Concurrence et continuations en OCaml

Christophe Deleuze

Concurrence et continuations en OCaml

C. Deleuze

Laboratoire de Conception et d'Intégration des Systèmes
50 rue Barthélémy de Laffemas
26902 Valence Cedex 09
France
christophe.deleuze@lcis.grenoble-inp.fr

Résumé

Nous décrivons et mettons en œuvre différentes méthodes pour fournir la concurrence légère en OCaml, en style *direct* ou *indirect*. Nous montrons que toutes ces approches font appel, explicitement ou implicitement, à la notion de continuation. Nous essayons de préciser leurs relations et comparons leur performances sur deux applications concurrentes simples. Presque toutes les mises en œuvre légères sont très largement meilleures que les threads système ou de la machine virtuelle.

1. Introduction

On peut définir la concurrence comme une propriété d'un système dans lequel plusieurs "fils d'éxécution" existent et progressent simultanément. Ces *threads* représentent des traitements entre lesquels il n'y a pas de dépendance temporelle par défaut : les dépendances sont exprimées explicitement par des opérations de synchronisation.

Les systèmes d'exploitation fournissent cette abstraction mais avec un coût relativement important, en mémoire utilisée par *thread* et en temps pour changer de contexte. L'ordonnancement est généralement préemptif ce qui rend délicat le partage de données entre les threads.

Il est possible d'obtenir une concurrence plus légère en la mettant en œuvre au niveau de l'application. De nombreuses approches ont été proposées avec généralement un ordonnancement coopératif. Dans cet article nous montrons que toutes ces approches font appel, explicitement ou implicitement, à la notion de continuation. Nous essayons de préciser leurs relations et de comparer leurs performances au travers de mises en œuvre simples dans le cadre du langage OCaml.

2. Un modèle simple de concurrence

Nous proposons un modèle simple de concurrence, défini par les primitives suivantes :

spawn prend un *thunk* et crée un nouveau thread à exécuter. Celui-ci ne commence son exécution que si/quand la primitive *start* a été invoquée.

yield suspend le thread, permettant aux autres de s'exécuter.

halt termine le thread. Si tous les threads sont terminés, *start* retourne.

start démarre les threads créés par *spawn* et bloque jusqu'à ce qu'ils soient tous terminés.

stop stoppe definitivement tous les threads. *start* retourne.

Les threads peuvent communiquer à l'aide de MVars. Introduites en Haskell [14], les MVars sont des variables mutables partagées qui permettent la communication et la synchronisation entre threads. On définit les trois opérations suivantes :

make_mvar crée une nouvelle MVar vide.

take_mvar retire la valeur d'une MVar.

put_mvar place une valeur dans une MVar.

Une MVar peut contenir une valeur ou être vide. Les opérations *take* et *put* bloquent si la MVar est respectivement vide ou pleine. Pour simplifier nous considérons que pour chaque MVar un seul thread souhaite écrire et un seul thread souhaite lire à un moment donné.

3. Deux applications

Nous décrivons ci-dessous deux exemples d'applications qui nous permettront d'évaluer le style de programmation requis par notre modèle et chacune de ses mises en œuvre. Elles serviront aussi à faire quelques comparaisons de performances.

3.1. Crible d'Ératosthène

Notre première application est le crible d'Ératosthène. Cette version concurrente est un classique notamment présentée dans [7] (qui indique que sa première mention est dans [13]). Une variante apparaît également dans [6]. Le programme est structuré en une chaîne de threads échangeant des messages :

integers est le générateur, il émet tous les entiers à partir de 2,

filter n relaie les nombres qu'il reçoit s'ils ne sont pas multiples de son paramètre n,

sift crée et insère un nouveau filtre dans la chaîne à chaque nombre reçu,

output affiche les nombres qu'il reçoit.

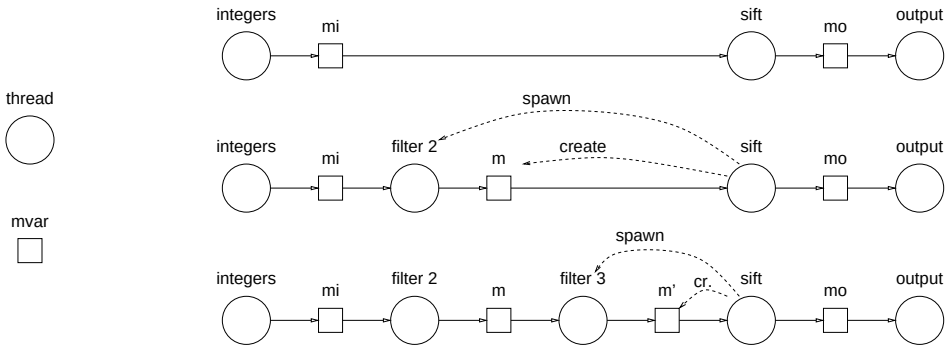


FIGURE 1 – Le crible concurrent

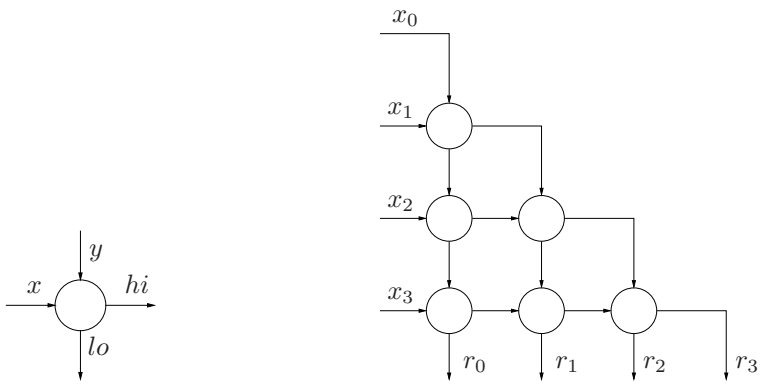
Ainsi le crible se construit comme une chaîne de threads *filter* encadrés par un générateur (*integers*) sur la gauche et un "étendeur" (*sift*) suivi d'un consommateur (*output*) sur la droite. Les messages sont échangés au moyen de MVars. La figure 1 montre les threads au démarrage et après que le premier puis le deuxième nombres premiers aient été trouvés.

