Cours 7 : Les continuations

2020 - 2021

Thèmes

- la notion de continuation, applicable à (presque) tous les langages
- la transformation CPS
- les continuations natives
- usages: inversion de contrôle, coroutines, processus utilisateurs, compilation, etc

La notion de continuation

- permet de représenter ce qui va s'exécuter **après** le code courant
- i.e. ce qui *continue* l'exécution
- en programmation fonctionnelle, une continuation est une fonction
- paramètre supplémentaire du code courant

Introduction de continuation: principe

- supposons `f : a -> b`, où l'on souhaite introduire une continuation
- `f` peut être suivie de `g : b -> c` quelconque, son "futur"
- on transforme donc `f` en `kf : a -> (b -> c) -> c`, intégrant ce futur
- le type `b` est devenu `(b -> c) -> c`, en fait très proche
- lien étroit avec la logique, où A est proche de $\neg\neg A = (A \to \bot) \to \bot$

Introduction de continuation: exemples

- transformation de programme appelée *Continuation-Passing Style*
- la factorielle (avec variable intermédiaire):

```
let rec fact n =
 if n = 0 then 1 else let fact_n_1 = fact (n-1) in 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);;
```

- Fibonacci (à réécrire avec variables intermédiaires):

```
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);;
```

Transformation CPS

- récursivité terminale ($-$ de pile *vs.* $+$ de mémoire)
- mieux contrôler/changer/linéariser le flôt de contrôle d'un programme
- utile par exemple pour écrire un compilateur (gestion du flôt de contrôle)

Exemple: la sémantique des exceptions

- on souhaite implanter/représenter la sémantique des exceptions
- une exécution peut terminer (succès) ou lever une exception (échec)
- on ajoute 2 continuations : k_-, d'échec; k_+, de succès
- pour chacune des constructions du langage de départ
- auquelles on ajoute le traitement des exceptions

$$
\begin{array}{ll}
[\![\textbf{try}\ \ expr_1\ \ \textbf{with}\ \mathsf{Exc\ v} \mathrel{->}\ \ expr_2]\!]\ \mathsf{k}_-\ \mathsf{k}_+ & \triangleq [\![expr_1]\!]\ (\textbf{fun}\ \mathsf{v} \mathrel{->} [\![expr_2]\!]\ \mathsf{k}_-\ \mathsf{k}_+)\ \mathsf{k}_+ \\
[\![\mathsf{raise\ (Exc\ v)}]\!]\ \mathsf{k}_-\ \mathsf{k}_+ & \triangleq \mathsf{k}_-\ \mathsf{v} \\
[\![(\textbf{fun}\ \mathsf{v} \mathrel{->} expr)]\!]\ \mathsf{k}_-\ \mathsf{k}_+ & \triangleq (\textbf{fun}\ \mathsf{v} \mathrel{->} [\![expr]\!]\ \mathsf{k}_-\ \mathsf{k}_+) \\
[\![(fn\ ar)]\!]\ \mathsf{k}_-\ \mathsf{k}_+ & \triangleq [\![ar]\!]\ \mathsf{k}_-\ (\textbf{fun}\ \mathsf{va} \mathrel{->} [\![fn]\!]\ \mathsf{k}_-\ (\textbf{fun}\ \mathsf{vf} \mathrel{->} \mathsf{vf\ va})) \\
[\![val]\!]\ \mathsf{k}_-\ \mathsf{k}_+ & \triangleq \mathsf{k}_+\ val
\end{array}
$$

Continuations natives

- la transformation CPS est complexe et intrusive (changement du code source)
- les continuations natives existent dans de nombreux langages : `call/cc` en Scheme
- ou bien des primitives inspirées de celles-ci : `setjmp`/`longjmp` en C, `yield` en Python, Scala, Racket, Haskell, etc
- ces primitives permettent de faire comme si on avait ajouté des paramètres supplémentaires de continuation dans le code des fonctions
- on étudiera une librairie de continuations (dites délimitées)

La librairie Delimcc (Oleg Kiselyov)

