Concurrence légère en OCaml : muthreads

Christophe Deleuze

Concurrence légère en OCaml : muthreads

Christophe 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 muthreads, une bibliothèque pour la concurrence légère en OCaml. Elle repose sur une boucle événementielle mais le paradigme fonctionnel permet d'atténuer dans une large mesure les inconvénients traditionnels de la programmation par événements, en introduisant la notion de *thread*. La bibliothèque ne propose qu'un ensemble réduit d'opérations (dont les MVars comme moyen privilégié de communication entre threads) qui peuvent se combiner pour construire des fonctionnalités complexes. La mise en œuvre des MVars peut être étendue pour permettre la communication entre threads s'exécutant dans des processus système différents, ce qui permet d'envisager l'exploitation du parallélisme matériel.

1. Introduction : threads et événements

Traditionnellement, les applications concurrentes ont été développées suivant deux modèles : threads et événements. Les threads, bien que permettant une programmation plus naturelle, "directe" ont pour inconvénient un coût significatif, aussi bien en mémoire monopolisée par chaque thread qu'en temps de commutation de contexte. C'est la raison pour laquelle l'approche à événements a souvent été mise en avant dans le cas où une concurrence massive ou légère était nécessaire. Cependant elle nécessite d'utiliser un ordonnancement coopératif et de découper les traitements en une série de *callbacks*, rendant le flot de contrôle de chaque traitement peu apparent [11, 15].

L'approche à événements connaît un regain d'intérêt dans le "grand public" depuis quelques années, notamment avec le langage JavaScript pour les applications web [6, 10]. Outre que les développeurs web n'ont pas le choix (JavaScript ne propose pas de threads), le fait que JavaScript considère les fonctions comme des valeurs de première classe simplifie la mise en œuvre de l'approche à événements.

Les langages fonctionnels comme OCaml se prêtent particulièrement bien à ce type d'approche, qui a déjà été explorée sous diverses formes comme le style trampoline [5], la monade de continuation [1] ou la monade de promesse.

Toutes ces formes sont des variantes de ce que l'on peut appeler le *style indirect*, illustré dans le code ci-dessous, où `yield` et `sleep` introduisent des points de coopération (le traitement se suspend pour laisser à d'autres l'opportunité de s'exécuter) :

```
let t () =
    print_string "je passe mon tour"; yield >>= fun () ->
    print_string "je dors un peu"; sleep 2. >>= fun () ->
    print_string "je me réveille"; ...
```

L'opérateur `>>=` représente la mise en séquence. Le style indirect assure que la pile d'exécution est vide à chaque opération bloquante (point de coopération), faisant apparaître la suite du traitement (ou

sa *continuation*) sous la forme d'une fonction qui peut alors être stockée pour relancer le traitement quand l'opération bloquante devient possible [3].

Muthreads est une bibliothèque OCaml qui fournit un modèle de programmation par threads rédigées en style trampoline, que l'on peut voir comme le résultat d'une *conversion CPS partielle* [14] : chaque opération bloquante prend en paramètre une fonction de continuation, représentant les traitements à réaliser quand l'opération aura été effectuée. La bibliothèque repose en interne sur une approche à événements et offre donc une concurrence extrêmement légère. C'est un logiciel de taille modeste (environ 1000 loc) avec un objectif de simplicité, aussi bien dans l'utilisation (choix des fonctionnalités primitives) que dans la réalisation.

Quelques applications utilisant la bibliothèque ont été réalisées, avec en particulier deux applications "réelles" : un serveur FTP et un résolveur DNS. Le résolveur DNS est notamment utilisé quotidiennement sur une machine personnelle depuis plus d'un an.

Dans la suite, nous commençons par donner un aperçu des fonctionnalités proposées par la bibliothèque et de la façon de les utiliser, puis décrivons leur mise en œuvre. Nous discutons ensuite le choix de ces fonctionnalités, et l'extension du modèle pour le support du parallélisme. Enfin, nous évoquons les autres bibliothèques comparables, avant de conclure et de donner quelques perspectives.

2. Vue d'ensemble des fonctionnalités

La bibliothèque fournit essentiellement :

- un mécanisme de communication et synchronisation entre les threads (les MVars),
- la gestion des entrées/sorties (calquée sur l'API système d'Unix),
- la gestion des exceptions,
- la temporisation de traitements,
- l'attente simultanée sur une MVar et une opération d'entrée/sortie.

Nous donnons dans cette partie un aperçu de ces fonctionnalités à travers des extraits (parfois simplifiés pour la clarté de l'exposé) du code du serveur FTP.

Concurrence en FTP Dans une communication FTP, une première connexion TCP (dite connexion de commande) est établie. Le client l'utilise pour émettre des commandes auxquelles le serveur répond. Ces commandes correspondent soit à des opérations simples (changement ou affichage du répertoire courant par exemple) soit à la préparation d'un transfert. Une deuxième connexion, dite de données, est créée pour chaque transfert de fichier ou listage de répertoire. Dans la spécification FTP [13], on considère qu'un processus PI *(protocol interpreter)* gère la connexion de commande, alors qu'un processus DTP *(data transfer process)* gère la connexion de données. La séparation en deux processus fait sens car des commandes [1] peuvent être émises (et doivent être traitées) sur la connexion de commande pendant un transfert sur la connexion de données.

