Effets et gestionnaires d'effets en OCaml 5
Première partie: La pratique

Xavier Leroy

Journées Francophones des Langages Applicatifs, 2023-02-01

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

OCaml 5

Deux nouveautés majeures :

- Le parallélisme à mémoire partagée.
  Domaines ($\approx$ *threads* lourds), GC parallèle, modèle mémoire faiblement cohérent, …
- Les effets définis par l'utilisateur et les gestionnaires d'effets.
  Une nouvelle structure de contrôle.
  Motivation : implémenter des *threads* légers en OCaml.

Quatre points de vue sur les effets

Technique : Continuations délimitées linéaires

Intuitif : Exceptions que l'on peut redémarrer

Pratique : Générateurs, coroutines, *threads* légers

Théorique : La suite des monades dans une théorie des effets

Ce cours = intuition + pratique. Le cours de demain = théorie.

AVERTISSEMENT

LES TRANSPARENTS QUI SUIVENT UTILISENT UNE SYNTAXE CONCRÈTE POUR LES EFFETS ET LEURS GESTIONNAIRES QUI N'EST PAS IMPLÉMENTÉE EN OCAML 5.0.

CETTE SYNTAXE PEUT CHOQUER LES UTILISATEURS SENSIBLES CAR ELLE N'A JAMAIS ÉTÉ APPROUVÉE PAR LE CORE DEV TEAM OCAML.

LES EXEMPLES S'ÉCRIVENT EN OCAML 5.0 SANS SYNTAXE SPÉCIALE, À L'AIDE DE FONCTIONS DE BIBLIOTHÈQUE. LE ≪VRAI≫ CODE OCAML 5.0 EST DISPONIBLE AVEC LE SUPPORT DU COURS ICI :

`https://xavierleroy.org/courses/JFLA-2023`

Exceptions redémarrables

Gérer les erreurs avec des exceptions

```
exception 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 =
  try
    sum_stringlist lst
  with Conversion_failure s ->
    printf "Bad input: %s\n" s; max_int
```

Corriger les erreurs avec des effets

```
effect Conversion_failure : string -> int

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 =
  try
    sum_stringlist lst
  with effect (Conversion_failure s) k ->
    printf "Bad input: %s, replaced with 0\n" s;
    continue k 0
```

Tada!

```
# 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
```

Effets et continuations

```
effect Conversion_failure : string -> int
let int_of_string s = ... perform (Conversion_failure s)
let safe_sum_stringlist lst =
  try ...
  with effect (Conversion_failure s) k -> ... continue k 0
```

Lorsque `perform` lève un effet, le contexte d'appel (la continuation) est capturé.

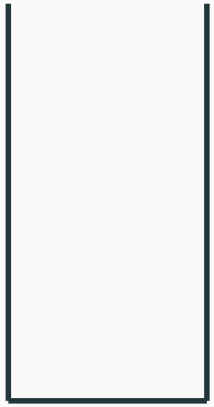
Le gestionnaire de l'effet a accès à cette continuation `k` et peut soit l'ignorer soit la relancer avec une valeur du type attendu par le contexte (ici : `int`).

Limitation : on ne relance pas plusieurs fois !

✘ `continue k 0 + continue k 1`

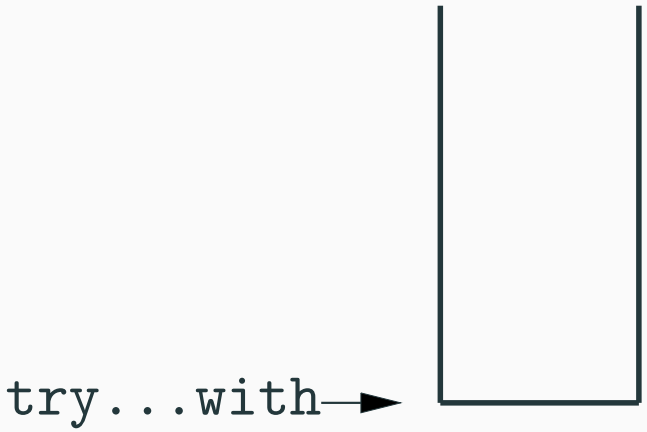
Intuitions en termes de piles d'appel

Exceptions = couper la pile.



