Structures de contrôle, cinquième cours

Pratique des effets : des exceptions aux gestionnaires d'effets

Xavier Leroy

2024-02-23

Collège de France, chaire de sciences du logiciel
xavier.leroy@college-de-france.fr

Exceptions

Les exceptions dans un langage fonctionnel

Une exception = une valeur (type `exn`) qui décrit une condition exceptionnelle (erreur, absence de résultat, ...).

Expressions :

$$
\begin{array}{rll}
e ::= & cst \mid x \mid \lambda x.\, e \mid e_1\ e_2 & \\
& \mid \texttt{raise}\ e & \text{levée d'une exception} \\
& \mid \texttt{try}\ e_1\ \texttt{with}\ x \to e_2 & \text{gestionnaire d'exceptions}
\end{array}
$$

`raise` e interrompt l'exécution et se branche au `try ... with` englobant le plus proche. Cette expression ne produit aucune valeur. (Cf. le type $\texttt{raise} : \forall \alpha, \texttt{exn} \to \alpha$.)

Les exceptions dans un langage fonctionnel

Une exception = une valeur (type `exn`) qui décrit une condition exceptionnelle (erreur, absence de résultat, ...).

Expressions :

$$
\begin{array}{rll}
e ::= & cst \mid x \mid \lambda x.\, e \mid e_1\ e_2 & \\
& \mid \texttt{raise}\ e & \text{levée d'une exception} \\
& \mid \texttt{try}\ e_1\ \texttt{with}\ x \to e_2 & \text{gestionnaire d'exceptions}
\end{array}
$$

$\texttt{try}\ e_1\ \texttt{with}\ x \to e_2$ évalue le corps e_1.

Si e_1 ne lève pas d'exception, sa valeur est celle du `try ... with`.

Si e_1 lève une exception, la valeur v de l'exception est liée à x et le gestionnaire (*handler*) e_2 est évalué.

Exemples d'utilisations d'exceptions

Rapport d'erreur (p.ex. un débordement arithmétique) :

```
let safe_add x y =
  let z = x + y in
  if (z lxor x) land (z lxor y) < 0 then raise Overflow;
  z

let sum_list l =
  try
    let s = List.fold_left safe_add 0 l in
    printf "Sum is %d\n" s
  with Overflow ->
    printf "Overflow!\n"
```

Exemples d'utilisations d'exceptions

Sortie anticipée d'une récursion :

```
let list_product l =
    let exception Zero in
    let rec product = function
      | [] -> 1
      | 0 :: _ -> raise Zero
      | n :: l -> n * product l
    in
    try product l with Zero -> 0
```

Exemples d'utilisations d'exceptions

Émuler `break` et `continue` :

```
exception Break in
exception Continue in
try
    for i = lo to hi do
        try
            ... raise Break ... raise Continue ...
        with Continue -> ()
    done
with Break -> ()
```

Exceptions levées et gérées dans la même fonction

$\approx$ *multi-level exit* (1er cours) $\approx$ `goto` en avant.

Sémantique à réductions sous contextes

Deux règles de réduction «en tête» pour `try...with` :

$$\texttt{try}\ v\ \texttt{with}\ x \to e \quad \xrightarrow{\varepsilon} \quad v$$

$$\texttt{try}\ D[\texttt{raise}\ v]\ \texttt{with}\ x \to e \quad \xrightarrow{\varepsilon} \quad e\{x \leftarrow v\}$$

Ici, D est un contexte ne contenant pas de `try...with` :

Contextes de réduction :

$$C ::= [\,] \mid C\ e \mid v\ C \mid \texttt{raise}\ C \mid \texttt{try}\ C\ \texttt{with}\ x \to e$$

Contextes de propagation d'exceptions :

$$D ::= [\,] \mid D\ e \mid v\ D \mid \texttt{raise}\ D$$

(Voir plus loin : la sémantique des gestionnaires d'effets là.)

Sémantique à réductions sous contextes

Soit p un programme sur le point de lever l'exception v :

$$p = C[\texttt{raise } v]$$

S'il y a un `try...with` qui englobe le `raise` v, on décompose

$$p = C'\,[\texttt{try } D[\texttt{raise } v] \texttt{ with } x \to e]$$

et on réduit

$$p \to C'\,[e\{x \leftarrow v\}]$$

S'il n'y a pas de `try...with` autour du `raise`, le programme p est ≪bloqué≫ sur une exception non rattrapée.

Le «style à retour d'exceptions»

Une alternative aux exceptions : représenter les erreurs dans les valeurs de retour des fonctions.

```
type ('a, 'e) result = V of 'a | E of 'e

let safe_add x y : (int, string) result =
  let z = x + y in
  if (z lxor x) land (z lxor y) < 0
  then E "overflow"
  else V z

let rec safe_add_list = function
  | [] -> V 0
  | x :: l ->
      match safe_add_list l with
      | V y -> safe_add x y
      | E e -> E e
```

La transformation ≪ERS≫ (*Exception-Returning Style*)

$$
\begin{aligned}
\mathcal{E}(cst) &= V\ cst \\
\mathcal{E}(x) &= V\ x \\
\mathcal{E}(\lambda x.\, e) &= V\ (\lambda x.\, \mathcal{E}(e)) \\
\mathcal{E}(e_1\ e_2) &= \texttt{match}\ \mathcal{E}(e_1)\ \texttt{with}\ E\ x_1 \to E\ x_1 \mid V\ v_1 \to \\
&\qquad \texttt{match}\ \mathcal{E}(e_2)\ \texttt{with}\ E\ x_2 \to E\ x_2 \mid V\ v_2 \to v_1\ v_2 \\
\mathcal{E}(\texttt{raise}\ e) &= \texttt{match}\ \mathcal{E}(e)\ \texttt{with}\ E\ x \to E\ x \mid V\ v \to E\ v \\
\mathcal{E}(\texttt{try}\ e_1\ \texttt{with}\ x \to e_2) & \\
&= \texttt{match}\ \mathcal{E}(e_1)\ \texttt{with}\ E\ x \to \mathcal{E}(e_2) \mid V\ v \to V\ v
\end{aligned}
$$

La transformation propage les résultats d'erreur ≪vers le haut≫, sauf pour `try...with`, qui récupère le résultat d'erreur.