Il apparaît donc naturel de structurer le programme en un thread principal attendant les demandes de connexion de commande et créant (avec `spawn`) un thread `pi` pour chacune d'elle, chaque thread `pi` gérant sa connexion et créant un thread `dtp` chaque fois que nécessaire, le temps d'un transfert.

```
let rec main s =
  accept s >>= fun (inp, _) ->
  spawn (pi inp);
  main s
```

Le code de `pi` est organisé comme un ensemble de fonctions mutuellement récursives, chacune mettant en œuvre les traitements correspondant à un état de la communication et choisissant sa

1. En fait seulement un petit sous-ensemble des commandes.

continuation parmi les autres fonctions. Cela reproduit la structure d'un automate tout en mettant en évidence le déroulement séquentiel "normal" des opérations :

```
let pi skt () =
  ...
  let rec greeting () = ...
  and login () =
    recv skt >>= fun m -> match m with
    | USER user ->
        rep skt 331  >>= fun () ->
        recv skt >>= fun m -> (match m with
        | PASS pass ->
            do_login user pass >>= fun (dir, id) ->
            if id <> -1 then
              rep skt 230 >>= inter
            else
              rep skt 530 >>= login
        | END -> do_end skt
        | _   -> rep skt 530 >>= login)

    | QUIT -> rep skt 221 >>= fun () -> close skt; terminate ()
    | END -> do_end skt
    | _ -> rep skt 503 >>= login

  and inter () =  ...
  and preparing_transfer () = ...
  and controlling_transfer () = ...
  and ...
  and update_stats dtr () = ...
  in
  greeting ()
```

On montre ici en détail la phase `login` qui appelle :
- `inter` pour la phase "d'interaction" (séquence de requêtes/réponses simples) si l'authentification a été réalisée correctement,
- `login` récursivement si l'authentification a échoué ou n'a pas été effectuée correctement,
- `do_end` si la connexion doit être fermée.

On note l'utilisation de fonctions "bloquantes" (`recv`, `rep`, `do_login`) et l'usage de la fonction `terminate` pour indiquer la fin du thread.

Entrées/sorties Pour les entrées/sorties sur les sockets chaque opération bloquante est remplacée par une version "trampoline" prenant la continuation à utiliser quand l'opération aura été effectuée. Le code ci-dessus utilise les fonctions `recv` pour recevoir et décoder un message, et `rep` pour encoder et envoyer une réponse. `recv` peut par exemple être réalisée de la façon suivante (`parse_cmd` décode une commande FTP à partir d'une chaîne d'octets) :

```
let recv skt k =
  let s = String.create 512 in
  read skt s 0 512 >>= fun l ->
  k (parse_cmd s l)
```

MVars Les threads peuvent échanger des données par des références partagées mais aussi et surtout à l'aide des MVars. Introduites en Haskell [12], les MVars sont des variables mutables partagées qui permettent la communication et la synchronisation entre threads. Une MVar peut contenir une valeur ou être vide. On définit classiquement les trois opérations suivantes :

- `make_mvar` crée une nouvelle MVar vide.
- `take_mvar` retire la valeur d'une MVar, bloque si elle est vide.
- `put_mvar` place une valeur dans une MVar, bloque si elle est pleine.

Les MVars sont conçues pour la communication entre deux threads uniquement, on considère donc que pour chaque MVar un seul thread souhaite écrire et un seul thread souhaite lire à un moment donné.

Un thread DTP et son PI doivent pouvoir communiquer dans les deux sens. D'une part le DTP doit informer le PI de la réponse à transmettre quand il a terminé (correctement ou suite à une erreur). D'autre part le PI doit pouvoir interrompre le DTP s'il reçoit une commande `ABORT` ou si la connexion de commande est fermée. Dans cet extrait de la fonction `preparing_transfert`, le PI crée deux MVars `res` et `abt` qu'il passe en paramètres au DTP.

```
let res = make_mvar () in
let abt = make_mvar ()
in
spawn (dtp_active (!uid,!cdir) !typ dtr ip p abt res);
rep skt 150 >>=
controlling_transfer dtr abt res
```

Si une commande `ABORT` est reçue (voir la fonction `loop` dans le paragraphe suivant), le PI informe le DTP en écrivant dans `abt`, puis récupère son résultat dans `res`. [2]

MVar et entrées/sorties Pendant un transfert, le thread PI attend le résultat fourni par le DTP sur la MVar `res`, mais il doit également pouvoir traiter quelques commandes sur la connexion de commande. `take_or_read` permet d'attendre à la fois sur une MVar et une socket. Les arguments sont ceux des deux possibilités, avec une continuation pour chaque. Ici on définit la fonction `take_or_recv` qui retire la valeur de la MVar et la passe à `kt`, ou lit un message sur la socket `skt`, le décode et le passe à `kr`.

```
let take_or_recv mv skt kt kr =
  let s = String.create 512 in
  take_or_read mv skt s 0 512
    kt
    (fun l -> kr (parse_cmd s l))
```