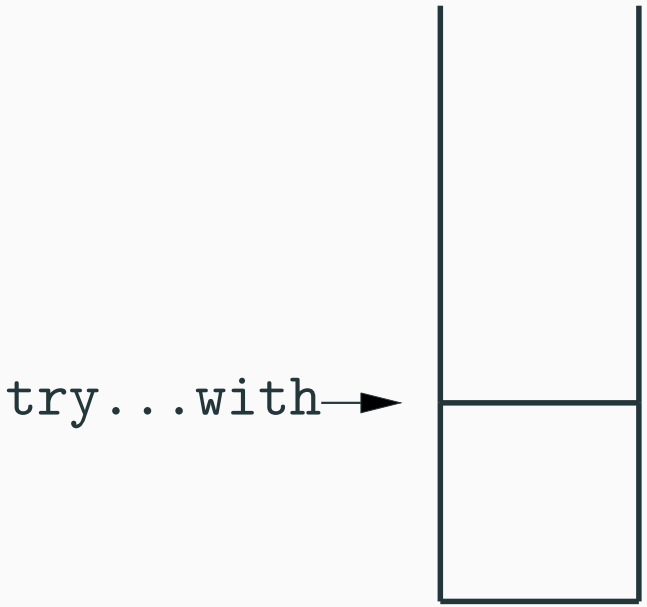
Intuitions en termes de piles d'appel

Exceptions = couper la pile.



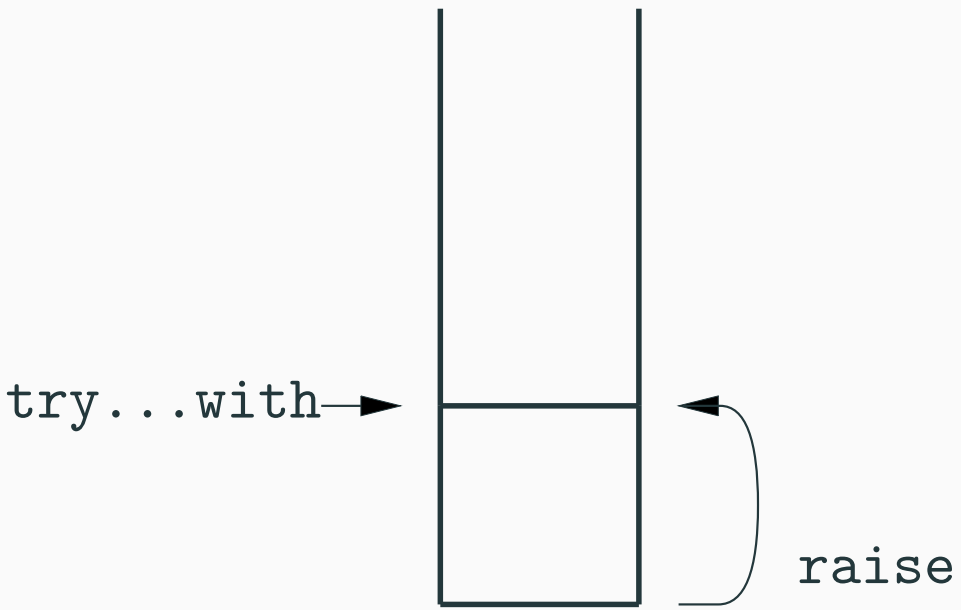
Intuitions en termes de piles d'appel

Exceptions = couper la pile.



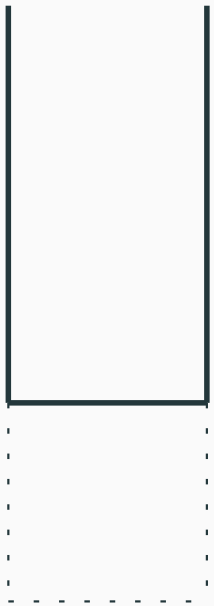
Intuitions en termes de piles d'appel

Exceptions = couper la pile.



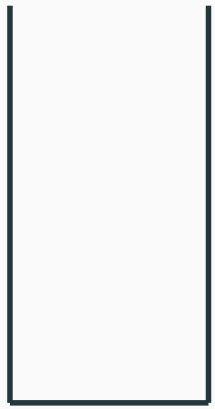
Intuitions en termes de piles d'appel

Exceptions = couper la pile.



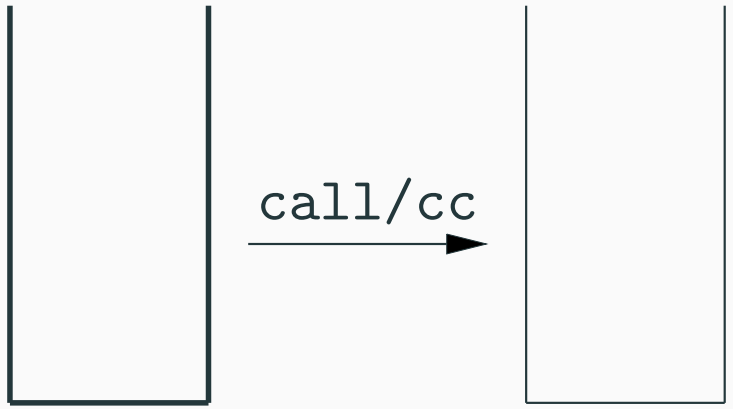
Intuitions en termes de piles d'appel

Continuations (naïves) = copier la pile (en partie) (vers le tas).