Alternative : le CPS «à deux canons»

Deux continuations : `k1` pour renvoyer une valeur, `k2` pour lever une exception.

```
let safe_add x y k1 k2 =
  let z = x + y in
  if (z lxor x) land (z lxor y) < 0
  then k2 "overflow"
  else k1 z

let rec safe_add_list l k1 k2 =
  match l with
  | [] -> k1 0
  | x :: l ->
      safe_add_list l (fun v -> safe_add x v k1 k2) k2
```

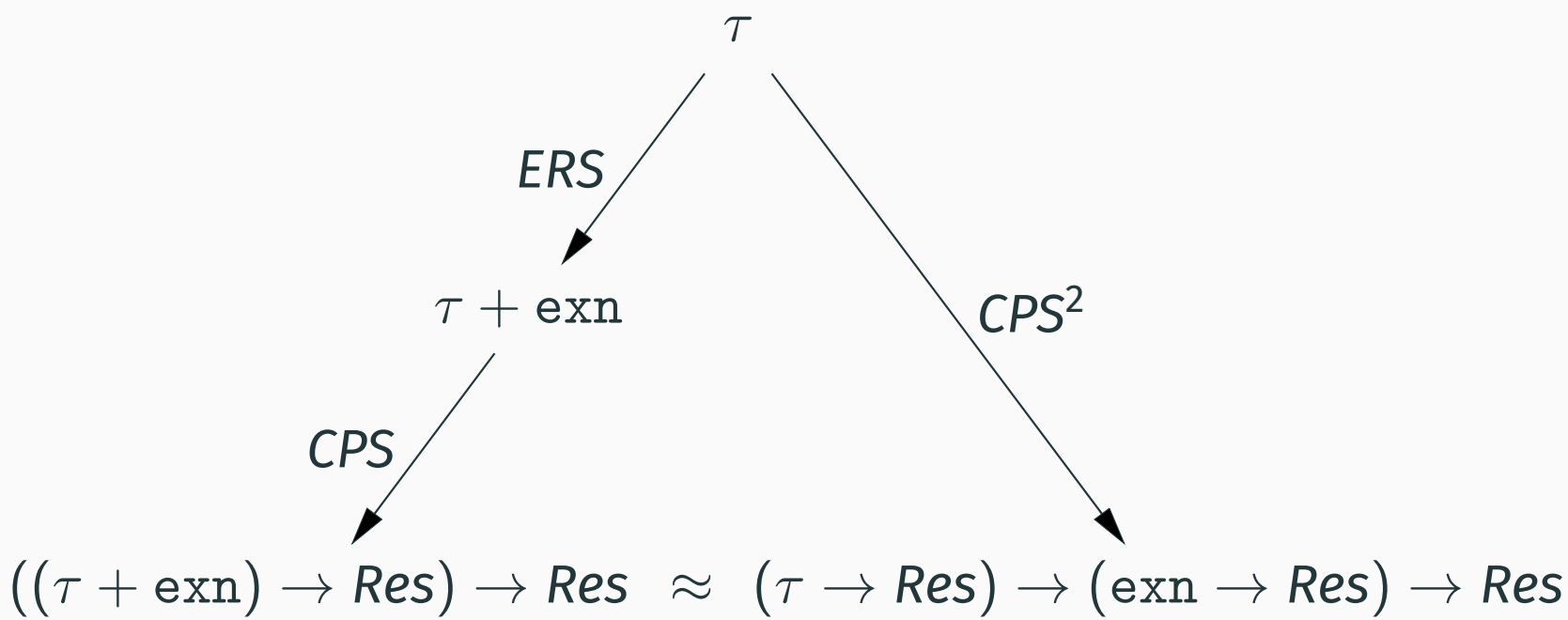
Une transformation CPS «à deux canons»

$$
\begin{aligned}
\mathcal{C}^2(cst) &= \lambda k_1.\, \lambda k_2.\, k_1\ cst \\
\mathcal{C}^2(x) &= \lambda k_1.\, \lambda k_2.\, k_1\ x \\
\mathcal{C}^2(\lambda x.\, e) &= \lambda k_1.\, \lambda k_2.\, k_1\ (\lambda x.\, \mathcal{C}^2(e)) \\
\mathcal{C}^2(e_1\ e_2) &= \lambda k_1.\, \lambda k_2.\, \mathcal{C}^2(e_1)\ (\lambda v_1.\, \mathcal{C}^2(e_2)\ (\lambda v_2.\, v_1\ v_2\ k_1\ k_2)\ k_2)\ k_2 \\
\mathcal{C}^2(\texttt{raise}\ e) &= \lambda k_1.\, \lambda k_2.\, \mathcal{C}^2(e)\ k_2\ k_2 \\
\mathcal{C}^2(\texttt{try}\ e_1\ \texttt{with}\ x \to e_2) & \\
&= \lambda k_1.\, \lambda k_2.\, \mathcal{C}^2(e_1)\ k_1\ (\lambda x.\, \mathcal{C}^2(e_2)\ k_1\ k_2)
\end{aligned}
$$

La transformation propage la continuation d'erreur k_2 «vers le bas» (vers les sous-expressions), sauf `try...with` qui installe une nouvelle continuation d'erreur.

Transformation CPS «à deux canons» ≈ transformation ERS puis transformation CPS

Pour un programme de type de base τ :



Même isomorphisme de types que
$(A + B) \to C \approx (A \to C) \times (B \to C)$.

Effets et gestionnaires d’effets

Effets et gestionnaires d'effets

Les effets algébriques : (Plotkin, Power, Pretnar, 2003, 2009)

Une théorie de la génération et de la spécification des effets (état mutable, E/S, exceptions, non-déterminisme, ...) dans les langages de programmation.

(→ 6e cours)

Les effets définis par l'utilisateur et leurs gestionnaires :

(Bauer & Pretnar, 2015)

Une puissante structure de contrôle inspirée par la théorie des effets algébriques.

Combine exceptions redémarrables et continuations délimitées.

Gérer les erreurs avec des exceptions

```
type exn += Conversion_failure of string

let int_of_string s =
  match int_of_string_opt s with
  | Some n -> n
  | None  -> raise (Conversion_failure s)

let sum_stringlist lst =
  lst |> List.map int_of_string |> List.fold_left (+) 0

let safe_sum_stringlist lst =
  match sum_stringlist lst with
  | res -> res
  | exception Conversion_failure s ->
      printf "Bad input: %s\n" s; max_int
```

Corriger les erreurs avec des effets

```
type _ eff += Conversion_failure : string -> int eff

let int_of_string s =
  match int_of_string_opt s with
  | Some n -> n
  | None   -> perform (Conversion_failure s)

let sum_stringlist lst =
  lst |> List.map int_of_string |> List.fold_left (+) 0

let safe_sum_stringlist lst =
  match sum_stringlist lst with
  | res -> res
  | effect Conversion_failure s, k ->
      printf "Bad input: %s, replaced with 0\n" s;
      continue k 0
```

Exemple d'exécution

Sans gestionnaire d'effet : comme une exception non rattrapée.

```
# let n = sum_stringlist ["1"; "xxx"; "2"; "yyy"]
Exception: Stdlib.Effect.Unhandled(Conversion_failure("xxx"))
```

Avec le gestionnaire : récupération et correction des erreurs.

```
# let n = safe_sum_stringlist ["1"; "xxx"; "2"; "yyy"]
Bad input xxx, replaced with 0
Bad input yyy, replaced with 0
val n : int = 3
```

(Exemples écrits et exécutés en OCaml 5.1.1 + syntaxe expérimentale `match with effect`. Pour l'utiliser : `opam switch create 5.1.1+effect-syntax` .)

Effets et continuations

```
let int_of_string s = ... perform (Conversion_failure s)

let safe_sum_stringlist lst =
  match ...
  with effect Conversion_failure s, k -> ... continue k 0
```

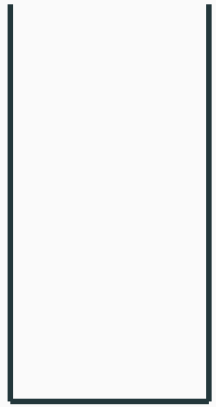
Lorsque `perform` lève un effet, sa continuation (délimitée) est capturée et transmise au gestionnaire en même temps que l'effet.

