Paradigmes de programmation

Cours 5 : Programmation fonctionnelle avancée

Benoît Montagu — benoit.montagu@inria.fr

Préparation à l'agrégation d'informatique — Automne 2022

Plan du cours

Au programme de ce cours :

- Retour sur les fonctions récursives terminales
- Continuations, style en passage par continuation
- Défonctionnalisation
- Suspensions : des fonctions dans les structures de données
- Paresse, structures de données paresseuses

Fonctions récursives terminales

Récursion terminale

Définition

Une fonction est récursive terminale si tous ses appels récursifs sont terminaux.
En anglais : « *tail call* », « *tail recursion* »

- Une fonction récursive terminale peut être compilée en une boucle
- Elle consommera alors un espace de pile constant
- Considérons une fonction récursive non terminale `f` et une version équivalente `f_tailrec` qui soit récursive terminale :
- ❓ Laquelle des deux s'exécute le plus rapidement ?
- On ne peut généralement pas conclure !
- Si `f_tailrec` alloue dans le tas plus de données que `f`, il y a fort à parier que `f_tailrec` soit plus lente…
- ⚠ On écrit en style récursif terminal pour pouvoir traiter des données de grande taille. On ne gagne en général pas en efficacité.
- ⚠ Un programme récursif terminal qui n'alloue pas de mémoire peut se révéler plus efficace que sa version non récursive terminale.

Comment transformer un programme en style récursif terminal ?

- On va décrire une méthode systématique pour effectuer cette transformation
- Elle procède en 2 étapes :
  1. Transformation en « passage par continuation »
  2. Défonctionnaliser les continuations
- Après l'étape 1 : on a déjà une fonction récursive terminale, mais elle introduit des clôtures intermédiaires
- Après l'étape 2 : on a éliminé ces clôtures

Continuations

Qu'est-ce qu'une continuation ?

- Une continuation est une fonction qui décrit comment continuer un calcul
- Elle prend en paramètre un résultat intermédiaire et produira le résultat final
- Par exemple : dans l'expression `(3 + 2) * 7`, un résultat intermédiaire est l'évaluation du sous-arbre gauche `3 + 2` et la contination la fonction `fun n -> n * 7`

Programmer en style « passage par continuation »

- En anglais : CPS, pour « continuation passing style »
- Un programme en style CPS prend en paramètre sa continuation
- On note habituellement les continuations avec la lettre k

Exemple :

```
1  let const n k = k n (* programme retournant une constante, en CPS *)
2  let plus n1 n2 k = k (n1 + n2) (* addition en CPS *)
3  let mult n1 n2 k = k (n1 * n2) (* multiplication en CPS *)
4  let expr k = (* `(3 + 2) * 7' en CPS *)
5    const 3 (fun n ->
6      const 2 (fun m ->
7        plus n m (fun nm ->
8          const 7 (fun p ->
9            mult nm p k))))
10 (* val expr : (int -> 'a) -> 'a *)
11 let result = expr (fun x -> x)
12 (* result = 35 *)
```

Quelques propriétés des programmes CPS

Un programme en CPS :

- Crée des clôtures intermédiaires
- Donne un nom à chaque calcul intermédiaire
  (ce nom est le paramètre des continuations)
- Fixe l'ordre dans lequel les calculs doivent avoir lieu
- Ne contient que des appels terminaux
- Peut donc s'exécuter dans un espace de pile constant
  (si les appels terminaux sont optimisés)
- A un type de la forme $\forall\alpha.\ \tau_1 \to (\tau_2 \to \alpha) \to \alpha$
  où τ_1 est le type du paramètre du programme, et
  où τ_2 est le type de la valeur que l'on souhaite calculer
- Est donc toujours un programme d'ordre supérieur
- Finit toujours par être appliqué à la continuation identité : `fun x -> x`

Comment transformer un programme en CPS ?