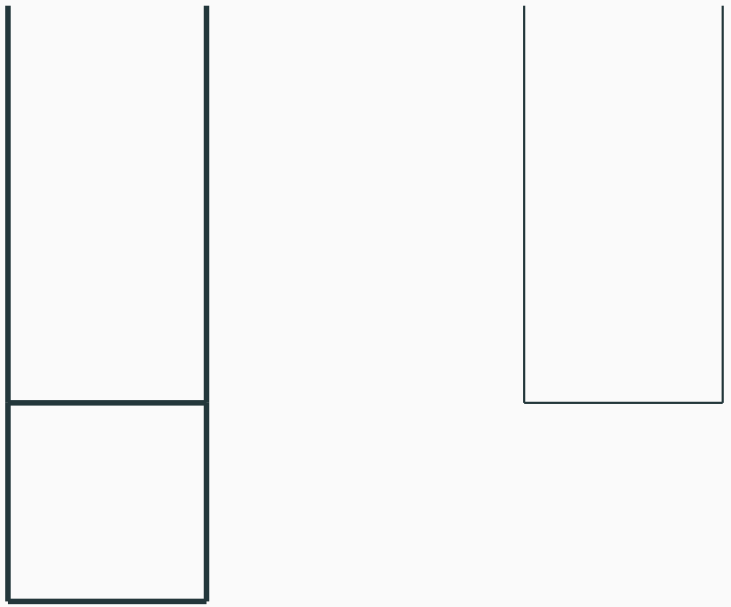
Intuitions en termes de piles d'appel

Continuations (naïves) = copier la pile (en partie) (vers le tas).



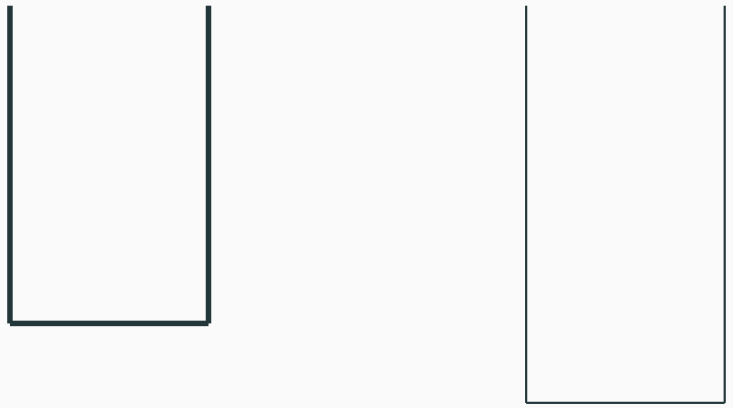
Intuitions en termes de piles d'appel

Continuations (naïves) = copier la pile (en partie) (vers le tas).



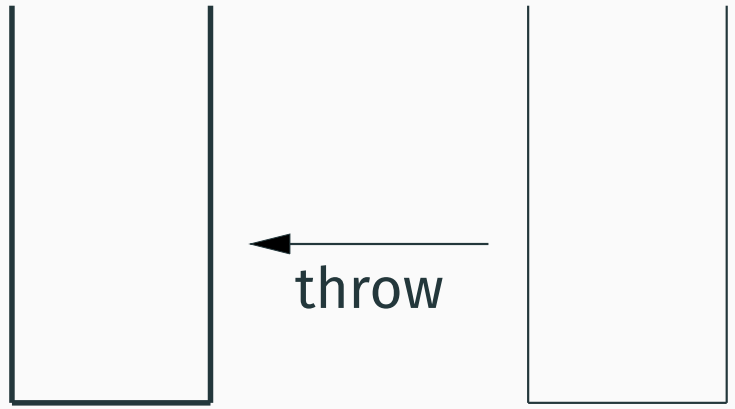
Intuitions en termes de piles d'appel

Continuations (naïves) = copier la pile (en partie) (vers le tas).



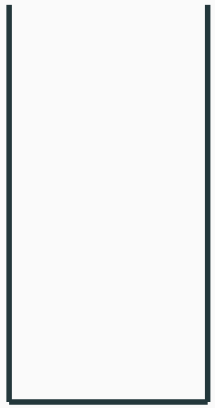
Intuitions en termes de piles d'appel

Continuations (naïves) = copier la pile (en partie) (vers le tas).