Le gestionnaire de l'effet peut soit ignorer cette continuation `k`, soit la relancer avec une valeur du type attendu par le contexte (ici, `int`).

Limitation (en OCaml, pas dans d'autres langages) :
la continuation est ≪linéaire≫ et ne peut être relancée plusieurs fois.

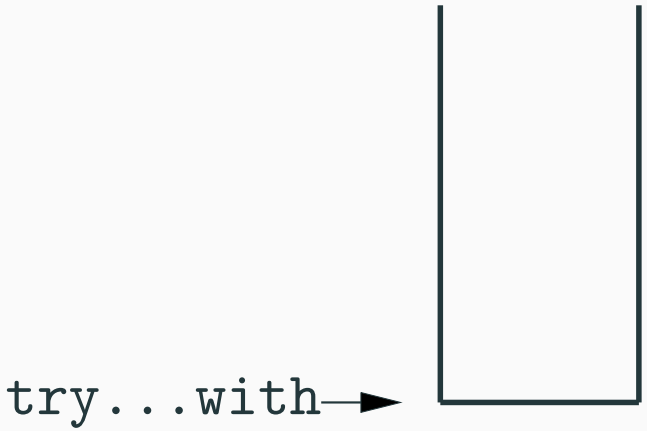
Intuitions en termes de piles d'appel

Exceptions = couper la pile.



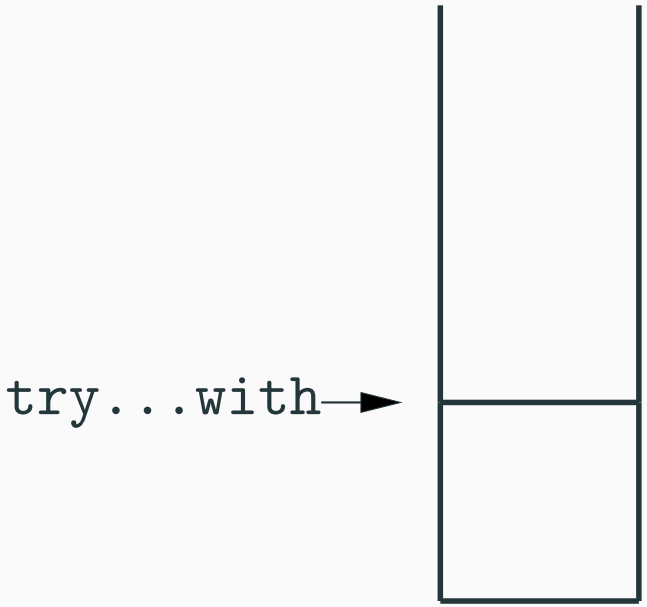
Intuitions en termes de piles d'appel

Exceptions = couper la pile.



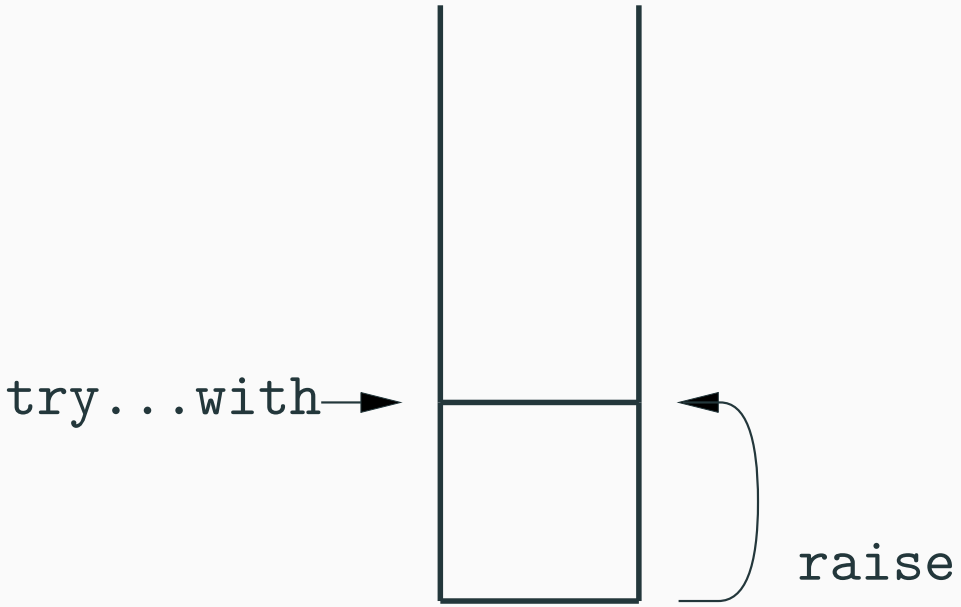
Intuitions en termes de piles d'appel

Exceptions = couper la pile.



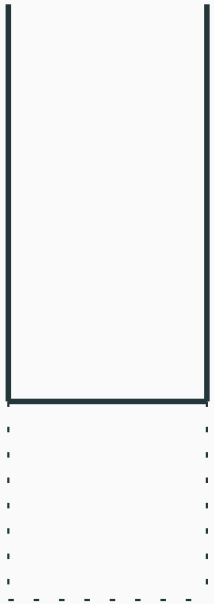
Intuitions en termes de piles d'appel

Exceptions = couper la pile.



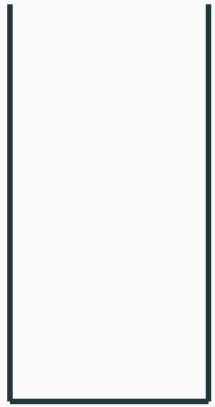
Intuitions en termes de piles d'appel

Exceptions = couper la pile.



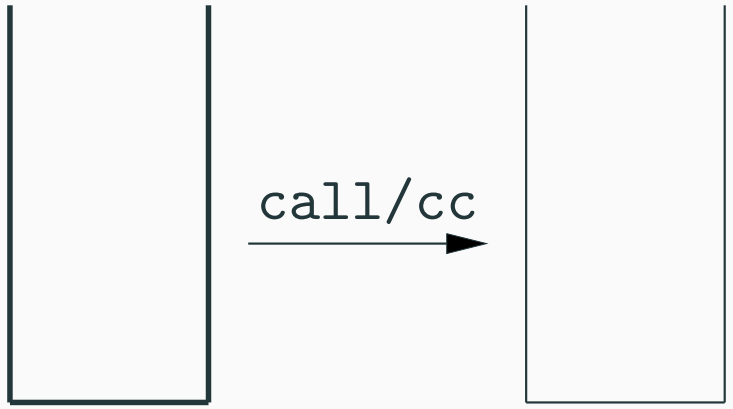
Intuitions en termes de piles d'appel

Continuations non délimitées naïves = copier la pile (vers le tas).