Le code, visible figure 12 en style direct et indirect, est particulièrement simple.

3.2. Tri concurrent

Notre second exemple est un tri concurrent décrit dans [6], qui explique que le tri bulle et le tri par insertion sont des versions *séquentialisées* de cet algorithme.

L'algorithme utilise un réseau de threads *comparateur* très simples, chacun utilisé pour trier une paire de valeurs depuis deux MVars d'entrée vers deux MVars de sortie. Un comparateur avec les entrées x et y et les sorties hi and lo est montré figure 2(a). La figure 2(b) montre un réseau pour trier une liste de 4 valeurs.



(a) Comparateur (b) Réseau de tri

FIGURE 2 – Tri concurrent

Les MVars n'apparaissent pas sur la figure, elles stockeront les valeurs initiales et finales et serviront pour la communication entre les comparateurs. Nous ne montrons pas le code par manque de place.

4. Réalisations

Notre ordonnanceur sera une simple boucle extrayant les threads d'une file FIFO (*runq*) et les exécutant. Nous pouvons d'ores et déjà implémenter les primitives *start* et *stop*. L'implémentation des autres primitives dépendra de chaque réalisation.

```
let runq = Queue.create ()
let enqueue t = Queue.push t runq
let dequeue () = Queue.take runq

exception Stop
let stop () = raise Stop
```

```
let start () =
  try
    while true do
      dequeue () ()
    done
  with Queue.Empty | Stop → ()
```

4.1. Réalisations en style direct

Dans le cas de la programmation en *style direct*, les primitives auront les signatures montrées sur la figure 3. Certaines de ces opérations sont potentiellement bloquantes mais se présentent comme des fonctions ordinaires. Le code du crible est montré figure 12(a).

Le support standard d'OCaml pour les threads (système ou VM) peut être utilisé pour réaliser les primitives avec ces signatures. Nous ne décrirons pas cette réalisation qui nous servira uniquement de référence pour comparer les performances des réalisations légères.

```
val yield : unit → unit
val spawn : (unit → unit) → unit
val halt : unit → unit

val start : unit → unit
val stop : unit → unit
```

```
type α mvar
val make_mvar : unit → α mvar
val take_mvar : α mvar → α
val put_mvar : α mvar → α → unit
```

FIGURE 3 – Signatures des primitives pour le style direct

Pour suspendre un thread et le réactiver plus tard, nous devons pouvoir sauver la "situation courante" du thread. Cela correspond à la notion de *continuation*, qui représente à un point de l'exécution d'une fonction ce qui reste à exécuter [2], c'est à dire le contexte de l'éxécution.

La primitive *call with current continuation* (ou *call/cc*) a été introduite par le langage Scheme [8]. Celle-ci réalise une copie de (ou "capture") la continuation actuelle et la *réifie* (la rend manipulable par le programme) en une valeur de type α *cont.* [1] Les continuations peuvent ainsi être manipulées explicitement par le programme comme n'importe quelle autre valeur. Une telle "continuation de première classe" peut être lancée (*throw*) avec un paramètre de type α auquel cas elle écrase la continuation courante, de façon que l'éxécution reprenne au point où la continuation avait été capturée. Ceci permet de manipuler le flot de contrôle du programme et en particulier d'introduire la notion de threads.

Nous avons réalisé une mise en œuvre basée sur *call/cc* similaire à celle décrite dans [4] mais ne la présenterons pas ici : la seule mise en œuvre existante de *call/cc* pour OCaml est décrite par son auteur comme "très naïve" avec des performances "terribles" [11].

Nous utiliserons la bibliothèque `caml-shift` [15] qui offre des *continuations délimitées*. Une telle continuation est un préfixe de ce qui reste à exécuter, représentée par une portion délimitée du contexte. Une telle continuation (aussi appelée partielle, composable, or sous continuation) retourne une valeur et peut donc être réutilisée et composée.

La littérature définit un certain nombre d'opérateurs associés. L'idée générale est qu'une telle continuation est construite en commençant par pousser un délimiteur (ou *prompt*) sur la pile puis en capturant la continuation jusqu'à un prompt. Dans cette bibliothèque *push_prompt* pousse un délimiteur, tandis que *take_subcont* retire de la pile le fragment jusqu'au prompt inclu et retourne le fragment (sans le prompt) sous forme d'une valeur de type (α, β) *subcont* [2] où α est le type de la valeur à passer au lancer de la continuation, β étant celui retourné par la continuation. Enfin, *push_subcont* pousse une continuation sur la pile (*ie* la lance).

Ainsi, pour suspendre un thread nous devons capturer sa continuation. Pour le réactiver nous devons pousser le prompt et la continuation. Nous choisissons d'encapsuler immédiatement la continuation capturée dans une fonction qui poussera le prompt et la continuation, ce qui est le comportement de l'opérateur *shift0* (celui-ci retire le prompt de la pile et l'inclut dans la continuation capturée) [9]. Ainsi l'ordonnanceur doit simplement exécuter la fonction pour relancer le thread (voir le *dequeue* () () dans la boucle *start* plus haut).

```
let prompt = new_prompt ()

let shift0 p f = take_subcont p (fun sk () →
  (f (fun c → push_prompt_subcont p sk (fun () → c))))
```

1. En Scheme, les continuations sont réifiées sous forme de fonctions. Appliquer la fonction revient à lancer la continuation. La description que nous donnons ici correspond plus à ce qui peut être mis en œuvre dans un langage typé statiquement comme OCaml.
2. Ce comportement correspond à l'opérateur *control0* [9].

```
let yield () = shift0 prompt (fun f → enqueue f)

let halt () = shift0 prompt (fun f → ())
```

halt "nettoie" la pile en supprimant le prompt et un éventuel reste de contexte. La définition de *shift0* donnée ci-dessus est une version optimisée[3] de

```
let shift0 p f = take_subcont p (fun sk () →
  (f (fun c → push_prompt p
       (fun () → push_subcont sk (fun () → c)))))
```

spawn encapsule son argument dans une fonction qui commence par pousser le prompt et se termine en appelant *halt* de façon à assurer que le prompt soit retiré de la pile quand le thread termine.

```
let spawn t = enqueue (fun () → push_prompt prompt (fun () → t (); halt ()))
```

