

Concurrence et continuations en OCaml

Christophe Deleuze

LCIS – Laboratoire de Conception et d'Intégration des Systèmes
Grenoble-INP, Valence

JFLA 2012 – février 2012

Introduction

- concurrence
 - fils d'exécution
 - indépendance temporelle
 - par exemple, modèle réactif
- échelle limitée par le système d'exploitation
 - processus/thread
 - préemptif
 - **coûteux**
- niveau application
 - coopératif
- panorama de quelques techniques existantes

Plan

- un modèle minimal
- une application : le crible
- style direct
 - captures de continuation
- style indirect
 - trampoline, monades, événements
- performances
- conclusion

Un modèle minimal

- *spawn* – crée un nouveau thread
- *yield* – le thread suspend son exécution

mvar : variable mutable partagée (vide ou pleine)

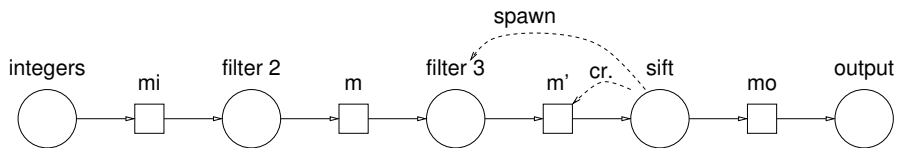
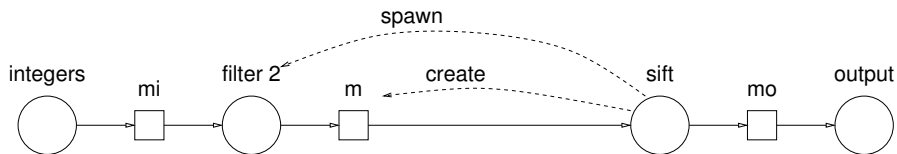
- *make_mvar* – crée une mvar vide
- *take_mvar* – retire la valeur de la mvar
- *put_mvar* – place une valeur dans la mvar

Une application : le crible

Chaîne de threads

- *integers* : générateur
 - produit les entiers à partir de 2
- *filter n* : bloque les multiples de n
- *sift* : “étendeur”
 - insère nouveau filtre pour chaque premier trouvé
- *output* : récepteur

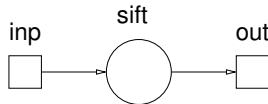
Une application : le crible



Style direct

ou “normal”

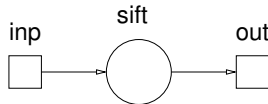
```
let rec sift inp out () =  
  let v = take_mvar inp in  
  put_mvar out v;  
  let mid = make_mvar () in  
  spawn (filter v inp mid);  
  sift mid out ()
```



Style direct

ou “normal”

```
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 ()
```



Les opérations bloquantes se présentent comme des fonctions normales

```
val take_mvar :  $\alpha$  mvar  $\rightarrow$   $\alpha$ 
```

```
val put_mvar :  $\alpha$  mvar  $\rightarrow$   $\alpha$   $\rightarrow$  unit
```

```
val yield : unit  $\rightarrow$  unit
```


Opérations bloquantes

Comment les mettre en œuvre ?

```
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 ()
```

Opérations bloquantes

Comment les mettre en œuvre ?

```
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 ()
```

continuation = “ce qui reste à exécuter” = le contexte = la pile

Opérations bloquantes

Comment les mettre en œuvre ?

```
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 ()
```

continuation = “ce qui reste à exécuter” = le contexte = la pile

Opérations bloquantes

Comment les mettre en œuvre ?

```
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 ()
```

continuation = “ce qui reste à exécuter” = le contexte = la pile

- capture (call/cc en Scheme) : **réifie** la continuation
- bibliothèque caml-shift : continuations délimitées
 - préfixe de la continuation
 - délimitée par un *prompt* dans la pile