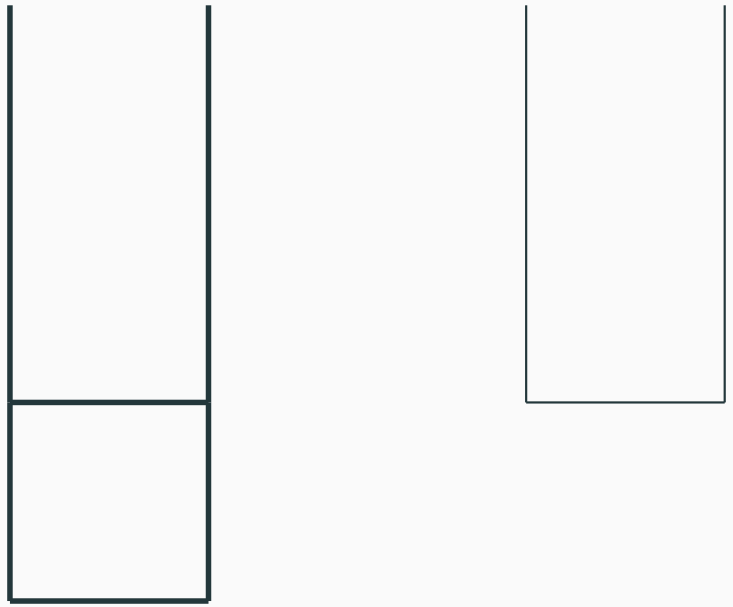
Intuitions en termes de piles d'appel

Continuations non délimitées naïves = copier la pile (vers le tas).



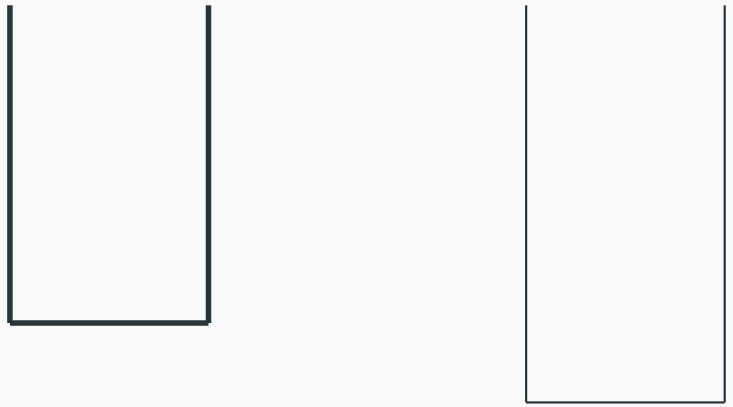
Intuitions en termes de piles d'appel

Continuations non délimitées naïves = copier la pile (vers le tas).



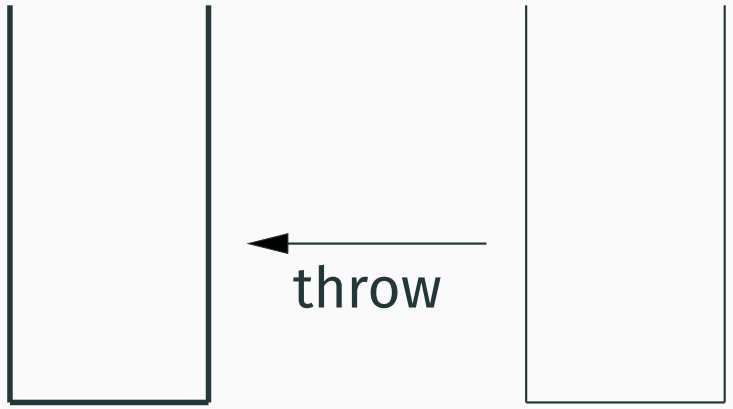
Intuitions en termes de piles d'appel

Continuations non délimitées naïves = copier la pile (vers le tas).



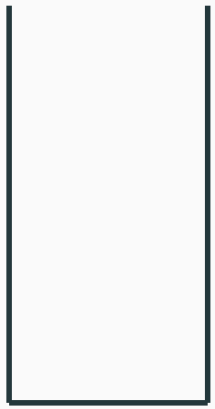
Intuitions en termes de piles d'appel

Continuations non délimitées naïves = copier la pile (vers le tas).



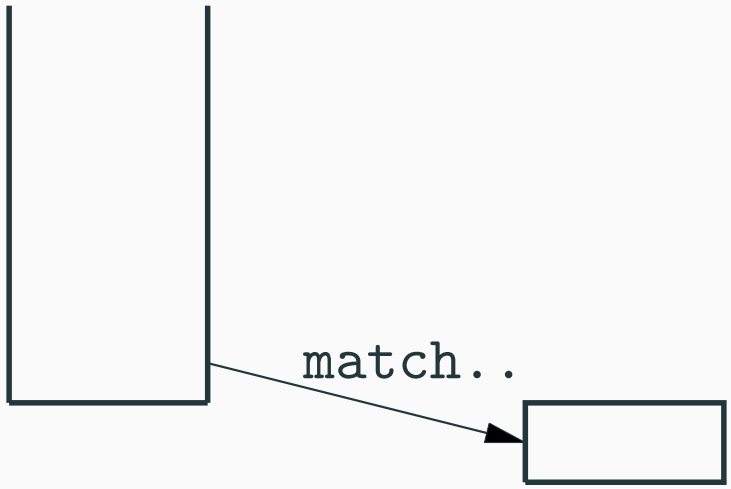
Intuitions en termes de piles d'appel

Gestionnaires d'effets = commuter entre plusieurs piles.



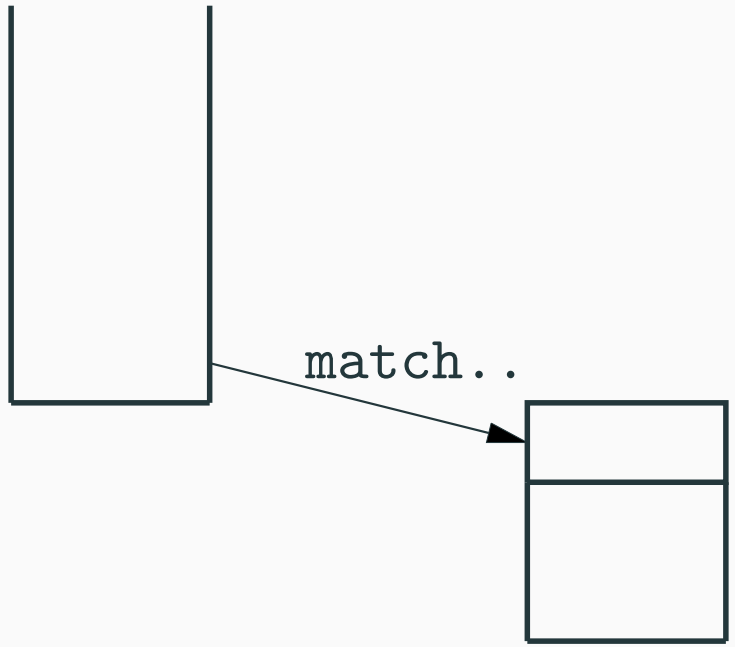
Intuitions en termes de piles d'appel

Gestionnaires d'effets = commuter entre plusieurs piles.



