (fib_n_2 + fib_n_1)));;

let fibo' n = kfibo n (fun fib_n -> fib_n);;
```

On constate dans ces exemples que l'utilisation des continuations permet de **linéariser** l'exécution, *i.e.*de choisir exactement dans quel ordre unique se font les sous-appels récursifs. Cette linéarisation fait qu'il n'y a alors plus qu'un seul sous-appel récursif, de plus **terminal**, synonyme d'efficacité (pas d'utilisation de la pile d'appels). Cette technique standard d'élimination de la récursivité directe est appelée **passage par continuation** ou **Continuation Passing Style** (en abrégé CPS). Elle est utilisée dans les cas où la récursivité directe est trop profonde et dépasse les capacités de la pile d'appels (ce qui est très souvent le cas dans des applications réalistes "industrielles").

Remarque : La mémoire économisée dans l'absence de pile d'appels est consommée dans la création des fonctions/continuations passées en paramètres, il n'y a pas de miracle ! De même, le temps de calcul sera équivalent, probablement un peu plus long en raison de la gestion des continuations (création/application).

2 Application : flôt de contrôle

Comme on peut le constater, les continuations permettent de représenter et manipuler finement le flôt de contrôle (pour linéariser les appels de fonctions par exemple). On peut également s'en servir pour augmenter les possibilités de contrôle et même représenter/réaliser d'autres "effets de bords" au-dessus d'un langage qui ne les possède pas nativement. Les techniques CPS sont donc beaucoup utilisées pour réaliser des compilateurs et interprètes. Ceci est illustré par l'ajout des exceptions (simples !) au noyau purement fonctionnel de OCaml. Pour cela, classiquement, on utilisera deux continuations, une dite de succès $\texttt{k}_+$ pour gérer les calculs qui terminent sans exceptions et une autre dite d'échec $\texttt{k}_-$ pour les exceptions. Une transformation possible, à l'aide de continuations, des expressions augmentées (i.e. avec exceptions) en expressions ordinaires est alors donnée par la fonction $[\![\cdot]\!]$ suivante :

$$
\begin{array}{ll}
[\![\texttt{try}\ \ expr_1\ \ \texttt{with Exc v ->}\ \ expr_2]\!]\ \texttt{k}_-\ \texttt{k}_+ & \triangleq [\![expr_1]\!]\ \texttt{(fun v -> } [\![expr_2]\!]\ \texttt{k}_-\ \texttt{k}_+\texttt{)}\ \texttt{k}_+ \\
[\![\texttt{raise (Exc v)}]\!]\ \texttt{k}_-\ \texttt{k}_+ & \triangleq \texttt{k}_-\ \texttt{v} \\
[\![\texttt{(fun v -> } expr\texttt{)}]\!]\ \texttt{k}_-\ \texttt{k}_+ & \triangleq \texttt{(fun v -> } [\![expr]\!]\ \texttt{k}_-\ \texttt{k}_+\texttt{)} \\
[\![(f\ n\ arg)]\!]\ \texttt{k}_-\ \texttt{k}_+ & \triangleq [\![arg]\!]\ \texttt{k}_-\ \texttt{(fun varg -> } [\![f\,n]\!]\ \texttt{k}_-\ \texttt{(fun vfn -> vfn varg))} \\
[\![val]\!]\ \texttt{k}_-\ \texttt{k}_+ & \triangleq \texttt{k}_+\ val
\end{array}
$$

Disclaimer : OCaml possède déjà un mécanisme d'exceptions, plus efficace car natif...

Les continuations permettent également de changer la granularité des API, de réaliser des inversions de contrôle, d'implanter des coroutines, du non-déterminisme, etc.

3 Continuations natives

Les continuations ont été vues comme de simples fonctions passées systématiquement en paramètre (supplémentaire) de toutes les fonctions définies par l'utilisateur. Ceci est l'angle historique qui permet de donner une sémantique propre aux continuations. Néanmoins, cette vision est quelquefois inopérante en pratique car elle oblige à modifier le code source de toutes les fonctions pour lesquelles on souhaite appliquer des techniques CPS, ce qui n'est ni possible ni souhaitable en général. En ce sens, la notion de continuation n'est pas du tout orthogonale aux autres constructions, ni indépendante vis-à-vis d'elles.

Heureusement, les continuations natives existent elles aussi. On les retrouve dans de nombreux langages, sous la forme de primitives comme `call/cc` en Scheme, `setjmp/longjmp` en C, `yield` en Python, mais également en Scala, Racket, Haskell, etc. Ces opérations permettent de créér dynamiquement, en interrompant l'exécution à n'importe quel endroit du programme, une continuation qui représente les calculs restant à faire pour terminer l'exécution de celui-ci. Une fois la continuation créée et stockée dans une variable, on peut l'appeler comme une fonction ce qui redémarre l'exécution normalement ou bien calculer une valeur sans utiliser la continuation. Pour cette raison, les continuations sont parfois appelées "resumable exceptions".

Les continuations natives "classiques", dites non délimitées (comme `call/cc`) sont inefficaces, sujettes à fuites de mémoire et finalement rendent le code assez illisible et incompréhensible. Elles sont de fait peu utilisées. On s'intéresse donc aux continuations "modernes", dites délimitées, plus spécifiquement à la paire d'opérations `reset`/`shift`, implémentées par exemple dans la librairie `Delimcc`.

Étudions l'exemple suivant de la concaténation de listes. Tout d'abord, on transforme le programme initial pour faciliter le passage en CPS.