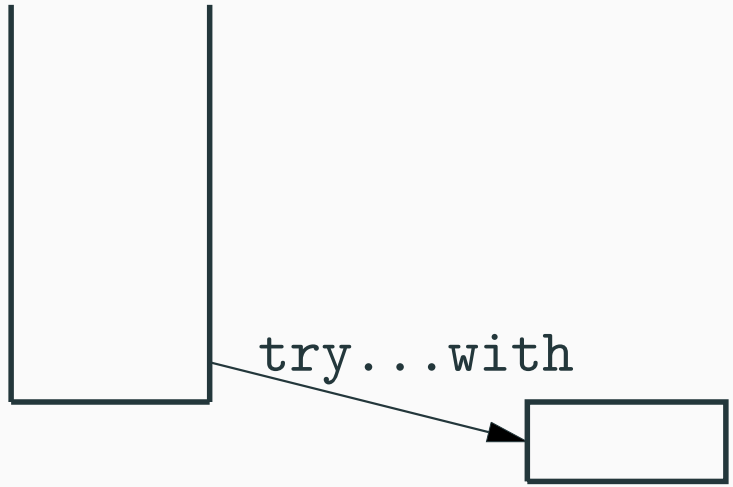
Intuitions en termes de piles d'appel

Gestionnaires d'effets = commuter entre plusieurs piles.



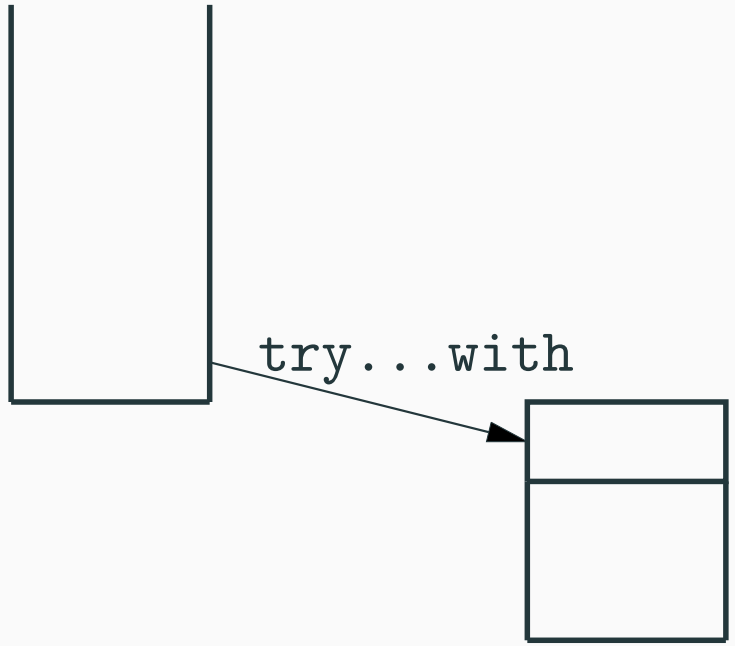
Intuitions en termes de piles d'appel

Gestionnaires d'effets = commuter entre plusieurs piles.



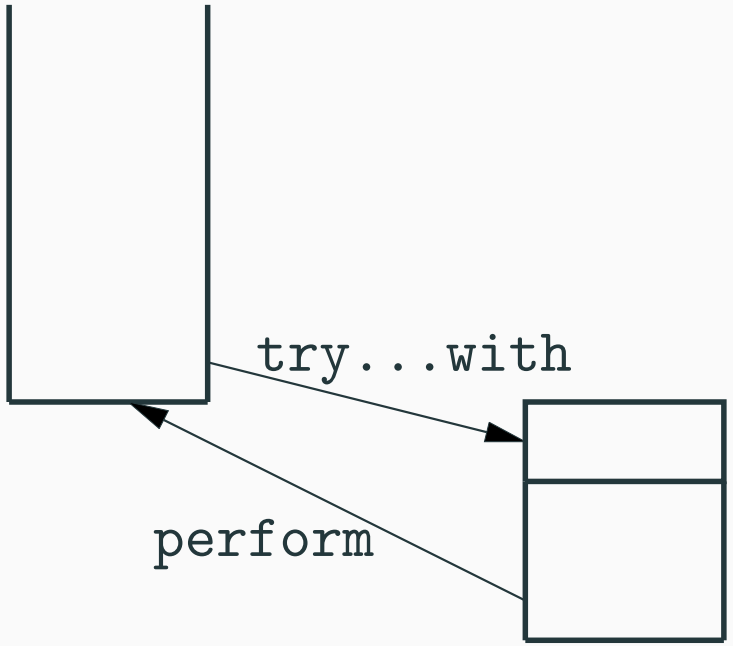
Intuitions en termes de piles d'appel

Gestionnaires d'effets = commuter entre plusieurs piles.