- l'implantation des continuations implique la recopie d'une partie de la pile d'appels
- new_prompt: unit -> 'a prompt, pour créer un marqueur de pile
- push_prompt: 'a prompt -> (unit -> 'a) -> 'a, pour délimiter l'usage des continuations, en insérant un marqueur dans la pile
- shift : 'a prompt -> (('b -> 'a) -> 'a) -> 'b, pour capturer la continuation courante, i.e. la pile depuis le marqueur jusqu'à l'appel en cours
- l'exécution de : push_prompt p (**fun** () -> ... (shift p (**fun** k -> *expr*)) ...)
  1. interrompt l'exécution en cours, lorsque doit être évaluée la sous-expression (shift ...) ci-dessus
  2. capture dans la variable k l'état courant (notamment la pile). (k *arg*) *reprendrait* alors l'évaluation de push_prompt p (**fun** () -> ... (*arg*) ...)
  3. exécute *expr* (dépendant de k ou non), qui sera le résultat global de toute l'expression push_prompt ...
- dans d'autres langages, push_prompt et shift apparaissent sans marqueur de pile p et sont renommées reset et shift
- il existe d'autres primitives, mais reset / shift sont suffisantes

Utilisation des continuations natives

- très proche des continuations explicites (mais **pas** équivalentes)
  - le type des expressions/fonctions ne change pas (pas de paramètre supplémentaire)
  - 2 types importants (et parfois $\neq$) : celui de l'expression (reset ...) et celui de la sous-expression (shift ...)
  - pas d'introduction de récursivité terminale, pas d'économie de pile d'appels
- souvent appelées : *resumable exceptions*
  - calcul interrompu, comme pour une exception
  - mais on peut le reprendre, en utilisant la continuation k
  - reset (**fun** () $->$ ... (shift (**fun** k $->$ *expr*)) ...) est un "**try** ... (raise (Exc k)) ... **with** Exc k $->$ *expr*"
  - dans le traitement d'erreur *expr*, on peut fournir à k une valeur qui remplace le (raise (Exc k)) et qui redémarre l'exécution "normalement" à l'endroit où le raise l'a arrêtée

Utilisation des continuations natives: la factorielle

- différentes versions de la factorielle:

```
(* version sans continuations *)
let rec fact n = if n = 0 then 1 else let fact_n_1 = fact (n-1) in n * fact_n_1 ;;
(* version avec continuations explicites *)
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);;
(* version avec continuations implicites/natives *)
let rec fact n =
 if n = 0 then shift (fun k -> k 1) else let fact_n_1 = fact (n-1) in n * fact_n_1 );;
let fact' n = reset (fun () -> fact n);;
```

- seul l'usage de k (i.e. (k 1)) nécessite réellement sa capture avec shift
- la continuation (fun fact_n_1 -> k (n * fact_n_1)) est le futur de fact (n-1)
- les continuations natives construisent implicitement cette valeur

Utilisation des continuations natives: la concaténation de listes

- différentes mises en forme de la fonction append:

```
let rec append l1 l2 =
  match l1 with
  | []      -> l2
  | t1::q1 -> t1::append q1 l2;;
(* fonction recursive locale, sans 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 ;;
(* avec variable intermediaire *)
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 ;;
```

Utilisation des continuations natives: la concaténation de listes

- introduction des continuations:

```
(* avec continuations explicites *)
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);;
(* avec continuations natives. Attention, n'economise pas la pile *)
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 );;
```

- la continuation k correspond à la fonction (**fun** l -> t_1::. . .::t_n::l) où [t_1::. . .::t_n] est la liste l1
- tous les éléments t_i parcourus sont stockés dans la pile d'appels capturée par k

Utilisation des continuations natives: la concaténation de listes

- usage très contraint, i.e. nécessairement `k : 'a list -> 'a list`

```
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 );;
```

- peut-on découpler le parcours de la concaténation à `l2` ?
- oui, à condition de changer les types:

```
type ('shift, 'reset) res =
  | Done    of 'reset
  | Request of ('shift -> ('shift, 'reset) res );;
```

- **type** `('shift, 'reset) res` est le nouveau type de retour
- `Done` **of** `'reset` représente les calculs normaux
- `Request` **of** `('shift -> ('shift, 'reset) res)` représente les calculs interrompus par un `shift`

Utilisation des continuations natives: la concaténation de listes

- le type de retour:

```
type ('shift, 'reset) res =
  | Done    of 'reset
  | Request of ('shift -> ('shift, 'reset) res );;
```

- le parcours seul de l1 :

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

- la concaténation à l2:

```
let append l1 l2 =
  match reset (Done (app l1)) with
  | Done _    -> assert false (* necessairement un shift dans app *)
  | Request k -> match k l2 with
                 | Done res -> res
                 | Request _ -> assert false (* un seul shift *) ;;
```