```
let rec append l1 l2 =
  match l1 with
  | []     -> l2
  | t1::q1 -> t1::append q1 l2;;

let append l1 l2 =
  let rec app_X_l2 l1 =
    match l1 with
    | []     -> l2
    | t1::q1 -> t1::app_X_l2 q1
 in app_X_l2 l1;;

let append l1 l2 =
  let rec app_X_l2 l1 =
    match l1 with
    | []     -> l2
    | t1::q1 -> let app_q1_l2 = app_X_l2 q1
                in t1::app_q1_l2
  in app_X_l2 l1;;
```

Puis, on transforme le programme en CPS, soit explicitement, soit par l'intermédiaire des primitives `reset`+/`shift`, dont l'effet est (presque) équivalent. On remarque que l'expression `app_X_l2 q1` a bien pour futur l'expression `t1::app_q1_l2`, qui en tant que dernière expression calculée, a elle-même pour futur `k`.

Appliquer `reset expr` revient à considérer que la conversion CPS commence à partir de `expr`. Appliquer `shift (fun k -> ...)` permet de capturer la continuation courante à l'endroit où le `shift` apparaît, comme si on l'avait effectivement passée en paramètre à travers toutes les expressions depuis le `reset`.

Ces primitives évitent d'avoir à transformer le code source pour utiliser des continuations, par contre elles n'offrent pas l'avantage d'économiser la pile d'appels. Dans cet exemple, si la liste `l1` est très grande (de l'ordre du million d'éléments), la version avec continuations explicites fonctionnera alors que la version avec les primitives fera déborder la pile d'appels.

De fait, l'implémentation de ces primitives revient à recopier une partie de cette pile d'appels lors de la construction de la continuation faite par `shift`, la partie se trouvant entre le `reset` le plus proche et le `shift`. Dans cet exemple, la continuation (finale) `k` correspond en fait à la fonction `(fun l -> t1::...::tn::l)` où `[t1::...::tn]` est la liste `l1`. Tous les éléments t_i sont stockés dans la pile d'appels.

