Continuations: programmer dans le futur

Qu'a-t-on vu de Caml?

- le cœur fonctionnel
  - des fonctions
  - les types somme et le filtrage
- des extensions
  - les exceptions
  - les aspects impératifs
    - références, tableaux
    - effets de bord, séquence, itération
- les modules: essentiellement un espace de *nommage* et d'*encapsulation*
  - les modules se situent 'à un autre niveau'
- fondamentalement, tout peut être compris à l'aide des fonctions
  - uniformité, minimalité
  - raisonner sur les programmes — le λ-calcul

Impératif en fonctionnel

```
let c = ref 1;

let f x =
  c := !c+1;
  if x>0 then 2*x else -2*x

let g k =
  let s = f k in
  s + 2*!c
```

```
let f x r =
  let r' = r+1 in
  (if x>0 then 2*x else -2*x), r'

let g k r =
  let (s,r') = f k r in
  (s + 2*r', r')
```

- une traduction 'systématique'
  - on ajoute un paramètre à toutes les fonctions: l'état du monde
  - celles-ci renvoient une valeur et un nouvel état
  - une fonction est un *transformateur d'état*
  - la version traduite 'ne dépend plus que d'elle-même'
- ça se généralise
  - on associe à une fonction la partie du monde qu'elle modifie
- variations: DÉMO `imp2func.ml`

Après les effets de bord, le contrôle

- pour éliminer les références, on passe aux fonctions l'état courant de la mémoire: le "présent"
  (c'est plus ou moins la transformation inverse qui est utilisée en Haskell)
- le rôle des **exceptions** (ML, Java) est de diriger le *flot* du calcul
  - dans un langage purement fonctionnel, le contrôle est géré par le système, le programmeur n'y a pas accès
    ...ou presque: les exceptions servent à dévier le flot
  - le contrôle est 'beaucoup plus' explicite dans un langage impératif
    - structure des programmes (`;`)
    - *instructions de contrôle (return break while goto...)*
- idée des **continuations**: *expliciter le contrôle* en manipulant le **futur** du calcul

Continuations – premiers exemples

- soit la fonction `let f x y = x*y`
  sa version avec continuation est `let f x y k = k (x*y)`
  ↪ on passe le résultat à `k`
- ainsi,

```
let g a b k =
 let t = 2*a in
 k (t*t+b)
```

- ou encore

```
let rec search a l k =
 if is_empty l then k false
 else if (a = head l)
      then k true
      else search a (tail l) k
```

Continuations – du côté de l'appelant

```
let toto x y k =
  let k' = fun v -> k (v*y) in
  titi x (x+1) k'
```

ou encore

```
let toto x y k =
  titi x (x+1) (fun v -> k (v*y))
```

on passe la main à `titi`, dont le résultat sera multiplié par `y`

exemple d'appel:

```
toto 3 5 (fun i -> (print_int i;print_newline()))
```

Continuations - exemples

chercher dans des arbres:

```
let rec search a t =
 if is_empty t then false
 else if (val_node t) = a then true
 else if (search a (left t))
  then true
  else (search a (right t))
```

devient

```
let rec search a t k =
 if is_empty t then (k false)
 else if (val_node t) = a then (k true)
 else
  search a (left t) (fun res ->
   if res then (k true)
   else (search a (right t) k))
```

⇝ tous les appels récursifs deviennent terminaux

Un peu de recul

- à chaque fois que l'on fait `let f a b k = ...(k r) ...`
  à la place de `let f a b = ...r...`
  - on remplace une valeur par un appel de fonction
  - le `k r` 'signifie' `return r` *(cf. en C)*
- typage:

```
# let f x y k = k (x+y);;
val f :  int -> int -> (int -> 'a) -> 'a = <fun>
```

  tout dépend d'un `'a` : type attendu par le futur
  - cf. le type de `raise :  exn -> 'a`
    (le contexte où l'exception va être rattrapée est inconnu)

- adopter le style par continuations, c'est *'réifier' le futur du calcul*
- on peut manipuler *plusieurs futurs possibles*, et ainsi parler d'*exceptions*

Continuations et exceptions

- deux continuations: futur normal, futur exceptionnel

```
let search a t k =
 let rec aux a t break k0 =
   if (is_empty t) then (k0 false)
   else if (val_node t) = a then (break true)
   else aux a (left t) break (fun res -> aux a (right t) break k0)
in aux a t k k
```

- ou, plus concis:

```
let search a t break =
 let rec aux t k =
   if (is_empty t) then (k false)
   else if (val_node t) = a then (break true)
   else aux (left t) (fun x -> aux (right t) k)
in aux t break
```