- Étant donné une fonction de départ `f` (dite en « style direct »)
- On veut écrire une fonction `f_cps` qui prend une continuation en paramètre, et qui calcule le même résultat que `f`
- Spécification : pour toute valeur `v` et toute continuation `k`,
  ```
  f_cps v k = k (f v)
  ```
- Cette spécification implique la propriété de correction attendue :
  ```
  f_cps v (fun z -> z) = f v
  ```
- Cette spécification :
  - Doit vous guider dans l'écriture de `f_cps`
  - Est l'invariant à maintenir à chaque appel récursif de `f_cps`

Transformation CPS : exemple sur un programme

Considérons la fonction `map_ib` sur les listes :

```
let rec map_ib (f: int -> bool) : int list -> bool list = function
| [] -> []
| x :: xs ->
  let y = f x in
  let ys = map_ib f xs in
  y :: ys
```

Sa transformée en CPS est :

```
let rec map_ib_cps (f: int -> bool) (l: int list)
  (k: bool list -> 'a) : 'a = match l with
| [] -> k []
| x :: xs ->
  let y = f x in
  map_ib_cps f xs (fun ys -> k (y :: ys))

let map_ib2 f l = map_ib_cps f l (fun ys -> ys)
```


Qu'avons-nous obtenu ?

- Une fonction récursive terminale
- Qui calcule le même résultat que la fonction initiale
- Mais qui est d'ordre supérieur, et qui produit des clôtures dynamiquement
- On ne peut pas aisément implémenter `map_ib_cps`
  dans des langages comme C
- Prochaine étape :
  revenir à une fonction d'ordre 1, sans clôtures intermédiaires

Défonctionnalisation

Défonctionnalisation

Buts : éliminer les clôtures intermédiaires

Dans notre cas : éliminer les continuations de la forme `(fun x -> ...)` que nous avons introduites au cours de la mise en CPS

Principes :

- *Réifier* les continuations :
  C-à-d définir un type de données `kont` pour représenter les continuations
- Implémenter une fonction `apply` qui applique une continuation :
  C-à-d `apply kont v = k v` lorsque `kont` représente la continuation `k`

Comment effectuer la défonctionnalisation ?

Étant donné notre fonction en CPS `f_cps`, on veut écrire sa version défonctionnalisée `f_defun`, qui a le même comportement que `f_cps`.

- On garde la même structure que le code de `f_cps`
- Définir un type algébrique `kont` qui a autant de cas que l'on a introduit de continuations (`fun x -> ...`) durant la mise en CPS
- Cela inclut la continuation identité utilisée tout à la fin ! (il faut un cas `KId`)
- Les arguments des constructeurs du type `kont` sont les *variables libres* des continuations
- Définir la fonction `apply`, ayant pour type $\texttt{kont} \to \tau_2 \to \tau_2$
- Remplacer les *appels* à des continuations par des appels à la fonction `apply`
- Remplacer les continuations (`fun x -> ...`) par leur version réifiée, de type `kont`
- Invariant : pour toute valeur `v`, pour toute continuation réifiée `kont`,
  `f_defun v kont = f_cps v (apply kont)`

Défonctionnalisation sur notre exemple `map_ib` (1)

Code obtenu après transformation CPS :

```
let rec map_ib_cps (f: int -> bool) (l: int list)
  (k: bool list -> 'a) : 'a = match l with
| [] -> k []
| x :: xs ->
  let y = f x in
  map_ib_cps f xs (fun ys -> k (y :: ys))

let map_ib2 f l = map_ib_cps f l (fun ys -> ys)
```

Étape nº 1 : on réifie les continuations, et on définit `apply` :

```
type kont =
| KId
| KCons of kont * bool (* pour les variables libres k et y *)

let rec apply kont (ys: bool list) : bool list = match kont with
| KId -> ys
| KCons (kont', y) -> apply kont' (y :: ys)
```

Défonctionnalisation sur notre exemple `map_ib` (2)

Code obtenu après transformation CPS :

```
let rec map_ib_cps (f: int -> bool) (l: int list)
  (k: bool list -> 'a) : 'a = match l with
| [] -> k []
| x :: xs ->
  let y = f x in
  map_ib_cps f xs (fun ys -> k (y :: ys))
let map_ib2 f l = map_ib_cps f l (fun ys -> ys)
```