Une MVar est un simple struct avec trois valeurs de type option : la valeur stockée dans la MVar, le thread bloqué sur une opération *take_mvar*, le thread bloqué sur une operation *put_mvar* avec la valeur qu'il veut placer. Quand un thread bloque, sa continuation est capturée et encapsulée par *shift0* puis stockée dans le champ approprié. La fonction sera replacée dans la file d'exécution quand le thread sera réactivé. Les *failwith* correspondent à notre hypothèse simplificatrice d'un seul thread souhaitant lire ou écrire dans une MVar donnée à un moment donné.

```
type α t = α → unit
type α mvar = { mutable v : α option;
                mutable read : α t option;
                mutable write : (unit t × α) option }

let make_mvar () = { v = None; read = None; write = None }

let put_mvar out v =
  match out with
  | { v = Some v'; read = _; write = None } →
      shift0 prompt (fun f → out.write ← Some (f, v))

  | { v = None; read = Some r; write = None } →
      out.read ← None; enqueue (fun () → r v)

  | { v = None; read = None; write = None } → out.v ← Some v

  | { v = _; read = _; write = Some _ } → failwith "put_mvar"

let take_mvar inp =
  match inp with
  | { v = Some v; read = None; write = None } → inp.v ← None; v

  | { v = Some v; read = None; write = Some(c, v') } →
      inp.v ← Some v'; inp.write ← None; enqueue c; v

  | { v = None; read = None; write = _ } →
      shift0 prompt (fun f → inp.read ← Some f)

  | { v = _; read = Some _; write = _ } → failwith "take_mvar"
```

4.2. Réalisations en style indirect

Dans le *style indirect* le code des applications est rédigé de manière à rendre les continuations explicites (sous forme de fermeture) à chaque point de coopération. Ainsi les continuations peuvent être manipulées sans avoir recours à une primitive de capture.

3. En fait la version non optimisée présente une fuite de mémoire (voir [10, Appendix B]).

Voyez le code du thread *sift* du crible sur la figure 12(b). Nous notons >>= (appelé *bind*) l'opérateur de composition séquentielle. [4] Ce opérateur est présent aux points de coopération, entre une opération potentiellement bloquante et sa continuation. Cette continuation est une fermeture qui sera exécutée quand l'opération bloquante sera terminée, elle en recevra le résultat en paramètre. Ce style impose de convertir les boucles impératives en fonctions récursives si elles contiennent une opération bloquante. Nous définirons une opération supplémentaire en style indirect : *skip*, l'opération vide.

Nous décrivons dans la suite la réalisation de notre modèle avec les styles *trampoline*, *monadique* (basé sur des continuations ou des promesses) et par événements. Tous sont des variantes du style indirect.

4.3. Style trampoline

Dans le style trampoline [5] chaque fonction bloquante reçoit sa continuation en paramètre. C'est une variante du style "passage de continuation" (CPS, [17]) où les continuations ne sont rendues explicites qu'aux points de coopération.

La figure 4 montre les signatures des opérations. Chaque opération potentiellement bloquante prend en paramètre sa continuation, une fonction à exécuter quand l'opération a été réalisée.

```
val skip : (unit → unit) → unit
val ( >>= ) : ((α → unit) → unit) → (α → unit) → unit

val yield : (unit → unit) → unit
val spawn : (unit → unit) → unit
val halt : unit → unit

val start : unit → unit
val stop : unit → unit

type α mvar
val make_mvar : unit → α mvar
val take_mvar : α mvar → (α → unit) → unit
val put_mvar : α mvar → α → (unit → unit) → unit
```

FIGURE 4 – Signatures des primitives pour le style trampoline

Par exemple l'opération *yield* (ci-dessous à gauche) prend en argument sa continuation, c'est à dire la fonction à exécuter quand le thread sera réactivé. On peut obtenir une syntaxe plus plaisante (ci-dessous à droite) si l'on définit l'opérateur infixe >>= comme appliquant son premier argument au deuxième (la continuation) [5] et si nous adoptons une indentation reflétant l'idée d'exécution en séquence.

```
print_string "hoho";
  yield (fun () →
    print_string "haha";
    yield (fun () →
      ...
        ))
```

```
let (>>=) inst (k : α → unit) : unit = inst k
...
print_string "hoho";
yield >>= fun () →
print_string "haha";
yield >>= fun () →
...
```

Rendre les continuations explicites n'est pas nécessairement aussi intrusif qu'on pourrait le penser. Dans le code du crible elles n'apparaissent en fait jamais (car les fonctions sont "récursives infinies"). Elles ne doivent être mentionnées que quand on construit des opérations bloquantes complexes, comme

4. C'est un emprunt à la syntaxe des monades mais ne représente pas nécessairement l'opérateur *bind* des monades comme nous le verrons.

5. Les programmeurs Haskell penseront à l'opérateur `$`.

les deux exemples ci-dessous. Le premier abstrait l'opération de suspension trois fois d'affilée, le deuxième le transfert d'une valeur d'une MVar à une autre :

```
let yield3 k =
  yield >>= fun () →
  yield >>= fun () →
  yield >>=
  k
```

```
let transfer_mvar m1 m2 k =
  take_mvar m1 >>= fun v →
  put_mvar m2 v >>=
  k
```

Réalisation Une fonction bloquante, comme *yield* ou *take_mvar*, reçoit sa continuation en paramètre, elle peut l'exécuter immédiatement ou, si elle doit bloquer, la stocker avant de retourner.