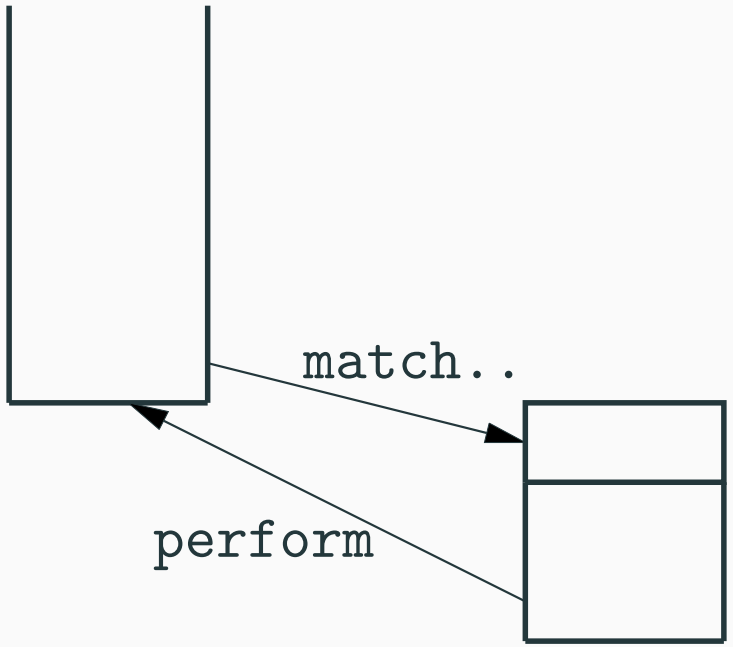
Intuitions en termes de piles d'appel

Gestionnaires d'effets = commuter entre plusieurs piles.



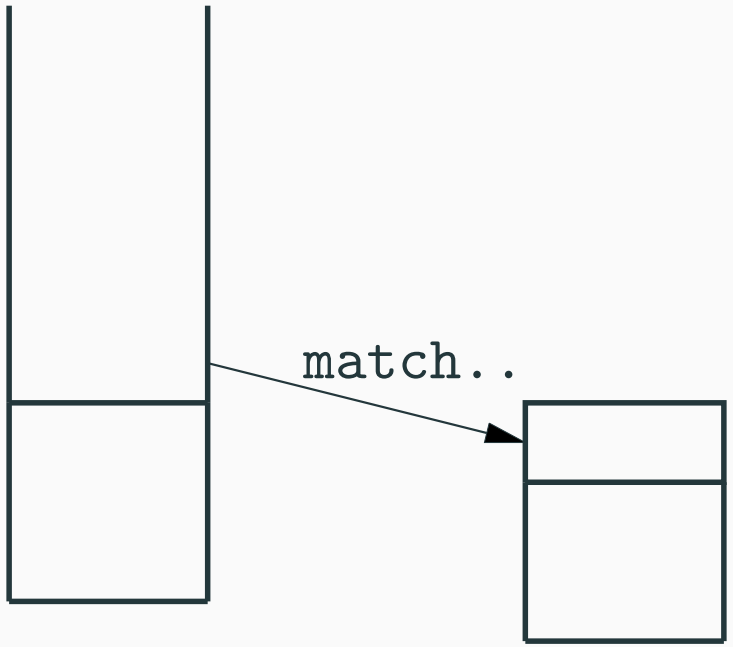
Intuitions en termes de piles d'appel

Gestionnaires d'effets = commuter entre plusieurs piles.



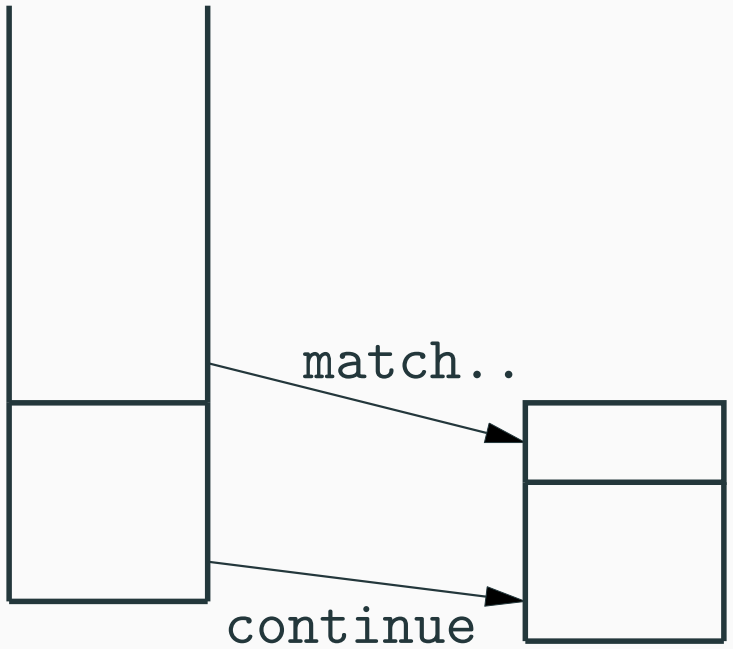
Intuitions en termes de piles d'appel

Gestionnaires d'effets = commuter entre plusieurs piles.



Intuitions en termes de piles d'appel

Gestionnaires d'effets = commuter entre plusieurs piles.



En OCaml : pas de copie de piles $\rightarrow$ continuations linéaires.

Gestionnaires «profonds», gestionnaires «superficiels»

Gestionnaire profond : (*deep handler*)

reste actif lorsqu'on relance une continuation ;

se désactive seulement lorsque le calcul termine normalement.

```
# let n = safe_sum_stringlist ["1"; "xxx"; "2"; "yyy"]
Bad input xxx, replaced with 0
Bad input yyy, replaced with 0
val n : int = 3
```

Gestionnaire superficiel : (*shallow handler*)

se désactive dès le premier effet géré.

```
# let n = safe_sum_stringlist ["1"; "xxx"; "2"; "yyy"]
Bad input xxx, replaced with 0
Exception: Stdlib.Effect.Unhandled(Conversion_failure("yyy"))
```

(En OCaml : `match...with` est «profond» ; la bibliothèque `Effect.Shallow` implémente la sémantique «superficielle».)

Inversion de contrôle sur un itérateur

Comme au 4e cours, on suppose donné un itérateur ≪interne≫ comme p.ex. celui sur les arbres binaires :

```
type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree
let rec tree_iter (f: 'a -> unit) (t: 'a tree) =
  match t with
  | Leaf -> ()
  | Node(l, x, r) -> tree_iter f l; f x; tree_iter f r
```

On souhaite en dériver un itérateur ≪externe≫ :

```
type 'a enum = Done | More of 'a * (unit -> 'a enum)
val tree_enum : 'a tree -> 'a enum
```

Inversion de contrôle sur un itérateur

```
let tree_enum (type elt) : elt tree -> elt enum =
  let module Inv = struct
    type _ eff += Next : elt -> unit eff
    let tree_enum (t: elt tree) : elt enum =
      match tree_iter (fun x -> perform (Next x)) t with
      | () -> Done
      | effect Next x, k -> More(x, fun () -> continue k ())
  end in
  Inv.tree_enum
```

On utilise un module local d'OCaml afin de déclarer un effet `Next` qui est local à la fonction, et qui a ≪le bon type≫ pour rendre la fonction `tree_enum` polymorphe.