```
let append l1 l2 =
  let rec app_X_l2 l1 k =
    match l1 with
    | []     -> k l2
    | t1::q1 -> app_X_l2 q1 (fun app_q1_l2 -> k (t1::app_q1_l2))
  in app_X_l2 l1 (fun app_l1_l2 -> app_l1_l2);;

let append l1 l2 =
  let rec app_X_l2 l1 =
    match l1 with
    | []     -> shift (fun k -> k l2)
    | t1::q1 -> t1::app_X_l2 q1
  in reset (app_X_l2 l1);;
```

Maintenant, on peut se poser la question de l'utilité réelle de ces mécanismes. En effet, l'utilisation de ces continuations est extrêmement contrainte, car leur type est défini complètement par le contexte où elles apparaissent. Ainsi, dans l'exemple de la fonction `append`, la continuation prend une liste, qui correspond au type de l'expression `shift (fun k -> ...)` forçé par son contexte et renvoie une liste, qui correspond au type de l'expression `(app_X_l2 l1)` argument du `reset`. À part appeler la continuation pour redémarrer et terminer l'exécution, que peut-on faire ? En utilisant un type somme, on peut autoriser une plus grande

variation des types et des usages. L'astuce est de distinguer les exécutions qui vont au bout de celles qui sont interrompues par la capture d'une continuation.

On souhaite par exemple séparer complètement le parcours de `l1` de la concaténation à `l2`. On commence par définir les différents retours possibles de la fonction de parcours, soit par terminaison normale (`Done`), soit par construction d'une continuation (`Request`). Dans notre cas simple, cette fonction ne termine jamais normalement. Ceci correspond à la déclaration du type de retour `res` et de la fonction récursive `app` qui ne mentionne pas `l2` ! Finalement, la fonction `append` récupère la continuation créée par `app` et termine l'exécution en l'appliquant à `l2`.

```
(* type 'res' des divers résultats possibles *)
type ('shift, 'reset) res =
 | Done    of 'reset
 | Request of ('shift -> ('shift, 'reset) res);;

let rec app l1 =
  match l1 with
  | []     -> shift (fun k -> Request k)
  | t1::q1 -> t1::app q1;;

let append l1 l2 =
  match reset (Done (app l1)) with
  | Done _    -> assert false
  | Request k -> match k l2 with
                 | Done res  -> res
                 | Request _ -> assert false;;
```

La librarie `Delimcc` propose une interface légèrement différente permettant de poser des "marqueurs" différents dans la pile, alors que `reset` pose toujours la même marque, c'est d'ailleurs la raison pour laquelle `shift` recopie la pile depuis le sommet jusqu'au dernier (marqueur de) reset seulement. La primitive `Delimcc.shift` permet de recopier la pile jusqu'à un marqueur quelconque. Ceci n'augmente pas l'expressivité mais facilite certaines utilisations avançées des continuations. La primitive `Delimcc.push_prompt` permet de poser un marqueur `prompt0`, c'est l'analogue de `reset`.

Voici une partie de l'interface de la librairie `Delimcc` en reprenant les conventions de nommage des variables de type `'reset` et `'shift` :

```
(* le type des marqueurs de pile *)
type 'reset prompt;;

val new_prompt   : unit -> 'reset prompt
val push_prompt  : 'reset prompt -> (unit -> 'reset) -> 'reset
val shift : 'reset prompt -> (('shift -> 'reset) -> 'reset) -> 'shift
```

Enfin, on transforme le code précédent en utilisant `Delimcc` :

```
(* creation d'un marqueur (unique pour notre application) *)
let prompt0 = Delimcc.new_prompt ();;

let rec app l1 =
  match l1 with
  | []     -> Delimcc.shift prompt0 (fun k -> Request k)
  | t1::q1 -> t1::app q1;;

(* push_prompt prompt0 remplace reset *)
let append l1 l2 =
  match Delimcc.push_prompt prompt0 (fun () -> Done (app l1)) with
  | Done res -> assert false
  | Request k -> match k l2 with
```

```
  | Done res  -> res
  | Request _ -> assert false;;
```

4 Application : optimisation de code

▷ **Exercice 1** *Soit la fonction suivante* `prod_int_list` *qui calcule le produit des entiers contenus dans une liste. On veut pouvoir optimiser le code en court-circuitant toutes les multiplications lorsqu'on rencontre un* `0`*. Proposer une solution à l'aide des continuations.*