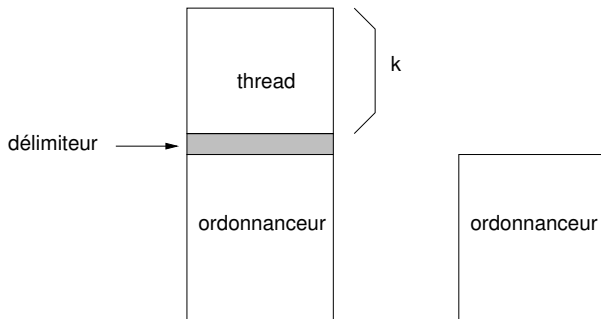
Capture de continuation

val *shift0* : *prompt* → ((α → *unit*) → *unit*) → α

let *yield* () = *shift0 prpt* (fun *k* → *enquee k*)

shift0 :

- réifie le fragment de pile et le passe à son deuxième argument
- retire le fragment, le prompt et retourne



Cas des mvars

```
type  $\alpha$  t =  $\alpha \rightarrow \text{unit}$ 
```

```
type  $\alpha$  mvar = { mutable v :  $\alpha$  option;  
                 mutable read :  $\alpha$  t option;  
                 mutable write : (unit t  $\times$   $\alpha$ ) option }
```

```
let take_mvar inp =
```

```
  match inp with
```

```
  | { v = Some v; read = None; write = None }  $\rightarrow$   
    inp.v  $\leftarrow$  None; v
```

```
  | { v = Some v; read = None; write = Some(kp, v') }  $\rightarrow$   
    inp.v  $\leftarrow$  Some v'; inp.write  $\leftarrow$  None; enqueue kp; v
```

```
  | { v = None; read = None; write = _ }  $\rightarrow$   
    shift0 prpt (fun k  $\rightarrow$  inp.read  $\leftarrow$  Some k)
```

Style indirect

- continuations **explicit**es (fermetures)
- ($\gg=$) “opérateur de composition séquentielle”

```
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 ()
```

Style indirect

- continuations **explicit**es (fermetures)
- ($\gg=$) “opérateur de composition séquentielle”

```
let rec sift inp out () =  
  take_mvar inp  $\gg=$  fun v →  
  put_mvar out v  $\gg=$  fun () →  
  let mid = make_mvar () in  
  spawn (filter v inp mid);  
  sift mid out ()
```


Style indirect

- continuations **explicit**es (fermetures)
- ($\gg=$) “opérateur de composition séquentielle”

```
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 ()
```

Trampoline

- une fonction bloquante prend sa continuation en paramètre
- variante du style *passage de continuation* (CPS)

let ($\gg=$) $f k = f k$

val *yield* : (*unit* → *unit*) → *unit*

val *take_mvar* : α *mvar* → (α → *unit*) → *unit*

val *put_mvar* : α *mvar* → α → (*unit* → *unit*) → *unit*

Trampoline

- une fonction bloquante prend sa continuation en paramètre
- variante du style *passage de continuation* (CPS)

let ($\gg=$) $f k = f k$

val *yield* : (*unit* \rightarrow *unit*) \rightarrow *unit*

val *take_mvar* : α *mvar* \rightarrow ($\alpha \rightarrow$ *unit*) \rightarrow *unit*

val *put_mvar* : α *mvar* \rightarrow $\alpha \rightarrow$ (*unit* \rightarrow *unit*) \rightarrow *unit*

Trampoline

let *yield k = enqueue k*

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(kp, v')* } →
 inp.v ← Some v'; inp.write ← None; enqueue kp; k v

 | { *v = None; read = None; write = _* } →
 inp.read ← Some(k)

Monade de continuation

type αt

val *return* : $\alpha \rightarrow \alpha t$

val ($\gg=$) : $\alpha t \rightarrow (\alpha \rightarrow \beta t) \rightarrow \beta t$

Monade de continuation

type αt

val *return* : $\alpha \rightarrow \alpha t$

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

type $\alpha t = (\alpha \rightarrow \text{unit}) \rightarrow \text{unit}$

val *put_mvar* : $\alpha \text{ mvar} \rightarrow \alpha \rightarrow \text{unit } t$

val *take_mvar* : $\alpha \text{ mvar} \rightarrow \alpha t$

val *yield* : $\text{unit} \rightarrow \text{unit } t$

Monade de continuation

type αt

val *return* : $\alpha \rightarrow \alpha t$

val ($\gg=$) : $\alpha t \rightarrow (\alpha \rightarrow \beta t) \rightarrow \beta t$

type $\alpha t = (\alpha \rightarrow \text{unit}) \rightarrow \text{unit}$

val *put_mvar* : $\alpha \text{ mvar} \rightarrow \alpha \rightarrow \text{unit } t$

val *take_mvar* : $\alpha \text{ mvar} \rightarrow \alpha t$

val *yield* : $\text{unit} \rightarrow \text{unit } t$

($\gg=$) est un peu plus compliqué...