Étape n° 2 : on utilise `apply` et les constructeurs de continuations :

```
let rec map_ib_defun (f: int -> bool) (l: int list) kont : bool list =
match l with
| [] -> apply kont []
| x :: xs ->
  let y = f x in
  map_ib_defun f xs (KCons (kont, y))
let map_ib3 f l = map_ib_defun f l KId
```


Correction de CPS + défonctionnalisation

On a jusqu'ici les propriétés suivantes :

- Invariant de défonctionnalisation :

```
f_defun v kont = f_cps v (apply kont)
```
(1)

- Correction de la réification de l'identité :

```
apply KId v = v
```
(2)

- Invariant de mise en CPS :

```
f_cps v k = k (f v)
```
(3)

On en déduit :

```
f_defun v KId = f_cps v (apply KId)     (* via équation (1) *)
              = apply KId (f v)         (* via équation (3) *)
              = f v                     (* via équation (2) *)
```

La transformation CPS suivie de la défonctionnalisation est donc correcte

Exercices

Exercice 1 :

- Transformer la fonction qui calcule factorielle en CPS
- Effectuer sa défonctionnalisation
- Comparer à la fonction récursive terminale « usuelle » de la factorielle : Que constatez-vous ? Commentez

Exercice 2 :

- Définir un type des expressions arithmétiques (addition, soustraction) qui ne contiennent pas de variables
- Définir un interpréteur en style direct qui évalue une expression arithmétique
- Le transformer en CPS, puis le défonctionnaliser
- Quelles différences observez-vous dans la structure du code vis-à-vis de la transformation effectuée sur `map_ib` ?

❗ Faites vraiment ces exercices

Pour aller plus loin

Olivier Danvy (2008). « Defunctionalized Interpreters for Programming Languages ». In :
Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming
Sous la dir. de James Hook et Peter Thiemann. ACM, p. 131-142

- Explique comment différents styles de sémantiques de langages de programmation sont reliés entre eux
- En utilisant la conversion en CPS et la défonctionnalisation
- Ainsi que quelques autres transformations de programmes

Suspensions

Itérateurs, classiquement en OCaml

```
type 'a tree =
| E
| N of 'a tree * 'a * 'a tree

(* Fonction d'itération "classique" sur les arbres *)
let rec tree_iter f = function
| E -> ()
| N (l, x, r) -> f x; tree_iter f l; tree_iter f r
```

C'est la fonction `iter` qui a le contrôle, pas le client :

```
(* Exemple de client, avec contrôle par la fonction d'itération *)
let () =
  print_endline "Iteration: controlled by the iterator";
  tree_iter (fun i ->
      print_int i; print_newline ()
    )
    t;
  print_endline "No more!"
```


Itérateurs, dans un style à la Java (1/3)

Dans ce style :

- Un itérateur est une valeur, qui a un état interne
- C'est le client qui a le contrôle

```
(* Type des itérateurs *)
type 'a iterator = { next: unit -> 'a }

(* Exception levée lorsqu'on demande un élément à un itérateur vide *)
exception Done

(* [next iterator] renvoie le prochain élément disponible, ou lance
   l'exception [Done]. Simultanément, l'état de l'itérateur est mis à
   jour. *)
let next iterator = iterator.next ()
```


Itérateurs, dans un style à la Java (2/3)

Dans ce style :

- Un itérateur est une valeur, qui a un état interne
- C'est le client qui a le contrôle

```
(* Crée un itérateur à partir d'un arbre *)
let from_tree t =
  (* État: la liste des arbres sur lesquels il reste à itérer *)
  let state = ref [t] in
  let rec next () = match !state with
    | [] -> raise Done
    | E :: ts -> state := ts; next ()
    | N (l, x, r) :: ts -> state := l :: r :: ts; x
  in
  { next = next }
```

Itérateurs, dans un style à la Java (3/3)

Dans ce style :

- Un itérateur est une valeur, qui a un état interne
- C'est le client qui a le contrôle

```
(* Exemple de client, avec contrôle par le client *)
let () =
  print_endline "Iteration: controlled by the client";
  let iterator = from_tree t in
  try
    while true do
      let i = next iterator in
      print_int i; print_newline ()
    done
  with Done -> print_endline "No more!"
```


Suspensions

Problème : comment définir des itérateurs...