Le code est très proche de celui utilisé avec les captures de continuations délimitées : la différence est que nous n'avons pas à les capturer puisqu'elles sont fournies explicitement.

```
let skip k = k ()
let yield k = enqueue k
let halt () = ()
```

```
let spawn t = enqueue t
let close k = fun () → k (fun _ → ())
```

```
let take_mvar inp k =
  match inp with
  | { v = Some v; read = None; write = None } → inp.v ← None; k v
  | { v = Some v; read = None; write = Some(c, v') } →
      inp.v ← Some v'; inp.write ← None; enqueue c; k v
  | { v = None; read = None; write = _ } → inp.read ← Some(k)
```

Un mot sur *spawn* : composer des fragments avec >>= produit généralement une fonction "ouverte", prenant une continuation, qu'il faut lui fournir afin de la "fermer" avant de l'exécuter comme thread. Cependant les threads définis pour le crible sont des récursions infinies qui ne prennent pas un tel paramètre. Nous séparons donc la fermeture de la fonction du lancement du thread (fonctions *close* et *spawn*).

4.4. Monade de continuation

La monade de continuation est définie dans [3]. Les monades sont utiles pour gérer les effets dans un environnement fonctionnel pur [14]. Une monade est un type $\alpha\ t$ qui offre (au moins) les primitives *return* et *bind* (noté comme opérateur infixe >>=) montrées figure 5. Comme le suggèrent les types, *return* v construit une valeur monadique de type $\alpha\ t$ "contenant" v et m >>= f "ouvre" m pour extraire la valeur v, la donne à f et retourne la valeur monadique retournée par f.

Ici nous pouvons définir type $\alpha\ t = (\alpha \to unit) \to unit$, une valeur de type $\alpha\ t$ est une fonction prenant une continuation de type $\alpha \to unit$. Nous pouvons alors voir les fonctions bloquantes comme retournant une valeur de type $\alpha\ t$ (figure 5).

L'opérateur *bind* est un peu plus compliqué que celui utilisé dans le style trampoline. Il prend en charge la gestion des continuations qui n'ont plus à apparaître explicitement dans la composition de fragments (nous avons changé très légèrement la définition de *yield* afin de lui faire prendre un argument ()) :

```
let yield3 () =
  yield () >>= fun () →
  yield () >>= fun () →
  yield ()
```

```
let transfer_mvar m1 m2 =
  take_mvar m1 >>= fun v →
  put_mvar m2 v
```

Pour rendre ceci plus clair, voici les définitions de *return* et *bind* ainsi que leurs types "développés" :

type $\alpha\ t$
val $return\ :\ \alpha\ \rightarrow\ \alpha\ t$
val $(\ >>=\ )\ :\ \alpha\ t\ \rightarrow\ (\alpha\ \rightarrow\ \beta\ t)\ \rightarrow\ \beta\ t$

val $spawn\ :\ (unit\ \rightarrow\ unit\ t)\ \rightarrow\ unit$
val $skip\ :\ unit\ t$
val $yield\ :\ unit\ \rightarrow\ unit\ t$
val $halt\ :\ unit\ \rightarrow\ unit\ t$
val $stop\ :\ unit\ \rightarrow\ unit\ t$
val $start\ :\ unit\ \rightarrow\ unit$

type $\alpha\ mvar$
val $make\_mvar\ :\ unit\ \rightarrow\ \alpha\ mvar$
val $put\_mvar\ :\ \alpha\ mvar\ \rightarrow\ \alpha\ \rightarrow\ unit\ t$
val $take\_mvar\ :\ \alpha\ mvar\ \rightarrow\ \alpha\ t$

FIGURE 5 – Signatures des primitives pour le style monadique

val $return\ :\ \alpha\ \rightarrow\ (\alpha\ \rightarrow\ unit)\ \rightarrow\ unit$
let $return\ a\ =$ fun $k\ \rightarrow\ k\ a$

val $(>>=)\ :\ ((\alpha\ \rightarrow\ unit)\ \rightarrow\ unit)\ \rightarrow\ (\alpha\ \rightarrow\ (\beta\ \rightarrow\ unit)\ \rightarrow\ unit)\ \rightarrow\ (\beta\ \rightarrow\ unit)\ \rightarrow\ unit$
let $(>>=)\ f\ k'\ =$ fun $k\ \rightarrow\ f$ (fun $r\ \rightarrow\ k'\ r\ k$)

c'est à dire que *bind* retourne une fonction qui accepte une continuation. Voyons la forme développée de notre exemple *transfer_mvar* :

$transfer\_mvar\ m1\ m2\ \triangleq$ fun $k\ \rightarrow\ take\_mvar\ m1$ (fun $r\ \rightarrow$ (fun $v\ \rightarrow\ put\_mvar\ m2\ v)\ r\ k$)

Enfin, *spawn* fournit la continuation vide et place la fonction en file d'exécution.

let $spawn\ (t\ :\ unit\ \rightarrow\ unit\ t)\ =\ enqueue$ (fun $()\ \rightarrow\ t\ ()$ (fun $()\ \rightarrow\ ()$))

Comme on le voit, tout ceci est très proche du style trampoline. Celui-ci a été proposé dans le cadre du langage Scheme, on peut voir la monade de continuation essentiellement comme une version typée du style trampoline.

4.5. Monade de promesse

Une *promesse* [12] est une valeur qui peut être utilisée pour accéder plus tard à une valeur qui n'est pas nécessairement disponible immédiatement.

Une opération qui bloquerait peut retourner immédiatement une promesse. Une promesse peut être *prête* si la valeur est disponible ou *bloquée* si elle ne l'est pas encore. La promesse peut être simplement passée jusqu'au point où la valeur qu'elle représente est effectivement requise : l'opération *claim* permet d'extraire cette valeur (elle peut bien évidemment bloquer).

Les signatures des opérations seront les même que pour la monade de continuation (figure 5). Les opérations bloquantes retournent une promesse, le *claim* étant effectué par *bind*. Dans $t\ >>=\ f$, *bind* devra soit :

- passer à f la valeur de t si elle est prête (et retourner alors la promesse retournée par f), ou
- retourner une promesse bloquée et faire en sorte que :
  - f reçoive la valeur promise dès que t devient prête,
  - la promesse bloquée retournée soit "la même" que celle retournée par f.