- l'ambiguïté sur le (type de) retour crée des cas de filtrages inutiles

Application: *resumable exceptions*

- séparation des cas nominaux et des cas d'erreurs
- récupération des erreurs: compliqué avec exceptions ordinaires
- on souhaite par exemple lire la première ligne d'un fichier, avec entrée d'un autre nom et reprise en cas d'erreur
- traitements séparés:

```
(* cas nominal de lecture *)
let cas_nominal nom =
  Format.printf "tentative de lecture du fichier '%s'@." nom;
  let f = open_in (if Sys.file_exists nom then nom else shift (fun k -> Request k)) in
  let l = input_line f in
  close_in f;
  Done l;;
(* cas d'erreur de lecture *)
let cas_erreur nom k =
  Format.printf "fichier '%s' inexistant, entrez un autre nom@." nom;
  k (read_line ());;
```

- `shift (fun k -> Request k)` remplace le nom du fichier à ouvrir
- `cas_nominal`: `string -> (string,string) res`
- `cas_erreur`: `string -> (string -> (...) res) -> (...) res`

Application: *resumable exceptions*

- séparation des cas nominaux et des cas d'erreurs
- récupération des erreurs: compliqué avec exceptions ordinaires
- on souhaite par exemple lire la première ligne d'un fichier, avec entrée d'un autre nom et reprise en cas d'échec
- programme principal combinant les différents traitements:

```
let lecture_ligne_interactive nom =
  match reset (fun () -> cas_nominal nom) with
  | Done l      -> l
  | Request k ->
    match cas_erreur nom k with
    | Done l -> l
    | _      -> assert false (* pas de shift dans le traitement d'erreur *) ;;
```

- le traitement d'erreur n'est pas réentrant
- il faut différents constructeurs Request*i* pour gérer des erreurs différentes
- on aurait alors la déclaration de type: type ('shift1, 'shift2, ..., 'reset) res

Application: *resumable exceptions*

- on souhaite maintenant lire un entier
- extension au traitement de plusieurs erreurs:

```
(* cas nominal de lecture *)
let cas_nominal nom =
  Format.printf "tentative de lecture du fichier '%s'@." nom;
  let f = open_in (if Sys.file_exists nom then nom else shift (fun k -> Request1 k)) in
  let i = try input_binary_int f with _ -> shift (fun k -> Request2 k) in
  close_in f;
  Done i;;
(* cas d'erreur de lecture : fichier inexistant *)
let cas_erreur_fichier nom k =
  Format.printf "fichier '%s' inexistant, entrez un autre nom@." nom;
  k (read_line ());;
(* cas d'erreur de lecture : pas d'entier dans le fichier *)
let cas_erreur_entier k =
  Format.printf "contenu incorrect, entrez un entier@.";
  k (read_int ());;
```

- cas_nominal: string -> (string, string, int) res
- cas_erreur_fichier : string -> (string -> (...) res) -> (...) res
- cas_erreur_entier : (int -> (...) res) -> (...) res

Application: *resumable exceptions*

- avec le type suivant qui permet de traiter 2 types d'erreur:

```
type ('shift1, 'shift2, 'reset) res =
| Request1 of ('shift1 -> ('shift1, 'shift2, 'reset) res)
| Request2 of ('shift2 -> ('shift1, 'shift2, 'reset) res)
| Done     of 'reset
```

- ou bien directement le type spécialisé:

```
type res = Request1 of (string -> res) | Request2 of (int -> res) | Done of int
```

- Request1 correspond à l'erreur de fichier (pas de fichier)
- Request2 correspond à l'erreur de contenu de fichier (pas d'entier)
- Done correspond à l'exécution nominale, qui renvoie l'entier lu dans le fichier

Application: *resumable exceptions*

- les traitements d'erreurs doivent toujours rester sous la portée d'un reset
- on a intérêt à factoriser ces traitements et éviter les **match** . . . **with** . . . imbriqués
- programme principal:

```
let traitement_erreur_entier prog =
  match reset (fun () -> prog ()) with
  | Request2 k -> cas_erreur_entier k
  | Request1 _ -> assert false (* erreur recuperee plus tot *)
  | Done i     -> Done i;;
let traitement_erreur_fichier prog nom =
  match reset (fun () -> prog nom) with
  | Request1 k -> cas_erreur_fichier k
  | resultat   -> resultat;;
let lecture_ligne_interactive nom =
  match traitement_erreur_entier
        (fun () -> traitement_erreur_fichier cas_nominal nom) with
  | Done i -> i
  | _      -> assert false (* toutes les erreurs sont recuperees plus tot *) ;;
```