- Tels que le contrôle est donné au client ?
- Et de manière *purement* fonctionnelle ?

Réponse : en utilisant des suspensions

Définition

Une suspension est une fonction, dont le but est de **retarder** un calcul

En OCaml, ce sera une fonction de la forme `fun () -> ...`

Dans ce cours : on enregistre des suspensions dans des structures de données

Listes suspendues

- On va définir des listes dont le calcul de la valeur est suspendu
- Autres dénominations : listes retardées, cascades, séquences...
- Elles sont utilisées dans le module `Seq` de la bibliothèque standard OCaml

```
type 'a seq = unit -> 'a node
and 'a node =
  | Nil
  | Cons of 'a * 'a seq

let next s = match s () with
| Nil -> None
| Cons (x, xs) -> Some (x, xs)
(* val next : 'a seq -> ('a * 'a seq) option *)
```


Listes suspendues : utilisation

On reprend le type des arbres vu précédemment :
on veut définir la liste des éléments d'un arbre, en retardant son calcul

```
type 'a tree =
| E
| N of 'a tree * 'a * 'a tree

(* Concatène deux listes suspendues *)
let rec concat l1 l2 = fun () -> match l1 () with
| Nil -> l2 ()
| Cons (x, xs) -> Cons (x, concat xs l2)
(* val concat : 'a seq -> 'a seq -> 'a seq *)

(* Crée la liste suspendue des éléments d'un arbre *)
let rec from_tree t = fun () -> match t with
| E -> Nil
| N (l, x, r) -> Cons (x, concat (from_tree l) (from_tree r))
(* val from_tree : 'a tree -> 'a seq *)
```

Listes suspendues : client

On peut maintenant utiliser cette liste suspendue pour itérer sur les éléments :

```
let () =
  print_endline "Iteration: controlled by the client";
  let rec browse iterator = match next iterator with
  | Some (i, rem) -> print_int i; print_newline (); browse rem
  | None -> print_endline "No more!"
  in
  browse (from_tree t)
```

Avec l'approche des itérateurs « à la Java » on avait le code suivant :

```
let () =
  print_endline "Iteration: controlled by the client";
  let iterator = from_tree t in
  try
    while true do
      let i = next iterator in
      print_int i; print_newline ()
    done
  with Done -> print_endline "No more!"
```


Listes suspendues : combinateurs (1/2)

On peut transformer une séquence avec `map` :
Les opérations ne seront effectuées que lorsque la séquence sera *consultée*

```
let rec map f s = fun () -> match s () with
| Nil -> Nil
| Cons (x, s') -> Cons (f x, map f s')
(* val map : ('a -> 'b) -> 'a seq -> 'b seq *)
```

On peut sélectionner *certains* éléments d'une séquence avec `filter` :

```
let rec filter p s = fun () -> match s () with
| Nil -> Nil
| Cons (x, s') ->
  if p x
  then Cons (x, filter p s')
  else filter p s' ()
(* val filter : ('a -> bool) -> 'a seq -> 'a seq *)
```


Listes suspendues : combinateurs (2/2)

On peut combiner deux séquences pour créer une séquence des paires :

```
let rec combine l1 l2 = fun () -> match l1 (), l2 () with
| Nil, _ | _, Nil -> Nil
| Cons (x1, x1s), Cons (x2, x2s) ->
  Cons ((x1, x2), combine x1s x2s)
(* val combine : 'a seq -> 'b seq -> ('a * 'b) seq *)
```

On peut aussi créer une séquence « infinie » :

```
let rec integers_from n = fun () ->
  Cons (n, integers_from (n+1))
(* val integers_from : int -> int seq *)
```

Pour obtenir un iterateur « pythonesque » bien connu :

```
let enumerate l = combine (integers_from 0) l
(* val enumerate : 'a seq -> (int * 'a) seq *)
```


Attention avec les suspensions

Lorque vous écrivez un programme qui utilise les suspensions, soyez attentifs :