Réalisation Voici une brève description d'une réalisation possible (figure 6). Celle-ci est très largement inspirée de `Lwt` (light weigth threads) [18], une bibliothèque OCaml de threads légères. Nous essayons de n'en retenir ici que les aspects les plus fondamentaux pour faciliter la comparaison avec les autres réalisations. Nous interprétons ici le type $\alpha\ t$ comme le type des promesses alors que [18] l'interprète comme le type des threads.

Le type $\alpha\ t$ des promesses pour une valeur de type α est un struct avec un champ mutable contenant l'état de la promesse. Celle-ci peut être *Ready* avec la valeur promise, ou *Blocked* avec une liste de "patients" *(waiters)* à exécuter quand elle sera prête, ou *Link* (liée) à une autre (cela signifie qu'elles se comporteront de la même façon, ayant été *connect*ées). *repr* permet d'obtenir la promesse "canonique".

Considérons l'évaluation de t >>= f. L'évaluation de t fournit une promesse. Si elle est prête sa valeur v est passée à f. Si elle est bloquée, sa valeur est *Blocked* w. Une nouvelle promesse bloquée *res* est créée, un patient est ajouté à la liste w, puis *res* est retournée.

Ainsi, quand t devient prête (par *fullfill*), le patient est exécuté. Il passe la valeur à f qui retourne une nouvelle promesse (appelons là p). *res* est alors *connect*ée à p. Le code pour *connect* montre que si p est prête *res* est comblée *(fullfilled)*. Si p est bloquée elle est changée en un *Link* sur *res* : cela garantit que toute opération ultérieure (telle que *fullfill*) sur p sera en fait effectuée sur *res*.

```
type α state =
  | Ready of α
  | Blocked of (α t → unit) list ref
  | Link of α t

and α t = { mutable st : α state }

let rec repr t =
  match t.st with
  | Link t' → repr t'
  | _ → t

let blocked () = { st = Blocked (ref []) }
let ready v = { st = Ready v }

let runq = Queue.create ()
let enqueue t = Queue.push t runq
let dequeue () = Queue.take runq

let fullfill t v =
  let t = repr t in
  match t.st with
  | Blocked w →
      t.st ← Ready v;
      List.iter (fun f → f t) !w
  | _ → failwith "fullfill"

let connect t t' =
  let t' = repr t' in
  match t'.st with
  | Ready v → fullfill t v
  | Blocked w' →
      let t = repr t in
      match t.st with
      | Blocked w → w := !w @ !w';
                    t'.st ← Link t
      | _ → failwith "connect"

let (>>=) t f =
  match (repr t).st with
  | Ready v → f v
  | Blocked w → let res = blocked () in
    w := (fun t → let Ready v = t.st in
                  connect res (f v)) :: !w;
    res
```

FIGURE 6 – Monade de promesse (1)

Sur la figure 7 on voit que l'ordonnanceur commence par combler la promesse *wait_start*. *spawn* assure que les threads l'attendent de façon à empêcher leur exécution avant le signal donné par *start*. *yield* crée une promesse bloquée p, ajoute en file d'exécution une fonction qui la comblera et retourne p. *bind* ajoutera à p un patient qui *connect*era la promesse de la suite avec sa propre *res*. Quand l'ordonnanceur exécute la fonction ajoutée par *yield*, p est comblée et le patient exécuté. Les MVars sont gérées de manière similaire.

Il n'est pas facile de visualiser l'enchaînement précis des opérations, aussi nous illustrons l'éxécution de notre exemple *yield3* sur la figure 8. Une promesse est symbolisée par un carré contenant un R pour *Ready*, B pour *Blocked* (pointant vers son thunk) ou L pour *Link* (avec le lien).

- (a) À la première occurence de *bind* (indicée 1), le premier *yield* (indicé a) retourne une promesse bloquée p_a. *bind* crée une nouvelle promesse bloquée res_1, ajoute à p_a un patient *connect*ant res_1 au reste de l'exécution et retourne res_1. Noter que *yield* a programmé un *fullfill* sur p_a.
- (b) Quand cela se produit, le patient de p_a est exécuté. $yield_b$ retourne une promesse bloquée p_b à laquelle $bind_2$ ajoute le reste du traitement comme patient. $bind_2$ retourne une nouvelle

```
let skip = ready ()
let halt () = ready ()
let yield () = let p = blocked () in
               enqueue (fun () → fullfill p ()); p
let wait_start = blocked ()
let spawn t = wait_start >>= t; ()

exception Stop
let stop () = raise Stop
```

```
let start () =
  fullfill wait_start ();
  try
    while true do
      dequeue () ()
    done
  with Queue.Empty | Stop → ()
```

```
type α mvar = { mutable v : α option;
                mutable read : α t option;
                mutable write : (unit t × α) option }

let put_mvar out v =
  match out with
  | { v = Some v'; read = _; write = None } →
      let w = blocked () in out.write ← Some (w, v); w

  | { v = None; read = Some r; write = None } →
      out.read ← None; enqueue (fun () → fullfill r v); ready ()

  | { v = None; read = None; write = None } → out.v ← Some v; ready ()
```

FIGURE 7 – Monade de promesse (2) : ordonnanceur, MVars

promesse bloquée res_2 qui est *connect*ée à res_1. Comme res_2 est bloquée elle devient un lien sur res_1.

- (c) Quand le patient de p_b est exécuté, $yield_c$ retourne une promesse bloquée p_c, qui est liée à res_1 (puisque res_2 est elle même liée à res_1) par *connect*.
- (d) Enfin, quand p_c est comblée, res_1 devient prête.

4.6. Programmation par événements

La programmation par événements est un paradigme populaire pour mettre en œuvre la concurrence au niveau application. La bibliothèque OCamlNet [16] inclut un petit module `equeue` (env. 160 lignes) permettant la programmation par événements.

Pour utiliser `equeue`, un "système d'événements" doit d'abord être créé. Les handlers y sont enregistrés. Chaque événement est présenté successivement aux handlers jusqu'à ce que l'un d'eux l'accepte. Un handler peut accepter l'événement ou le refuser en lançant l'exception *Reject*. Une fois l'événement traité le handler reste en place sauf s'il lance l'exception *Terminate*.