Inversion de contrôle sur un itérateur

```
let tree_enum (type elt) : elt tree -> elt enum =
  let module Inv = struct
    type _ eff += Next : elt -> unit eff
    let tree_enum (t: elt tree) : elt enum =
      match tree_iter (fun x -> perform (Next x)) t with
      | () -> Done
      | effect Next x, k -> More(x, fun () -> continue k ())
  end in
  Inv.tree_enum
```

Pour chaque élément `x` de l'arbre, l'effet `Next x` est levé.
Le gestionnaire reçoit `x` et la continuation `k` qui permet de reprendre le parcours.

Inversion de contrôle sur un itérateur

```
let tree_enum (type elt) : elt tree -> elt enum =
  let module Inv = struct
    type _ eff += Next : elt -> unit eff
    let tree_enum (t: elt tree) : elt enum =
      match tree_iter (fun x -> perform (Next x)) t with
      | () -> Done
      | effect Next x, k -> More(x, fun () -> continue k ())
  end in
  Inv.tree_enum
```

Quand le parcours est terminé, `tree_iter` renvoie `()`, que le gestionnaire transforme en `Done`.

Inversion de contrôle sur un itérateur

```
let tree_enum (type elt) : elt tree -> elt enum =
  let module Inv = struct
    type _ eff += Next : elt -> unit eff
    let tree_enum (t: elt tree) : elt enum =
      match tree_iter (fun x -> perform (Next x)) t with
      | () -> Done
      | effect Next x, k -> More(x, fun () -> continue k ())
  end in
  Inv.tree_enum
```

Le gestionnaire change le type du calcul :

`tree_iter ... t` a le type `unit`, mais

`match tree_iter ...` a le type `elt enum`.

Comparaison `callcc` / gestionnaire d'effets

Avec `callcc` : (4e cours)

```
callcc (fun k ->
  tree_iter
    (fun x ->
      callcc
        (fun k' ->
          k (More(x, k'))))
    t;
    Done)
```

Deux `callcc` : un pour sortir, un pour préparer la reprise.

Le calcul `More(x, ...)` se fait dans la fonction itérée.

Avec un gestionnaire d'effets :

```
match
  tree_iter
    (fun x -> perform (Next x))
    t
with
| () -> Done
| effect Next x, k ->
    More(x, fun () -> resume k ())
```

Un seul `perform` pour sortir tout en capturant la continuation de reprise.

Le calcul `More(x, ...)` se fait dans le gestionnaire.

Inversion de contrôle sur un itérateur

On peut généraliser cette construction afin d'inverser n'importe quel itérateur interne sur n'importe quel type de données :

```
let enum_of_iter
      (type elt) (type collection)
      (iter: (elt -> unit) -> collection -> unit)
      : collection -> elt enum =
  let module Inv = struct
    type _ eff += Next : elt -> unit eff
    let enum coll =
      match iter (fun x -> perform (Next x)) coll with
      | () -> Done
      | effect Next x, k -> More(x, fun () -> continue k ())
  end in Inv.enum
```

Transformer, réémettre des effets

(M. Pretnar, *An introduction to algebraic effects and handlers*, 2015.)

Un effet de sortie `Print` :

```
type _ eff += Print : string -> unit eff

let print s = perform (Print s)

let abc () = print "a"; print "b"; print "c"
```

Transformer, réémettre des effets

L'effet peut être interprété comme une ≪vraie≫ sortie :

```
let output f =
  match f () with
  | () -> print_newline()
  | effect Print s, k -> print_string s; continue k ()
```

Mais on peut aussi collecter les sorties en une chaîne :

```
let collect f =
  match f () with
  | () -> ""
  | effect Print s, k -> s ^ continue k ()
```

`collect abc` produit la chaîne de caractères `"abc"`.

Transformer, réémettre des effets

On peut aussi réémettre l'effet `Print` après l'avoir transformé, p.ex. pour inverser l'ordre des sorties :

```
let reverse f =
  match f () with
  | () -> ()
  | effect Print s, k -> continue k (); print s
```

ou pour leur ajouter un numéro d'ordre :

```
let number f =
  begin match f () with
  | () -> (fun lineno -> ())
  | effect Print s, k ->
      (fun lineno ->
         print (sprintf "%d:%s\n" lineno s);
         continue k () (lineno + 1))
  end 1
```

Implémenter les *threads* coopératifs avec des gestionnaires d'effets

Une bibliothèque de *threads* coopératifs

L'interface naturelle en ≪style direct≫ :

`spawn: (unit -> unit) -> unit`

Démarre un nouveau *thread*.

`yield: unit -> unit`

Suspend le *thread* courant;
passe la main à un autre *thread*.

`terminate: unit -> unit`

Arrête définitivement le *thread* courant.

Définition des effets correspondants

Les opérations sont définies trivialement comme levant des effets (qui seront gérés par l'ordonnanceur).

```
type _ eff +=
  | Spawn : (unit -> unit) -> unit eff
  | Yield : unit eff
  | Terminate : unit eff

let spawn f = perform (Spawn f)
let yield () = perform Yield
let terminate () = perform Terminate
```

L'état de l'ordonnanceur

Une file d'attente de *threads* suspendus après un appel à `yield`, prêts à être relancés.

```
let runnable : (unit -> unit) Queue.t = Queue.create()

let suspend f = Queue.add f runnable

let restart () =
  match Queue.take_opt runnable with
  | None -> ()
  | Some f -> f ()
```

L'ordonnanceur

```
let rec run (f: unit -> unit) =
  match f() with
  | () -> restart ()
  | effect Terminate, k -> restart ()
  | effect Yield, k -> suspend (continue k); restart ()
  | effect Spawn f, k -> suspend (continue k); run f
```

L'ordonnanceur

```
let rec run (f: unit -> unit) =
  match f() with
  | () -> restart ()
  | effect Terminate, k -> restart ()
  | effect Yield, k -> suspend (continue k); restart ()
  | effect Spawn f, k -> suspend (continue k); run f
```

Le *thread* courant termine normalement :

on relance un autre *thread*.

L'ordonnanceur

```
let rec run (f: unit -> unit) =
  match f() with
  | () -> restart ()
  | effect Terminate, k -> restart ()
  | effect Yield, k -> suspend (continue k); restart ()
  | effect Spawn f, k -> suspend (continue k); run f
```

Le *thread* courant a appelé `terminate` :
on ignore la continuation `k` (le thread ne redémarrera jamais),
et on relance un autre *thread*.

L'ordonnanceur

```
let rec run (f: unit -> unit) =
  match f() with
  | () -> restart ()
  | effect Terminate, k -> restart ()
  | effect Yield, k -> suspend (continue k); restart ()
  | effect Spawn f, k -> suspend (continue k); run f
```

Le *thread* courant a appelé `yield` :
on stocke la continuation `k` comme prête à redémarrer,
et on relance un autre *thread*.

L'ordonnanceur

```
let rec run (f: unit -> unit) =
  match f() with
  | () -> restart ()
  | effect Terminate, k -> restart ()
  | effect Yield, k -> suspend (continue k); restart ()
  | effect Spawn f, k -> suspend (continue k); run f
```

Le *thread* courant a appelé `spawn f` :
on stocke la continuation `k` comme prête à redémarrer,
et on lance l'exécution de `f`.