- ▶ Raisonner sur la *terminaison* est délicat
- ▶ Savoir *quand* s'effectue quel calcul est difficile
- ▶ Raisonner sur la consommation de *mémoire* est difficile (fuites de mémoire)
- ⚠ Les calculs sont seulement suspendus :
  si vous demandez à *re*faire des calculs suspendus, ils seront *re*faits
- 🛈 suspension ≠ paresse

Paresse

Paresse

L'évaluation paresseuse :

- Permet d'éviter le calcul de ce qui n'est pas strictement nécessaire
- Et permet d'éviter le *re*calcul de ce qui a déjà été calculé
- « paresse = suspension + partage »
- Un calcul paresseux est effectué *au plus une fois*
- C'est une forme de mémoïsation !
- Mot-clef `lazy` en OCaml pour contrôler explicitement l'usage de la paresse :
  - `lazy e` « gèle » l'évaluation de `e` : valeur de type `'a lazy_t`
  - On dit parfois que `lazy` crée un « glaçon »
  - `Lazy.force v` « dégèle » le glaçon `v`
- En OCaml : l'évaluation est par défaut *stricte* (« *eager evaluation* »)
- En Haskell : l'évaluation est par défaut *paresseuse* (« *lazy evaluation* »/ « *call by need* »)

Suspension vs. paresse : exemple

```
let suspendu = fun () -> print_endline "Hello!"
let () =
  suspendu (); suspendu ()
(* "Hello!" est imprimé deux fois *)


let paresseux = lazy (print_endline "Hi!")
let () =
  Lazy.force paresseux; Lazy.force paresseux
(* "Hi!" n'est imprimé qu'une fois *)
```


Paresse : une implémentation (1/4)

```
(* Valeurs suspendues *)
type 'a suspended =
| Delayed of (unit -> 'a)                (* pas encore évalué *)
| Value of 'a                    (* déjà évalué vers une valeur *)
| Exception of exn (* déjà évalué, et ayant levé une exception *)

(* Valeurs lazy: glaçons *)
type 'a deferred = 'a suspended ref
```

Paresse : une implémentation (2/4)

```
(* [make f] crée un glaçon pour le calcul de [f].
   [lazy e] est un raccourci pour [make (fun () -> e)]
*)
let make f = ref (Delayed f)
(* val make : (unit -> 'a) -> 'a deferred *)
```


Paresse : une implémentation (3/4)

```
(* [force v] dégèle le calcul du glaçon [v] *)
let force r =
  match !r with
  | Delayed f -> begin
    try
      let v = f () in
      r := Value v;
      v

    with exn -> begin
      r := Exception exn;
      raise exn
    end
  end
  | Value v -> v
  | Exception exn -> raise exn
(* val force : 'a deferred -> 'a *)
```


Paresse : une implémentation (4/4)

```
(* [force2] : implémentation alternative de [force] *)
let force2 r =
  match !r with
  | Delayed f -> begin
    match f () with
    | v -> begin
      r := Value v;
      v
    end
    | exception exn -> begin
      r := Exception exn;
      raise exn
    end
  end
  | Value v -> v
  | Exception exn -> raise exn
(* val force2 : 'a deferred -> 'a *)
```


Structures de données paresseuses

Une structure de données est paresseuse lorsque :

- Elle contient des informations qui décrivent des calculs retardés
- Ces informations sont calculées petit à petit, à la demande, lorsque des opérations sur ces structures sont effectuées
- Ce sont généralement des structures mutables : les éléments retardés sont *remplacés* par une version où leur calcul a progressé
- Qui ont une interface observationnellement immuable

LE livre de référence :

Chris Okasaki (avr. 1998). Purely Functional Data Structures. Cambridge University Press

- Structures de données fonctionnelles
- Seulement certaines sont paresseuses
- Méthodes pour raisonner sur la complexité amortie : méthode du banquier, méthode du physicien (introduites par Tarjan)

Exemple : lazy skew heaps

Dinesh MEHTA (2005). Handbook of Data Structures and Applications. Boca Raton, Fla : Chapman & Hall/CRC. ISBN : 9781584884354
Chapter 40 : Functional Data Structures (Chris Okasaki)

```
module type ORDERED = sig (* signature des types ordonnés *)
  type t
  val compare: t -> t -> int
end

module type HEAP = sig (* signature des tas *)
  type elt (* type des éléments dans le tas *)
  type t   (* type des tas *)
  val empty: t
  val find_min: t -> elt option
  val merge: t -> t -> t
  val insert: elt -> t -> t
  val delete_min: t -> t
end
```