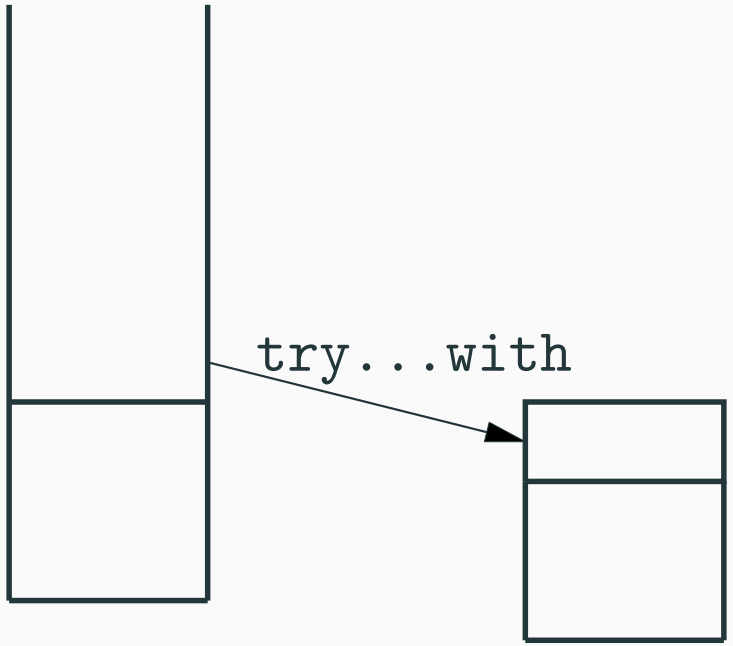
Intuitions en termes de piles d'appel

Gestionnaires d'effets = commuter entre plusieurs piles.



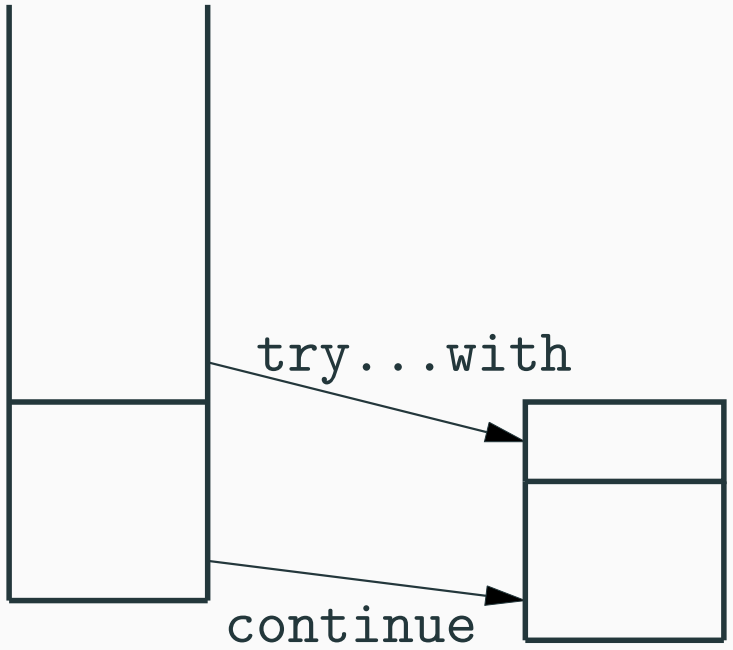
Intuitions en termes de piles d'appel

Gestionnaires d'effets = commuter entre plusieurs piles.



Intuitions en termes de piles d'appel

Gestionnaires d'effets = commuter entre plusieurs piles.



Effets en OCaml 5.0 : pas de syntaxe, juste une bibliothèque

Module `Effect` et sous-module `Effect.Deep`.

```
type _ t = ..   (* type extensible des effets *)

val perform : 'a t -> 'a

type ('a,'b) continuation

val continue: ('a, 'b) continuation -> 'a -> 'b

type 'a effect_handler =
  { effc: 'b. 'b t -> (('b, 'a) continuation -> 'a) option }
                (* Some = je gère, None = je passe *)

val try_with: ('b -> 'a) -> 'b -> 'a effect_handler -> 'a
```

L'exemple en OCaml 5.0

```
open Effect
open Effect.Deep

type _ Effect.t += Conversion_failure : string -> int Effect.t

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

let sum_stringlist lst = ...

let safe_sum_stringlist lst =
  try_with sum_stringlist lst
  { effc = (fun (type c) (eff: c Effect.t) ->
      match eff with
      | Conversion_failure s -> Some (fun (k: (c,_) continuation) ->
          printf "Bad input: %s, replaced by 0\n" s;
          continue k 0)
      | _ -> None
    ) }
```

Deep handlers et *Shallow handlers*

Deux sémantiques pour les gestionnaires d'effets :

- *Deep* : le `continue` reste dans la portée du gestionnaire ; il continue à gérer les effets suivant le `continue`.
- *Shallow* : le gestionnaire disparaît dès le premier effet capturé ; il faut mettre un autre gestionnaire autour du `continue`.

Ce tutoriel utilise la sémantique *deep*, généralement plus simple.

La sémantique *shallow* est aussi disponible en OCaml 5.0 et peut être utile pour implémenter des ≪protocoles≫ d'effets.