L'ordonnanceur

```
let rec run (f: unit -> unit) =
  match f() with
  | () -> restart ()
  | effect Terminate, k -> restart ()
  | effect Yield, k -> suspend (continue k); restart ()
  | effect Spawn f, k -> suspend (continue k); run f
```

Alternative :

```
  | effect Spawn f, k ->
      suspend (fun () -> run f); continue k ()
```

Dans les deux cas il faut faire `run f`, et non juste `f()`,
pour que les effets de `f()` soient gérés.

Un exemple d'utilisation

Un code «client», écrit en style direct :

```
let task name n =
  for i = 1 to n do printf "%s%d " name i; yield() done

let _ =
  run (fun () ->
    spawn (fun () -> task "a" 6);
    spawn (fun () -> task "b" 3);
    task "c" 4)
```

Affichage : `a1 b1 a2 c1 b2 a3 c2 b3 a4 c3 a5 c4 a6`

Ajout de communications par passage de messages

`new_channel : unit -> 'a channel`

Crée un nouveau canal pour passer des messages de type `'a`.

`recv : 'a channel -> 'a`

Reçoit un message depuis un canal.

`send : 'a channel -> 'a -> unit`

Envoie un message sur un canal.

On choisit une sémantique de «rendez-vous» (π-calcul) :
`send ch v` bloque jusqu'à ce qu'un autre *thread* fasse `recv ch` ;
les deux *threads* redémarrent ;
`recv ch` renvoie la valeur `v`.

La structure d'un canal de communication

Un canal = deux files d'attentes,
une pour les tâches bloquées sur `send` en attente d'un `recv`,
l'autre pour les tâches bloquées sur `recv` en attente d'un `send`.

```
type 'a channel = {
    senders: ('a * (unit, unit) continuation) Queue.t;
    receivers: ('a, unit) continuation Queue.t
  }

let new_channel () =
  { senders = Queue.create(); receivers = Queue.create() }
```

À tout instant, au moins une des files est vide.

Les opérations sur les canaux

Comme d'habitude, on transforme en effets les opérations qu'on ne sait pas implémenter localement.

```
type _ eff +=
  | Send : 'a channel * 'a -> unit eff
  | Recv : 'a channel -> 'a eff

let send ch v = perform (Send(ch, v))
let recv ch = perform (Recv ch)
```

L'ordonnanceur étendu avec les canaux

```
let rec run (f: unit -> unit) =
  match f () with
    ...
  | effect Send(ch, v), k ->
      begin match Queue.take_opt ch.receivers with
      | Some rc -> suspend (continue k); continue rc v
      | None    -> Queue.add (v, k) ch.senders; restart()
      end
  | effect Recv ch, k ->
      begin match Queue.take_opt ch.senders with
      | Some(v, sn) -> suspend (continue sn); continue k v
      | None        -> Queue.add k ch.receivers; restart()
      end
```

Sémantique des gestionnaires d’effets

Un mini-langage fonctionnel avec effets

Expressions :

$$
\begin{array}{rll}
e ::= & cst \mid x \mid \lambda x.\, e \mid e_1\ e_2 & \\
& \mid \texttt{perform}\ e & \text{effectuer l'effet } e \\
& \mid \texttt{handle}\ e\ \texttt{with}\ e_{ret}, e_{eff} & \text{gérer les effets dans } e
\end{array}
$$

`perform` e interrompt l'exécution et se branche au `handle` englobant le plus proche.

Un mini-langage fonctionnel avec effets

Expressions :

$$
\begin{aligned}
e ::= {} & cst \mid x \mid \lambda x.\, e \mid e_1\ e_2 \\
& \mid \texttt{perform}\ e && \text{effectuer l'effet } e \\
& \mid \texttt{handle}\ e\ \texttt{with}\ e_{ret}, e_{eff} && \text{gérer les effets dans } e
\end{aligned}
$$

$\texttt{handle}\ e\ \texttt{with}\ e_{ret}, e_{eff}$ évalue le corps e.

Si e s'évalue en v sans lever d'effets, on applique e_{ret} à v.

Si e lève un effet f, on applique e_{eff} à (f, k)
où f est la valeur de l'effet et k la continuation du `perform`.

Codage du `match...with effect`

En ajoutant des types algébriques extensibles et du filtrage, on peut coder

$$
\begin{array}{l}
\texttt{match}\ e\ \texttt{with} \\
\mid x \to e_0 \\
\mid \texttt{effect}\ F_1\, x_1, k \to e_1 \\
\vdots \\
\mid \texttt{effect}\ F_n\, x_n, k \to e_n
\end{array}
$$

par

$$
\begin{array}{l}
\texttt{handle}\ e\ \texttt{with} \\
\quad (\lambda x.\, e_0), \\
\quad (\lambda (f, k).\ \texttt{match}\ f\ \texttt{with} \\
\qquad \mid F_1\, x_1 \to e_1 \mid \ldots \mid F_n\, x_n \to e_n \\
\qquad \mid \_ \to k\, (\texttt{perform}\ \ f))
\end{array}
$$

Sémantique à réductions sous contexte

(Très proche de la sémantique des exceptions ici.)

Deux règles de réduction ≪en tête≫ pour `handle` :

$$\texttt{handle}\ v\ \texttt{with}\ e_1, e_2 \xrightarrow{\varepsilon} e_1\, v$$

$$\texttt{handle}\ D[\texttt{perform}\ v]\ \texttt{with}\ e_1, e_2 \xrightarrow{\varepsilon} e_2\, (v, (\lambda v'.D[v']))$$

Ici, D est un contexte ne contenant pas de `handle` :

Contextes de réduction :

$$C ::= [\,] \mid C\, e \mid v\, C \mid \texttt{perform}\ C \mid \texttt{handle}\ C\ \texttt{with}\ e_1, e_2$$

Contextes de propagation d'effets :

$$D ::= [\,] \mid D\, e \mid v\, D \mid \texttt{perform}\ D$$

Gestionnaires profonds, gestionnaires superficiels

$$\texttt{handle}\ D[\texttt{perform}\ v]\ \texttt{with}\ e_1, e_2 \xrightarrow{\varepsilon} e_2\ (v, \lambda v'.\, D[v'])$$

La règle ci-dessus donne une sémantique ≪superficielle≫ au gestionnaire : il ne s'applique plus lorsque la continuation D est relancée.

La sémantique ≪profonde≫ s'obtient en réinstallant le gestionnaire autour de la continuation D :

$$\texttt{handle}\ D[\texttt{perform}\ v]\ \texttt{with}\ e_1, e_2 \xrightarrow{\varepsilon} e_2\ (v,\ \lambda v'.\, \texttt{handle}\ D[v']\ \texttt{with}\ e_1, e_2)$$

Transformation CPS pour les continuations délimitées

(M. Materzok, D. Biernacki, *Subtyping delimited continuations*, 2011.)