Le principe de la réalisation est le suivant : la création d'un thread consiste à mettre en place un handler et à le déclencher par un événement approprié. Au cours d'une opération bloquante, le handler enregistre la continuation comme nouveau handler en attente de l'événement indiquant que le traitement peut continuer, puis lance *Terminate*. Un thread bloqué en lecture sur une MVar crée un nouvel événement, le place dans la MVar et enregistre sa continuation en attente de cet événement. Le prochain thread à écrire dans la MVar insérera l'événement ce qui permettra au premier thread de reprendre l'exécution. *yield* insère lui même l'événement sur lequel il place sa continuation en attente.

Nous ne montrons pas le code par manque de place mais faisons deux remarques. Tout d'abord cette réalisation a une faiblesse majeure liée à `equeue` : la file d'événements impose un type fixe pour les événements, les opérations sur les MVars perdent leur polymorphisme. Ensuite, il ne s'agit en fait que d'une variante du style trampoline où le système d'événements met en œuvre l'ordonnanceur.

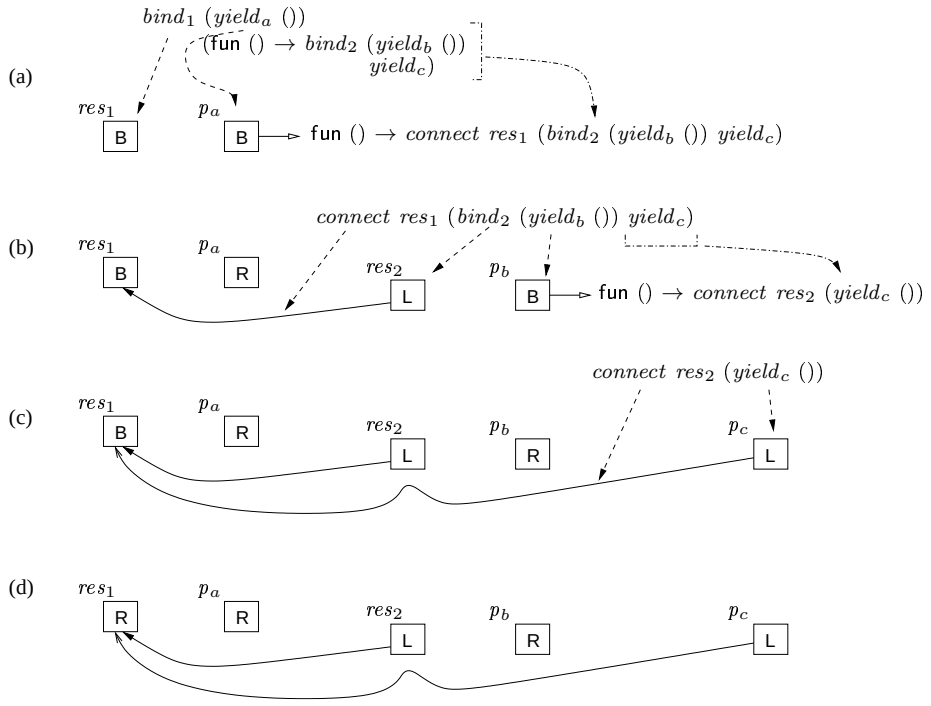


FIGURE 8 – Promesses : exécution de *yield3*

5. Performance

Nous avons utilisé les deux applications exemples pour comparer les performances de ces différentes mises en œuvres. Les temps d'exécution ont été mesurés à l'aide de la fonction *Unix.times*, l'usage mémoire donné par le champ *top_heap_words* fourni par la fonction *quick_stat* du module `Gc` de OCaml.

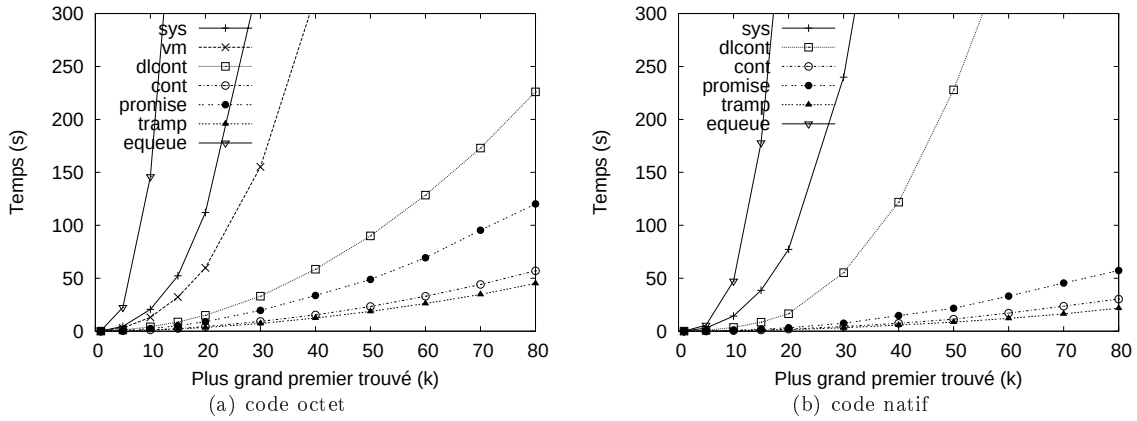
Les programmes ont été exécutés sur un PC avec processeur Intel Core 2 Duo à 2.33 GHz et 2 Go de mémoire, un noyau linux 2.6.26 pour architecture amd64. Les versions des logiciels sont OCaml 3.11.2, `caml-shift` d'août 2010, `equeue` 2.2.9.

Nos deux applications sont basées sur des threads très simples qui coopérent intensivement. L'intérêt de sieve est qu'il crée constamment de nouveaux threads. sorter requiert un nombre de threads qui peut devenir énorme (pour trier une liste de 3000 nombres, ce sont environ 4,5 millions de threads qui sont créés).