```
let rec prod_int_list l =
  match l with
  | []   -> 1
  | t::q -> t * prod_int_list q;;
```

5 Application : Resumable Exceptions

Dans un développement classique d'application, on a naturellement envie de séparer le traitement du cas nominal sans erreur des cas exceptionnels, pour des raisons de génie logiciel et de lisibilité du code. Néanmoins, l'usage des exceptions pour ce faire oblige soit l'arrêt de l'application en cas d'erreur, soit des mécanismes compliqués de reprise qui impactent et parasitent la structure du code. Les continuations permettent de définir des exceptions avec reprise possible sur erreur et ainsi de pallier ce problème.

▷ **Exercice 2** *Définir un mécanisme de lecture de la première ligne d'un fichier avec entrée interactive d'un autre nom de fichier en cas d'échec (si le fichier n'existe pas). On veillera à séparer le cas nominal du cas d'erreur. On utilisera les primitives suivantes :*

```
Sys.file_exists : string -> bool
open_in : string -> in_channel
read_line : unit -> string
input_line : in_channel -> string
close_in : in_channel -> unit
```

6 Application : programmation concurrente et Green Threads

Une possibilité intéressante en programmation est de permettre la création et manipulation de *threads*, i.e. de processus légers, au niveau utilisateur, comme une librairie standard, sans faire (trop) appel aux couches systèmes. Cette solution évite les appels systèmes de gestion de processus, réputés lourds, et rend de fait la notion de processus relativement indépendante du système. En effet, la copie (de parties) de la pile d'appels est réalisable simplement, en assembleur ou à l'aide de primitives systèmes simples.

Ces processus utilisateurs, plus économes en ressources système, sont appelés *green threads*. Ils s'apparentent aux coroutines car ils sont non-préemptifs (coopératifs) et peuvent être implantés grâce aux continuations.

On commencera par un cas simple de deux coroutines `ping` et `pong` qui rendent la main à un `scheduler` après chaque affichage d'un message et s'arrêtent après 10 affichages (voir code ci-dessous). Ce `scheduler` se contente de passer le flôt de contrôle de l'une à l'autre dans une boucle de traitement.

▷ **Exercice 3** *Compléter le code suivant afin d'implanter la description du comportement des deux coroutines ci-dessus.*

```
type (`shift, 'reset) res =
| Done of 'reset
```

```
| Request of 'shift -> ('shift, 'reset) res;;
let ping () =
  begin
    for i = 1 to 10
    do
      print_endline "ping !";
    done;
  end;;
let pong () =
  begin
    for i = 1 to 10
    do
      print_endline "pong !";
    done
  end;;
```

▷ **Exercice 4** *Définir maintenant la fonction* `scheduler : unit -> unit` *afin d'implanter la description du comportement du* scheduler *ci-dessus. On pourra gérer une file d'au plus deux processus à exécuter.*

7 Application : coroutines et *yield*

Les continuations permettent de définir la primitive *yield*, présente en Python par exemple, qui permet une forme de coroutine simple. On considère le code suivant qui utilise une primitive `yield` dans la définition d'une fonction `iter` qui renvoie un à un les entiers contenus dans une arbre binaire, de gauche à droite. Ce générateur d'entiers est ensuite utilisé par la primitive `foreach` qui se contente d'appliquer à chaque entier une fonction de type `int -> unit`. On l'utilise ici pour afficher ces entiers.

```
type tree =
| Node of tree * int * tree | Empty;;

let rec iter_tree t =
  match t with
  | Node (l, i, r) -> (iter_tree l; yield i; iter_tree r)
  | Empty          -> ();;

let print t = foreach (fun i -> print_int i) iter_tree t;;
```

▷ **Exercice 5** *Définir les primitives* `yield` *et* `foreach`*, à l'aide de continuations, afin que le code ci-dessus fonctionne comme attendu.*