Un exemple de *shallow handler*

Corrige la première erreur de conversion mais pas les suivantes.

```
let safe1_sum_stringlist l =
  continue_with (fiber sum_stringlist) l
    { retc = (fun x -> x);
      effc = (fun (type c) (eff: c Effect.t) ->
        match eff with
        | Conversion_failure s -> Some (fun (k: (c,_) continuation) ->
            printf "This time I'll let it pass\n";
            continue_with k 0
              { retc = (fun x -> x);
                effc = (fun (type c) (eff: c Effect.t) -> None);
                exnc = (fun exn -> raise exn) })
        | _ -> None);
      exnc = (fun exn -> raise exn) }
```

Itérateurs et générateurs

Itérateurs (fonctionnels)

Appliquer une fonction de type $\alpha \to$ `unit` sur chacun des éléments de type α d'une collection.

Exemple : `List.iter : ('a -> unit) -> 'a list -> unit`.

Exemple : itérer sur tous les nombres premiers (représentables).

```
let iter_primes (f: int -> unit) : unit =
  for n = 2 to max_int do
    if isprime n then f n
  done

let _ = iter_primes (fun n -> printf "%d\n" n)
```

Générateurs

Produire à la demande, un par un, les éléments d'une collection.

Exemple : la classe `Iterator` en Java (`hasNext`, `next`).

Exemple : les «séquences» (module `Seq`) en OCaml.

```
let gen_primes : int Seq.t =
  let rec gen n : int Seq.t =
    if isprime n
    then (fun () -> Seq.Cons(n, gen (n + 1)))
    else gen (n + 1)
  in gen 2

let p = Seq.drop 1000 gen_primes |> Seq.take 20 |> List.of_seq
```

D'un itérateur à un générateur par inversion de contrôle

Inversion de contrôle : « f appelle g » devient « g interroge f ».

Exemple : implémenter `gen_primes` à partir de `iter_primes` en inversant le contrôle grâce à l'effet `Next_prime`.

```
effect Next_prime : int -> unit
```

```
let gen_primes : int Seq.t =
  match iter_primes (fun n -> perform (Next_prime n)) with
  | () ->
        Seq.empty
  | effect (Next_prime n) k ->
        fun () -> Seq.Cons(n, continue k ())
```

Note : il faut aussi gérer le cas où `iter_primes` termine normalement, d'où `match ... with effect ...`.

Effets en OCaml 5.0 : pas de syntaxe, juste une bibliothèque

Le combinateur `match_with` généralise `try_with` en permettant de gérer aussi les cas de sortie normale et de sortie sur exception.

```
type ('a,'b) handler =
    { retc: 'a -> 'b;
      exnc: exn -> 'b;
      effc: 'c.'c t -> (('c,'b) continuation -> 'b) option }

val match_with: ('c -> 'a) -> 'c -> ('a,'b) handler -> 'b
```

Transformation générale d'un itérateur en générateur

```
let iterator_to_sequence
      (type elt) (type collection)
      (iter: (elt -> unit) -> collection -> unit)
      : collection -> elt Seq.t =
  let effect Next : elt -> unit in
  fun coll ->
    match iter (fun elt -> perform (Next elt)) coll with
    | () -> Seq.empty
    | effect (Next elt) k ->
            fun () -> Seq.Cons(elt, continue k ())
```

Note : on peut générer des effets dynamiquement et localement.
(En OCaml 5.0 : pas de syntaxe, utiliser des modules locaux.)

Application : *same fringe problem*

Est-ce que deux collections contiennent les mêmes éléments dans le même ordre ?

```
let same_fringe
    (iter1: ('elt -> unit) -> 'coll1 -> unit)
    (iter2: ('elt -> unit) -> 'coll2 -> unit)
    coll1 coll2 =
  Seq.for_all2 (=) (iterator_to_sequence iter1 coll1)
                   (iterator_to_sequence iter2 coll2)
```

Exemple :

```
same_fringe List.iter IntSet.iter
            [1; 2; 3]
            (IntSet.add 2 (IntSet.add 1 (IntSet.singleton 3)))
```

Coroutines

Les coroutines

(Par opposition à subroutine ≈ procédure.)

Exécution pseudo-parallèle, par entrelacement,
de plusieurs tâches.

Chaque tâche rend la main explicitement (instruction `yield`)
pour qu'on puisse passer à la tâche suivante.
(Pas de préemption.)

```
corun
  { print "1"; yield; print "2" }   (* doit produire "1324" *)
  { print "3"; yield; print "4" }
```

Utilisation typique : simulation de réseaux synchrones.

Mon exemple fétiche : animation d'algorithmes de tri,
synchronisés sur les étapes de comparaison.