Dans la suite du texte et sur les graphiques, nous noterons `sys` la mise en œuvre utilisant les threads système, `vm` les threads de la machine virtuelle OCaml, `dlcont` les captures de continuation, `tramp` le style trampoline, `cont` la monade de continuation, `promise` la monade de promesse, et `equeue` les événements.

La figure 9 montre les temps d'exécution pour le crible. `equeue` est terriblement lent, bien pire que `sys`. Le problème est dans la réalisation du module `Equeue`. Comme nous l'avons dit, les événements sont présentés à chaque handler jusqu'à ce qu'un l'accepte, ce qui revient à faire de l'attente active sur les MVars. Un examen du code source montre également que l'ajout d'un handler ou d'un événement a un coût linéaire. Ce dernier point peut facilement être corrigé mais n'a pas d'impact radical sur les performances. `equeue` ne peut donc pas gérer un trop grand nombre de threads.

Les autres mises en œuvre légères sont toutes bien au dessous de `vm` et `sys`, `tramp` étant la meilleure

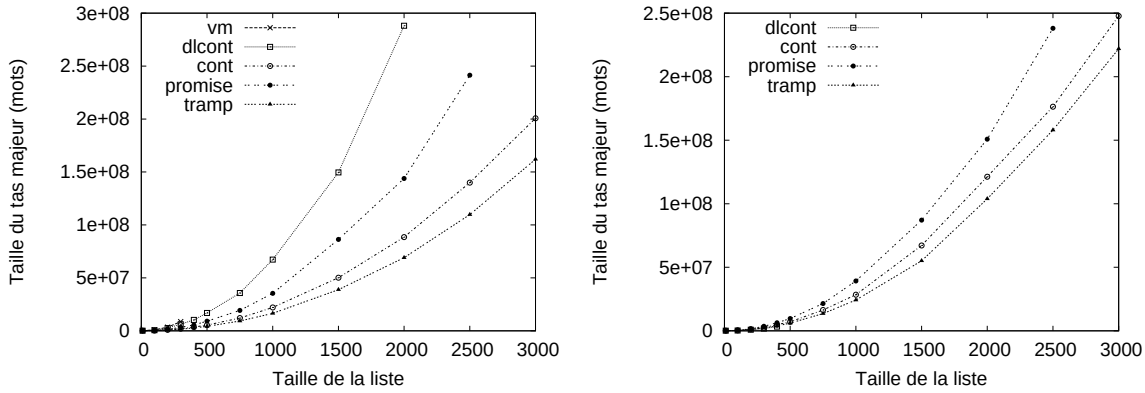


(a) code octet (b) code natif

FIGURE 9 – sieve, temps d'exécution

avec `cont` à peine au dessus. `promise` est deux fois plus lent, `dlcont` quatre fois et bien plus en code natif.

Les graphes pour l'usage mémoire du tri sont sur la figure 10. Nous ne montrons pas `sys` car il faudrait aussi mesurer la mémoire utilisée par le système d'exploitation lui-même. La aussi `tramp` et `cont` sont clairement les meilleurs, avec `promise` nettement au dessus. En code octet `dlcont` est confondu avec `vm`. En code natif, `dlcont` est trop lent pour trier au delà de 300 nombres. La page web de la bibliothèque `caml-shift` indique que les opérations sur les continuations sont bien plus coûteuses en code natif.



(a) code octet (b) code natif

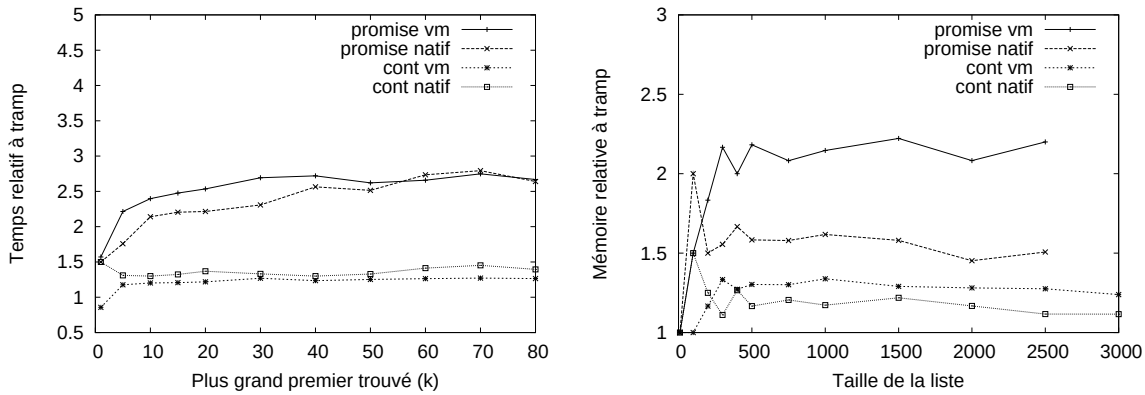
FIGURE 10 – sorter, usage mémoire

Comme nous l'avons vu, `tramp` et `cont` sont des réalisations extrêmement proches, tandis que `promise` repose sur le même cadre (monade) que `cont`. Nous nous intéressons donc de plus prêt à leur comparaison. La figure 11 montre le "sur-coût" de `cont` et `promise` sur `tramp`. Celui de `cont` est assez faible mais clairement apparent, aussi bien en temps qu'en mémoire. Cela peut paraître étonnant vu l'extrême similitude des deux réalisations. Seul l'opérateur >>= est un peu plus compliqué avec `cont` puisqu'il construit deux fermetures pour assurer le passage du paramètre continuation.

Le sur-coût de `promise` est moins surprenant (et plus important). Nous avons vu que la gestion d'un point de coopération impliquait un certain nombre d'opérations telle que création de promesse,

pattern matching, modification de référence. On note que le sur-coût en mémoire est nettement moins important en code natif.

Bien sûr nos deux applications sont assez extrêmes en ce que chaque thread fait très peu de travail entre chaque point de coopération, ce qui fait ressortir les différences de coût dans la gestion de ces points. C'était le but recherché mais on peut penser que pour beaucoup d'applications les différences que nous relevons ici seraient noyées dans les coûts de l'application elle même.