Exemple : skew heaps, version stricte (1/2)

```
module Strict(X: ORDERED) : HEAP with type elt = X.t =
struct
  type elt = X.t
  type t = Empty | Node of elt * t * t

  let empty = Empty

  let find_min = function
  | Empty -> None
  | Node (x, _, _) -> Some x
```


Exemple : skew heaps, version stricte (2/2)

```
  let rec merge s1 s2 = match s1, s2 with
  | Empty, s | s, Empty -> s
  | Node (x1, l1, r1), Node (x2, l2, r2) ->
    if X.compare x1 x2 < 0
    then Node (x1, merge s2 r1, l1) (* s2 passe à gauche, l1 <-> r1 *)
    else Node (x2, merge s1 r2, l2) (* s1 reste à gauche, l2 <-> r2 *)

  let insert x s =
    merge (Node (x, empty, empty)) s

  let delete_min = function
  | Empty -> Empty
  | Node (_x, l, r) -> merge l r
end
```

- `merge s1 s2` en $O(|s1| + |s2|)$ pire cas et amorti
- `insert x s` en $O(|s|)$ pire cas et amorti

Exemple : skew heaps, version paresseuse (1/2)

```
module Paresseux(X: ORDERED) : HEAP with type elt = X.t =
struct
  type elt = X.t
  type heap = Empty | Node of elt * t * t
  and t = heap lazy_t

  let empty = Lazy.from_val Empty

  let find_min s = match Lazy.force s with
  | Empty -> None
  | Node (x, _, _) -> Some x
```

Exemple : skew heaps, version paresseuse (2/2)

```
let rec merge s1 s2 = match Lazy.force s1, Lazy.force s2 with
| Empty, _ -> s2
| _, Empty -> s1
| Node (x1, l1, r1), Node (x2, l2, r2) ->
  if X.compare x1 x2 < 0
  then Lazy.from_val (Node (x1, merge s2 r1, l1))
  else Lazy.from_val (Node (x2, merge s1 r2, l2))
(* pending merges are left nodes, left args of merge are non-pending *)

let insert x s =
  merge (Lazy.from_val (Node (x, empty, empty))) s

let delete_min s = match Lazy.force s with
| Empty -> s
| Node (_x, l, r) -> merge l r
```

- `insert x s` en $O(\log |s|)$ amorti (subtil, avec méthode du banquier)
- Complexité pire cas inchangée

Exemple : lazy skew heaps

Dinesh Mehta (2005). Handbook of Data Structures and Applications. Boca Raton, Fla : Chapman & Hall/CRC. ISBN : 9781584884354

Dans le chapitre 40 :

- Okasaki donne aussi une implémentation en Java
- Objet mutable, mais observationnellement immuable
- Champ mutable `private boolean pendingMerge` pour tracer quelles opérations `merge` sont retardées
- La structure implémentée est *persistante*
- La complexité amortie est (rapidement) justifiée

Prudence avec la paresse

- ⚠ Comme avec les suspensions, la paresse complique le raisonnement sur les ressources (temps, mémoire)
- `lazy` peut coûter cher : on ne gagne pas à tous les coups à être paresseux
- Mais la paresse peut s'avérer intéressante dans des cas précis
- En pratique, on l'utilise avec parcimonie, et on mesure son coût en pratique
- Le raisonnement formel sur la complexité de programmes paresseux est un sujet de recherche actif

Conclusion

Programmation fonctionnelle avancée : bilan

Éléments abordés dans ce cours :

- Comment rendre un programme toujours récursif terminal ?
- Passage par continuations
- Défonctionnalisation
- Suspensions
- Paresse

Éléments de programmation fonctionnelle non abordés :

- Filtrage de motifs
- Exceptions
- Modules, foncteurs
- GADTs
- Monades…

Prochain (et dernier) cours : programmation objet