val ($\gg=$) : $((\alpha \rightarrow \text{unit}) \rightarrow \text{unit}) \rightarrow$
 $(\alpha \rightarrow (\beta \rightarrow \text{unit}) \rightarrow \text{unit}) \rightarrow$
 $(\beta \rightarrow \text{unit}) \rightarrow \text{unit}$

let ($\gg=$) $f k' = \text{fun } k \rightarrow f (\text{fun } r \rightarrow k' r k)$

Monade de promesse

- les mêmes signatures... mais αt ne représente pas la même chose
- **promesse** : valeur qui peut être utilisée pour accéder plus tard à une valeur non encore disponible (*proxy*).
- opération bloquante retourne une promesse
 - valeur mutable *prête* ou *bloquée*

$$\alpha t \quad \rightarrow \quad (\alpha \rightarrow \beta t) \rightarrow \beta t$$

`take_mvar inp >>= fun v → ...`

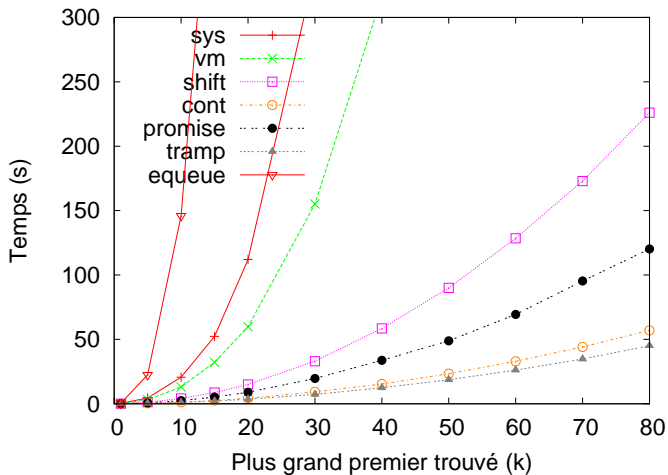
- `take_mvar` retourne promesse p_a
- `>>=` : si p_a prête, “ouvre” et passe à la continuation qui produit p_b
- sinon retourne nouv. promesse bloquée *res* et “associe” à p_a :
 - exécution de la continuation, qui produit p_b
 - liaison de *res* avec p_b

Programmation événementielle

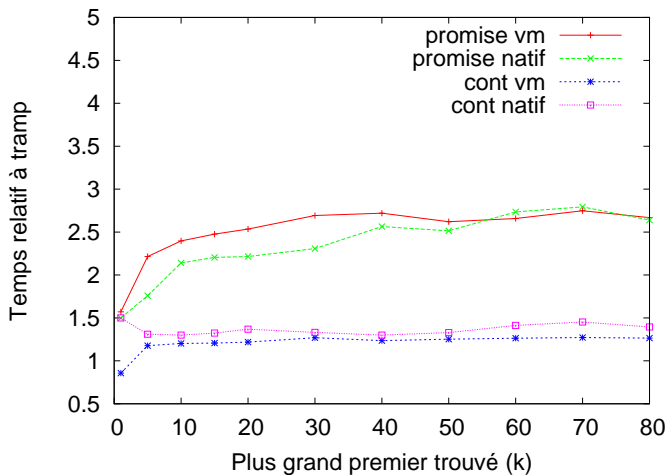
- classique en impératif... “handler” ou “callback”
- proche de la notion de continuation !

- module `queue` de OCamlNet
- très lent (pas adapté)

Temps sur le crible (code octet)



Comparaison de tramp, cont et promise



Conclusion

- continuation
- style direct coûteux
 - lié à la capture
- indirect
 - peut gérer des millions de threads (sauf equeue)
- “vraies” réalisations
 - Lwt : monade de promesse
 - Muthreads : trampoline
 - async de JaneStreet (?)