Application: inversion de contrôle

- soit un évaluateur d'expressions arithmétiques
- construit avec *parser* et *lexer* (`ocamlyacc` et `ocamllex`)
- fichier `parser_expression.mly`: les tokens

```
%token PLUS MINUS
%token TIMES MOD DIV
%token LPAR RPAR
%token LET EQUAL IN
%token <int> NUMBER
%token <string> IDENT
%token END
%nonassoc IN
%left PLUS MINUS
%left TIMES MOD DIV
/* Type renvoye pour le nom terminal fichier */
%type <(string * int) list -> int> fichier
/* Le non terminal fichier est l'axiome */
%start fichier
```

Application: inversion de contrôle

- soit un évaluateur d'expressions arithmétiques
- construit avec *parser* et *lexer* (`ocamlyacc` et `ocamllex`)
- fichier `parser_expression.mly`: les règles de production

```
fichier : expression END { $1 }

expression :
    expression PLUS expression                  { (fun env -> $1 env + $3 env) }
  | expression MINUS expression                 { (fun env -> $1 env - $3 env) }
  | expression TIMES expression                 { (fun env -> $1 env * $3 env) }
  | expression DIV expression                   { (fun env -> $1 env / $3 env) }
  | expression MOD expression                   { (fun env -> $1 env mod $3 env) }
  | IDENT                                       { (fun env -> List.assoc $1 env) }
  | NUMBER                                      { (fun env -> $1) }
  | LET IDENT EQUAL expression IN expression    { (fun env -> $6 (($2, $4 env)::env)) }
  | LPAR expression RPAR                        { $2 }
```

- les variables sont définies dans un environnement env

Application: inversion de contrôle

- soit un évaluateur d'expressions arithmétiques
- construit avec *parser* et *lexer* (`ocamlyacc` et `ocamllex`)
- fichier `lexer_expression.mll`:

```
{ open Parser_expression
  exception LexicalError of string
}
rule main = parse
  | ['\n' '\t' ' ']+          { main lexbuf }
  | "+"                       { PLUS }
  | "-"                       { MINUS }
  | "*"                       { TIMES }
  | "/"                       { DIV }
  | "%"                       { MOD }
  | "="                       { EQUAL }
  | "let"                     { LET }
  | "in"                      { IN }
  | "("                       { LPAR }
  | ")"                       { RPAR }
  | ['0'-'9']+ as texte       { NUMBER (int_of_string texte) }
  | ['a'-'z''A'-'Z']+ as texte { IDENT texte }
  | eof                       { END }
  | _ as texte                { raise (LexicalError texte) }
```

Application: inversion de contrôle

- soit un évaluateur d'expressions arithmétiques
- construit avec *parser* et *lexer* (`ocamlyacc` et `ocamllex`)
- programme principal "ordinaire":

```
let parse_and_eval lexbuf =
  Parser_expression.fichier Lexer_expression.main lexbuf [];;
```

- l'exécution est monolithique: quid de la récupération d'erreur ?
- les codes sources du *parser* et du *lexer* sont non modifiables (regénérés à chaque fois)
- on aimerait une solution où les tokens sont lus incrémentalement
- avec possibilité de revenir en arrière, de changer de flux d'entrée, etc

Application: inversion de contrôle

- soit un évaluateur d'expressions arithmétiques
- construit avec *parser* et *lexer* (`ocamlyacc` et `ocamllex`)
- programme principal avec inversion de contrôle: définitions

```
let parse_and_eval_inv lexbuf =
  let lexer_inv lexbuf =
    ( shift (fun k -> Request k); Lexer_expression.main lexbuf) in
  reset (fun () -> Done (Parser_expression.fichier lexer_inv lexbuf []));;

let next_token =
  function
  | Done res ->
      begin
        Format.printf "Resultat = %d@." res;
        Done res
      end
  | Request k -> k ();;
```

- initialisation: `parse_and_eval_inv : Lexing.lexbuf -> (unit, int) res`
- passage au token suivant: `next_token : (unit, int) res -> (unit, int) res`

Application: inversion de contrôle

- soit un évaluateur d'expressions arithmétiques
- construit avec *parser* et *lexer* (`ocamlyacc` et `ocamllex`)
- programme principal avec inversion de contrôle: exécution

```
let lexbuf = Lexing.from_channel (open_in Sys.argv.(1));;
(* etat initial : aucun token lu *)
let state0 = parse_and_eval_inv lexbuf ;;
(* etat du parser apres 1 token lu *)
let state1 = next_token state0 ;;
(* etat du parser apres 2 tokens lus *)
let state2 = next_token state1 ;;
(* etat du parser apres 3 tokens lus *)
let state3 = next_token state2 ;;
...
```

- state0, state1, . . ., sont de type `(unit, int) res`