Implémenter les coroutines avec des effets

Deux opérations de base :

- `yield : unit -> unit`
  pour ≪passer la main≫ au calcul suivant
- `spawn : (unit -> unit) -> unit`
  pour lancer un nouveau calcul en parallèle.

On n'a aucune idée de comment les implémenter… alors on en fait des effets !

```
effect Yield : unit
let yield () = perform Yield
effect Spawn : (unit -> unit) -> unit
let spawn f = perform (Spawn f)
```

Ordonnancement des tâches

Classiquement, on a une tâche en cours d'exécution et une file d'attente de 0, 1 ou plusieurs tâches prêtes à redémarrer.

```
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 ()
```

Gestion des effets et ordonnancement des tâches

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

Gestion des effets et ordonnancement des tâches

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

Pour `Yield` : la continuation est ajoutée à la fin de la queue des tâches prêtes ; la première tâche prête est relancée.

Gestion des effets et ordonnancement des tâches

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

Pour `Yield` : la continuation est ajoutée à la fin de la queue des tâches prêtes ; la première tâche prête est relancée.

Pour `Spawn f` : on a choisi de lancer la fonction `f` plus tard et de redémarrer immédiatement après le `spawn`. On aurait pu faire

```
  | effect (Spawn f) k -> suspend (continue k); corun f
```

Gestion des effets et ordonnancement des tâches

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

Pour `Yield` : la continuation est ajoutée à la fin de la queue des tâches prêtes ; la première tâche prête est relancée.

Pour `Spawn f` : on a choisi de lancer la fonction `f` plus tard et de redémarrer immédiatement après le `spawn`. On aurait pu faire

```
  | effect (Spawn f) k -> suspend (continue k); corun f
```

En revanche il faut faire `suspend (corun f)` et non `suspend f`, sinon les effets de `f ()` ne sont pas gérés.

Exemple d'utilisation

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

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

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

Vive le style direct!

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

Ces calculs sont écrits en style direct avec les structures de contrôle usuelles d'OCaml, et s'exécutent à pleine vitesse tant qu'ils ne lèvent pas d'effets.

Comparer avec le style CPS ou le style monadique qui étaient nécessaires auparavant :

```
let cps_task name n k =
  let rec task i k =
    if i >= n then k () else begin
      printf "%s%d " name i;
      yield (fun _ -> task (i + 1) k)
    end
  in task 1 k
```

Ajout de canaux de communication

Un type `'a channel` des canaux transmettant des valeurs de type `'a` et trois fonctions

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

On choisit une sémantique ≪rendez-vous≫ (CCS, π-calcul) :
`send ch v` bloque jusqu'à ce qu'une autre tâche fasse `recv ch` ;
les deux tâches redémarrent ;
`recv ch` retourne 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
  }
```

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 :

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

effect Send : 'a channel * 'a -> unit
let send ch v = perform (Send(ch, v))

effect Recv : 'a channel -> 'a
let recv ch = perform (Recv ch)
```

L'ordonnanceur étendu avec les canaux

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

Entrées/sorties asynchrones

Gérer des masses d'entrées-sorties

Exemple : serveur ou proxy Web avec des milliers de connexions simultanées.

Modèle synchrone :

- Un *thread* système par connexion.
- Entrées/sorties bloquantes, faciles à programmer.
- Ne passe pas à l'échelle.

Modèle asynchrone :

- Les connexions sont multiplexées sur un ou un petit nombre de *threads* système.
- Entrées/sorties non bloquantes + *polling* intelligent.
- Difficile à programmer.

Entrées-sorties non bloquantes

Que faire lorsqu'il n'y a rien à lire ? ou que l'écriture bloquerait ?

lancer un effet, bien sûr !

```
effect Wait_for_reading : file_descr -> unit
effect Wait_for_writing : file_descr -> unit

let rec recv fd buf pos len fl =
  try
    Unix.recv fd buf pos len fl
  with Unix_error((EAGAIN|EWOULDBLOCK), _, _) ->
    perform (Wait_for_reading fd); recv fd buf pos len fl

let rec send fd buf pos len fl =
  try
    Unix.send fd buf pos len fl
  with Unix_error((EAGAIN|EWOULDBLOCK), _, _) ->
    perform (Wait_for_writing fd); send fd buf pos len fl
```

Recenser les tâches en attentes sur des descripteurs de fichiers

```
let waiting_reads
   : (file_descr, (unit,unit) continuation) Hashtbl.t
   = Hashtbl.create 16
let waiting_writes
   : (file_descr, (unit,unit) continuation) Hashtbl.t
   = Hashtbl.create 16
```

Relancer une tâche en attente sur le descripteur `fd` :

```
let restart_io tbl fd =
  let k = Hashtbl.find tbl fd in
  Hashtbl.remove tbl fd;
  suspend (continue k)
```

