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
```

- les exceptions sont une forme de **contrôle**, les *itérateurs* en sont une autre

Continuations, suite

- impératif en fonctionnel:
  - état (cases mémoire) $\leftrightarrow$ paramètres de plus

    les fonctions vues comme *transformateurs d'état*
  - itérations $\leftrightarrow$ fonctions récursives
- contrôle en fonctionnel: les continuations

  notamment les exceptions

```
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
```

Contrôle et continuations: itérateurs

- les constructions impératives pour l'itération permettent aussi de "programmer le contrôle"

  `while test do body done`

```
let rec do_while test body k =
 if test()
  then body (fun () -> do_while test body k)
  else k()
```

DÉMO `do_while_cont.ml`

- typage:

```
val do_while :
(unit->bool) -> ((unit -> 'a) -> 'a) -> (unit -> 'a) -> 'a = <fun>
```

test, body, k

Des `break` dans les `while`

- en C, l'instruction `break` permet de sortir directement d'une boucle `while` (indépendamment de la condition booléenne dudit `while`)
- légère modification:

```
let rec do_while test body k =
 if test()
 then body (fun () -> do_while test body k)then body k (fun () ->
do_while test body k)
 else k()
```

- on passe `k` à `body`
  - ici, `k()` joue le rôle du `break` en C
  - noter que le type de `body` change
    - un argument de plus
      futur "normal", futur en cas de sortie brutale

Un style de programmation

- les continuations sont une affaire de *style*
- possibilité de *changer de destin*
  - on choisit parmi des destins
  - on ne fait que des appels de fonctions (pas de `jump`, `goto`)
  - une vision uniformisée des opérateurs de contrôle (exceptions, itérateurs, ...)

  $\hookrightarrow$ *"banalisation de l'héroïsme"*
- les continuations peuvent également être utilisées comme *technique de compilation*

Compiler avec les continuations

Transformations de programmes

- le compilateur de *Standard ML* travaille sur la forme CPS *(continuation passing style)* des termes
- ceci est facilité par des transformations de programmes telles que le *lambda lifting:*

```
let f x y =                    let g t y' = t+y'
  let g t = t+y          ⇝     let f x y =
in (g x)+(g y)                   (g x y)+(g y y)
```

- toutes les fonctions à plat, *sans variable libre*

  ... mais pas (encore) en forme CPS
- une "auto-compilation" de ML
  - moins lisible
  - plus simple à exécuter

Continuations et compilation

DÉMO `prodprimes.ml` un style proche du *langage machine*

```
let rec prodprimes_cps (n,c) =
  if n=1 then c 1 else
    let k b =
      if b then
        let j p =
          let a = n*p in c a
        and m=n-1 in
          prodprimes_cps (m,j)
      else
        let h q = c qlet h q = c
qlet  h q = c q
        and i=n-1 in
          prodprimes_cps (i,h)
    in
      isprime (n,k)
```

`b p a m q i`
noms temporaires
(*registres*)

`c k j h`: continuations
adresses où "sauter"

`h q = c q`
optimisation: raccourci

```
else
  let i = n-1 in
    prodprimes_cps (i,c)
in ...
```

Compilation par continuations – principes

- les arguments "de travail" d'une fonction sont *atomiques*
  $\longrightarrow$ registres, *les arguments sont prêts*
- autres arguments: une ou plusieurs *continuations*
- $\hookrightarrow$ des "bouts de code"
  - élémentaires
  - purement fonctionnels
- tous les appels sont *terminaux*: une fonction "passe toujours la main": pas besoin de pile

"Exceptions" en C – sorties non locales

Principe

- `return`: sortir de la fonction courante (un niveau)
- mécanisme de sauvegarde du contexte:
  - `jmp_buf` type de donnée pour la sauvegarde de l'état d'exécution $\longrightarrow$ registre de pile *sp*, registre de programme *pc*, autres registres (variable registers)
  - `int setjmp (jmp_buf)`
    stocke la valeur de l'état courant dans le `jmp_buf`
    renvoie 0, *sauf si on y arrive depuis un* `longjmp`
  - `void longjmp (jmp_buf, int)`
    a pour effet de revenir au `setjmp` correspondant
- les appels à `setjmp` sont à manier avec précaution (ne pas tenter de faire trop de finesses) $\longrightarrow$ cf `info libc`
  `Non-Local Exits`

Un exemple

```
#include <setjmp.h>
#include <stdlib.h>
#include <stdio.h>
jmp_buf main_loop;

int main (void){
  while (1)
    if (setjmp (main_loop))    /* ≠0 si vient d'un longjump */
      printf ("Back at main loop....\n");
    else
      do_command ();
  ...}

void do_command (void){
⋮
    longjmp (main_loop, -1); /* si quelque chose rate..  */
...  }
```

Quelques remarques

- `if (setjmp(main_loop)) {...} else {...}`
  cela ressemble à `try...with` → macros
- on doit utiliser `longjmp` avec une valeur $\neq 0$
- les objets dans la pile situés au-dessus *(en-dessous...)* du niveau spécifié par le `jmp_buf` sont perdus

  plus généralement, un point de retour n'est valide que durant la vie de la fonction où il est défini
- situations d'utilisation de `setjmp / longjmp`
  - plusieurs boucles imbriquées
  - parcours de structures de données récursives: *backtrack*

    en gros, comme pour les exceptions