Pour des continuations non délimitées (`callcc`), la transformée CPS prend en argument une continuation k, et assure que

$$\mathcal{C}(e)\ k \xrightarrow{*} k\ cst \quad \text{si} \quad e \xrightarrow{*} cst$$

Pour des continuations délimitées, la transformée CPS prend en arguments $n+1$ continuations $k_0, \ldots, k_n$,
où n est le nombre de délimiteurs englobants,
et chaque k_i est la continuation jusqu'au prochain délimiteur.

$$\mathcal{C}(e)\ k_0\ k_1\ \ldots\ k_n \xrightarrow{*} k_0\ cst\ k_1\ \ldots\ k_n \quad \text{si} \quad e \xrightarrow{*} cst$$

Transformation CPS pour le noyau pur du langage

$$
\begin{aligned}
\mathcal{C}(cst) &= \lambda k.\ k\ cst \\
\mathcal{C}(x) &= \lambda k.\ k\ x \\
\mathcal{C}(\lambda x.\, e) &= \lambda k.\ k\ (\lambda x.\, \mathcal{C}(e)) \\
\mathcal{C}(e_1\ e_2) &= \lambda k.\, \mathcal{C}(e_1)\ (\lambda v_1.\, \mathcal{C}(e_2)\ (\lambda v_2.\, v_1\ v_2\ k))
\end{aligned}
$$

Mêmes définitions que pour la traduction CPS en appel par valeur habituelle. Elles restent correctes lorsque $\mathcal{C}(e)$ est appliqué à plusieurs continuations, p.ex.

$$\mathcal{C}(cst)\ k_0\ k_1\ \ldots\ k_n = (\lambda k.\ k\ cst)\ k_0\ k_1\ \ldots\ k_n \to k_0\ cst\ k_1\ \ldots\ k_n$$

Transformation CPS pour les continuations délimitées

On formalise ici les opérateurs $\texttt{shift}_0$ et $\texttt{reset}_0$
(O. Danvy et A. Filinksi, 1989).

Un délimiteur ajoute une continuation triviale en tête de liste :

$$\mathcal{C}(\texttt{delim}\ \ e) = \mathcal{C}(e)\ (\lambda x.\lambda k.\ k\ x)$$

de sorte que, dans le cas $e \xrightarrow{*} cst$,

$$\begin{aligned} \mathcal{C}(\texttt{delim}\ \ e)\ k_0\ k_1\ \ldots\ k_n &= \mathcal{C}(e)\ (\lambda x.\lambda k.\ k\ x)\ k_0\ \ldots\ k_n \\ &\xrightarrow{*} (\lambda x.\lambda k.\ k\ x)\ cst\ k_0\ \ldots\ k_n \\ &\to k_0\ cst\ k_1\ \ldots\ k_n \end{aligned}$$

Transformation CPS pour les continuations délimitées

Symétriquement, l'opérateur de capture réifie la première continuation en une valeur et l'enlève de la liste :

$$\mathcal{C}(\texttt{capture}\ \ (\lambda k.e)) = \lambda k.\, \mathcal{C}(e)$$

de sorte que

$$\mathcal{C}(\texttt{capture}\ \ (\lambda k.\, e))\ k_0\ k_1\ \ldots\ k_n = \mathcal{C}(e)[k \leftarrow k_0]\ k_1\ \ldots\ k_n$$

L'exécution de e va continuer avec k_1, la continuation «après» le délimiteur le plus proche.

La continuation jusqu'à ce délimiteur, k_0, a été capturée sous forme du paramètre k de e.

Transformation CPS pour les gestionnaires d'effets

(D. Hillerström, S. Lindley, R. Atkey, *Effect handlers via generalised continuations*, 2020.)

L'approche précédente + l'approche ≪à deux canons≫ :
la transformée CPS prend $2n + 2$ continuations en arguments,
avec n = nombre de gestionnaires englobants.

$$\mathcal{C}(e)\ k_0\ h_0\ k_1\ h_1\ \ldots\ k_n\ h_n$$

Les continuations délimitées $k_0, \ldots k_n$ sont appelées pour renvoyer des valeurs résultat.

Les continuations délimitées $h_0, \ldots h_n$ sont appelées pour lever des effets.

Transformation CPS pour la levée d'un effet

Pour le noyau pur du langage : on applique la transformation CPS en appel par valeur usuelle.

Pour la levée d'un effet :

$$\mathcal{C}(\texttt{perform}\ e) = \mathcal{C}(e)\ (\lambda f.\ \lambda k.\ \lambda h.\ h\ (f, \lambda x.\ k\ x\ h))$$

e est évaluée en une valeur d'effet f.

On capture la continuation normale k, ainsi que la continuation d'effets h, et on lance h en lui donnant f comme valeur d'effet et $k' = \lambda x.\ k\ x\ h$ comme moyen de reprendre après `perform`.

(L'application de k à h implémente la sémantique ≪profonde≫.)

Transformation CPS pour les gestionnaires d'effets

Un gestionnaire d'effets ajoute une continuation normale et une continuation d'effets :

$$\mathcal{C}(\texttt{handle } e \texttt{ with } e_1, e_2) = \mathcal{C}(e)\ (\lambda v.\lambda h.\mathcal{C}(e_1)\ v)\ \mathcal{C}(e_2)$$

Dans le cas où $e \xrightarrow{*} cst$,

$$\begin{aligned}
&\mathcal{C}(\texttt{handle } e \texttt{ with } e_1, e_2)\ k_0\ h_0\ \ldots k_n\ h_n \\
&\quad = \quad \mathcal{C}(e)\ (\lambda v.\lambda h.\mathcal{C}(e_1)\ v)\ \mathcal{C}(e_2)\ k_0\ h_0\ \ldots k_n\ h_n \\
&\quad \xrightarrow{*} \quad (\lambda v.\lambda h.\mathcal{C}(e_1)\ v)\ cst\ \mathcal{C}(e_2)\ k_0\ h_0\ \ldots k_n\ h_n \\
&\quad \xrightarrow{*} \quad \mathcal{C}(e_1)\ cst\ k_0\ h_0\ \ldots k_n\ h_n
\end{aligned}$$

Dans le cas où e lève un effet f avec la continuation k_f, la continuation $C(e_2)$ est appliquée à (f, k_f) et à la pile $k_0\ h_0\ \ldots$

Point d’étape

Point d'étape

Les gestionnaires d'effets :

- Un opérateur de contrôle permettant de manipuler (en style direct) les continuations délimitées.
- Une présentation sous forme d'exceptions redémarrables, plus familière que les opérateurs de contrôle antérieurs.
- Un nouveau style de programmation :
  des codes utilisateur qui lancent des effets pour les services dont ils ont besoin, les services étant réalisés par un gestionnaire appelant.

Bibliographie

Bibliographie

La version d'OCaml utilisée pour les exemples :

`opam update && opam switch create 5.1.1+effect-syntax`

Une introduction générale aux gestionnaires d'effets :

- Matija Pretnar : *An Introduction to Algebraic Effects and Handlers*, ENTCS 319, 2015. https://doi.org/10.1016/j.entcs.2015.12.003

Les transformations CPS pour les effets :

- Daniel Hillerström, Sam Lindley, Robert Atkey : *Effect Handlers via Generalised Continuations*, J. Funct. Program. 30, 2020. https://doi.org/10.1017/S0956796820000040

L'implémentation des effets dans OCaml version 5 :

- KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, Anil Madhavapeddy : *Retrofitting Effect Handlers onto OCaml*, PLDI, 2021. https://arxiv.org/abs/2104.00250