(a) temps sur le crible (b) mémoire sur le tri

FIGURE 11 – Sur-coût de `cont` et `promise` sur `tramp`

Nous terminons par deux remarques. Les continuations utilisées pour mettre en place un système de threads sont "à usage unique" ce qui peut se prêter à une mise en œuvre optimisée de l'opération de capture [1] et en une récupération mémoire plus efficace. Pour `equeue`, terminer un handler est relativement coûteux (lancé d'exception). Il semble qu'il ait été conçu avec l'idée que les handlers (relativement peu nombreux) restent actifs longtemps. L'usage que nous en faisons (très grand nombre de handlers, à usage unique) est à l'opposé de cette idée, ce qui peut expliquer ses mauvaises performances.

6. Conclusion et perspectives

Nous avons décrit, mis en œuvre et (un peu) évalué plusieurs façons de réaliser la concurrence légère en OCaml. Le code complet des mises en œuvre est disponible sur le web (`http://christophe.deleuze.free.fr/lwc/`). Le style direct implique de capturer les continuations, ce qui est relativement coûteux (mais bien moins que les threads système ou de la VM). Le style indirect impose au programmeur de faire apparaître explicitement les continuations.

Nous retenons que presque toutes les réalisations sont capables de gérer au moins des millions de threads. La seule exception est `equeue` qui ne semble pas conçu pour gérer la concurrence massive. De plus nous avons constaté que la programmation par événement n'est en fait essentiellement qu'une technique d'ordonnancement pour le style trampoline. Nous aurions donc pu la supprimer de notre panorama mais il nous semblait intéressant de faire apparaître ce point.

Concernant les performances le style trampoline est le plus léger, la monade de continuation n'étant que légèrement au dessus. La monade de promesse l'est plus sensiblement mais les différences restent limitées, en dessous d'un facteur trois. Par contre la capture de continuations est nettement plus coûteuse.

Les réalisations présentées ici sont des jouets, une bibliothèque réaliste devrait gérer au moins les entrées/sorties et les exceptions. `Lwt` est une telle bibliothèque basée sur la monade de promesse. Nous

développons actuellement une bibliothèque basée sur le style trampoline pour la concurrence légère en OCaml. Des applications réalistes telles qu'un serveur FTP et un résolveur DNS sont également développées.

```
open Lwc

let rec integers out i () =
  put_mvar out i;
  integers out (i + 1) ()

let rec output inp () =
  let v = take_mvar inp in
  if !print then Printf.printf "%i " v;
  if v < !last then output inp () else stop ()

let rec filter n inp out () =
  let v = take_mvar inp in
  if v mod n ≠ 0 then put_mvar out v;
  filter n inp out ()

let rec sift inp out () =
  let v = take_mvar inp in
  put_mvar out v;
  let mid = make_mvar () in
  spawn (filter v inp mid);
  sift mid out ()

let sieve () =
  let mi = make_mvar () in
  let mo = make_mvar () in
  spawn (integers mi 2);
  spawn (sift mi mo);
  spawn (output mo);
  start ()
```

(a) Style direct

```
open Lwc

let rec integers out i () =
  put_mvar out i >>= integers out (i + 1)

let rec output inp () =
  take_mvar inp >>= fun v →
  if !print then Printf.printf "%i " v;
  if v < !last then output inp () else (stop (); halt())

let rec filter n inp out () =
  take_mvar inp >>= fun v →
  (if v mod n ≠ 0 then put_mvar out v else skip) >>=
  filter n inp out

let rec sift inp out () =
  take_mvar inp >>= fun v →
  put_mvar out v >>= fun () →
  let mid = make_mvar () in
  spawn (filter v inp mid);
  sift mid out ()

let sieve () =
  let mi = make_mvar () in
  let mo = make_mvar () in
  spawn (integers mi 2);
  spawn (sift mi mo);
  spawn (output mo);
  start ()
```

(b) Style indirect

FIGURE 12 – Le programme du crible

Références

[1] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. In *In Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 99–107, 1996.

[2] Chung chieh Shan. Shift to control. In Olin Shivers and Oscar Waddell, editors, *Proceedings of the 5th workshop on Scheme and functional programming*, pages 99–107. Indiana University, 2004.

[3] Koen Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9(3):313–323, May 1999. Functionnal Pearls.

[4] D. P. Friedman. Applications of continuations. Invited Tutorial, Fifteenth Annual ACM Symposium on Principles of Programming Languages, January 1988.

[5] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In *International Conference on Functional Programming*, pages 18–27, 1999.

[6] Mark P. Jones and Paul Hudak. Implicit and explicit parallel programming in haskell. Technical Report YALEU/DCS/RR-982, Department of Computer Science, Yale University, New Haven, CT 06520-2158, 1993.

[7] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *Information processing*, pages 993–998, Toronto, August 1977.

[8] R. Kelsey, W. Clinger, and J. Rees (eds.). Revised^5 report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1), 1998.

[9] Oleg Kiselyov. How to remove a dynamic prompt: static and dynamic delimited continuation operators are equally expressible. Technical Report 611, Indiana University, March 2005.

[10] Oleg Kiselyov. Delimited control in OCaml, abstractly and concretely system description. Technical report, March 2010. Also on FLOPS 2010.

[11] Xavier Leroy. OCaml-callcc: call/cc for OCaml. OCaml library.

[12] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. *SIGPLAN Not.*, 23:260–267, June 1988.

[13] M. Douglas McIlroy. Coroutines. Internal report, Bell telephone laboratories, Murray Hill, New Jersey, May 1968.

[14] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction, Marktoberdof Summer School 2000*, pages 47–96. IOS Press, 2001.

[15] Amr Sabry, Chung chieh Shan, and Oleg Kiselyov. Native delimited continuations in (byte-code) OCaml. OCaml library, 2008.

[16] Gerd Stolpmann. Ocamlnet.

[17] Gerald Jay Sussman and Guy L. Steele. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, December 1998. Reprint from AI memo 349, December 1975.

[18] Jérôme Vouillon. Lwt: a cooperative thread library. In *ML '08: Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 3–12, New York, NY, USA, 2008. ACM.