L'ordonnancement des tâches et des E/S

S'il n'y a pas de tâche prête à tourner, regarder les tâches en attente d'E/S, voir quels descripteurs sont prêts, et débloquer les tâches correspondantes.

```
let rec restart () =
  match Queue.take_opt runnable with
  | Some f -> f ()
  | None ->
      let rd_fds = Hashtbl.fold (fun fd _ a -> fd::a) waiting_reads []
      and wr_fds = Hashtbl.fold (fun fd _ a -> fd::a) waiting_writes []
      in
      if rd_fds = [] && wr_fds = [] then () else begin
        let rdy_rd_fds, rdy_wr_fds, _ =
          Unix.select rd_fds wr_fds [] (-1.) in
        List.iter (restart_io waiting_reads) rdy_rd_fds;
        List.iter (restart_io waiting_writes) rdy_wr_fds;
        restart()
      end
```

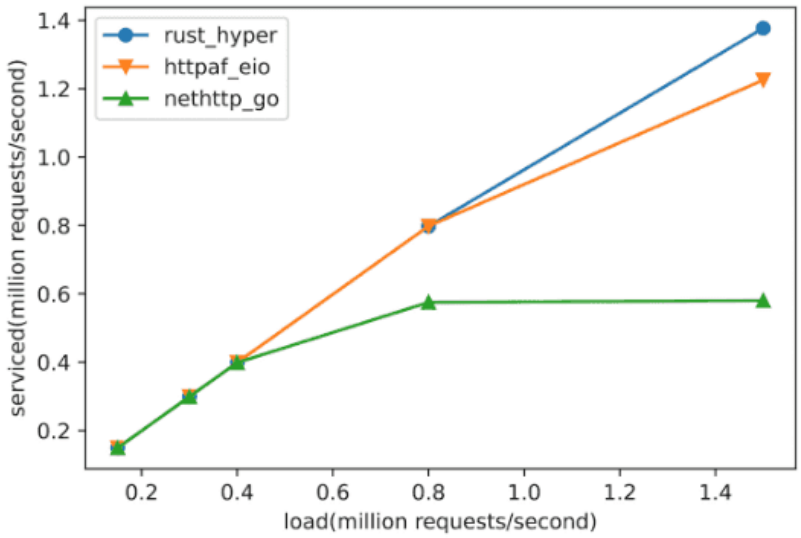
L'ordonnanceur étendu avec les entrées/sorties

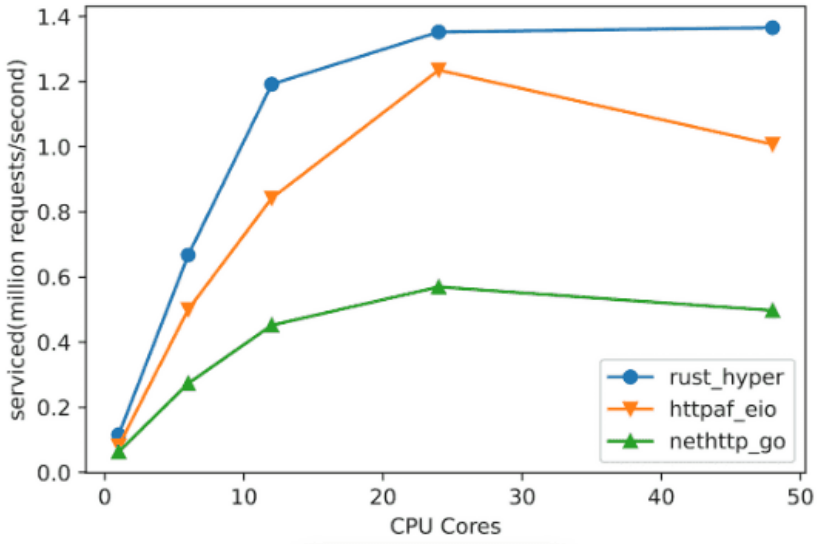
```
let rec corun (f: unit -> unit) =
  match f () with
  | () -> restart ()
  | effect Yield k -> suspend (continue k); restart ()
  | effect (Spawn f) k -> suspend (corun f); continue k ()
  | effect (Wait_for_reading fd) k ->
      Hashtbl.add waiting_reads fd k; restart()
  | effect (Wait_for_writing fd) k ->
      Hashtbl.add waiting_writes fd k; restart()
```

Le projet EIO

https://github.com/ocaml-multicore/eio

Eio provides an effects-based direct-style IO stack for OCaml 5.0. For example, you can use Eio to read and write files, make network connections, or perform CPU-intensive calculations, running multiple operations at the same time. It aims to be easy to use, secure, well documented, and fast.





Point d’étape

Effets et gestionnaires d’effets

=

de nouvelles structures de contrôle

définissables par l’utilisateur ou en bibliothèques

et compatibles avec la programmation en style direct