Pendant qu'un transfert est géré par le DTP, le PI exécute du code de la forme suivante :

```
  let rec loop () =
    take_or_recv res skt
      (fun r -> rep skt r >>= update_stats dtr)      (* DTP is done  *)
      (fun m ->                                      (* cmd received *)
        match m with
        | ABORT  ->
            put_mvar abt () >>= fun () ->
```

2. Deux réponses sont envoyées, une pour la fin de l'opération sur la connexion de données, une pour acquitter la commande `ABORT`.

```
            take_mvar res   >>= fun r ->
            rep skt r       >>= fun () -> (* from DTP *)
            rep skt 226     >>=           (* from PI: successful abort *)
            update_stats dtr
        | ...
  in
  loop ()
```

La même fonctionnalité est utilisée dans l'autre sens : le DTP est occupé avec des opérations de lecture ou écriture sur sa socket, mais doit aussi être à l'écoute de la MVar `abt`.

Exceptions La construction `try/with` de OCaml ne fait généralement pas ce que l'on voudrait quand on l'applique à un thread. Dans l'exemple ci-dessous le `try/with` protège le `do_open` et pas les traitements représentés par `do_upload` :

```
try
  let f = do_open () in
  do_upload do_read skt f >>= fun r ->
  close_out f; k r
with _ -> k 550        (* if do_open has raised an exception *)
```

En effet, quand le thread `do_upload` se suspend pour la première fois, la fonction retourne et le contrôle sort du `try/with`[3]. La fonction `trywith` fournit le fonctionnement attendu : protéger une séquence d'opérations comprenant potentiellement des opérations bloquantes. Pour cela elle introduit un contexte de récupération des exceptions. Le premier argument est l'opération à protéger, le deuxième le traite-exception et le troisième la continuation à exécuter après que l'opération ait terminé ou qu'une exception ait été attrapée et traitée. Le traite-exception peut accepter toute exception ou seulement un type particulier. Si l'exception est refusée la pile de contexte est déroulée à la recherche d'un autre contexte de récupération des exceptions. Une exception non rattrapée entraîne la terminaison du thread.

Dans l'exemple ci-dessous toute exception lancée durant l'exécution de l'opération `dtp_switch` (qui est une fonction de la forme `fun k -> ...` et doit passer à sa continuation `k` un résultat numérique) sera attrapée et la valeur 425 (correspondant à un code d'erreur) sera passée à la continuation. Si seules les exceptions `Failed` devaient être attrapées, le traite exception aurait été écrit `(fun Failed k -> k 425)`.

```
trywith
  (dtp_switch user typ dtr s' abt cnt)
  (fun _ k -> k 425)
  (fun r -> put_mvar res r >>= fun () -> close s'; terminate ())
```

Temporisations Une temporisation permet d'annuler[4] une opération qui prend trop de temps. En plus du temps on fournit l'opération à temporiser ainsi que deux continuations à exécuter respectivement si le temporisateur expire et si l'opération termine. Nous redéfinissons ici la fonction `recv` introduite plus haut pour permettre au PI d'attendre une commande tout en fermant la connexion si aucune commande n'a été reçue pendant le temps `control_idle`.

3. Et donc, les opérations de `do_upload` jusqu'au premier point de coopération sont en fait protégées.
4. Par annuler on entend interrompre mais pas défaire les éventuels effets partiels de l'opération.

```
let recv skt k =
  let s = String.create 512 in
  timeout control_idle
    (read skt s 0 512)              (* timed operation *)
    (fun () ->                      (* timer expired   *)
      send skt 421 "Timeout: closing control connection" >>= fun () ->
      close skt; terminate ())
    (fun l -> k (parse_cmd s l))    (* read operation completed *)
```

3. Réalisation

Typage Une première version de la bibliothèque utilisait un typage "naïf" où la partie "continuation" d'un thread avait le type `unit -> unit`. L'expérience a montré que certaines erreurs stupides peuvent alors être facilement commises (et difficilement détectées !) :

- l'oubli du cas `else` d'une alternative,
- l'oubli d'utiliser la continuation.

```
let miss_else m =
   take_mvar m >>= fun v -> if v>0 then begin print_int v; terminate () end

let miss_k m k = take_mvar m >>= fun v -> print_int v
```

Ce type d'erreur est problématique car, comme nous le verrons plus loin, l'ordonnanceur doit savoir quand une fonction/thread retourne s'il doit dépiler un élément de contexte pour la relancer (par exemple une opération soumise à une temporisation a terminé) ou pas.

La bibliothèque définit un type abstrait `t` (dont nous verrons la définition concrète plus tard) retourné par la fonction `terminate` (provoquant la terminaison du thread) et par les primitives bloquantes. L'opérateur `>>=` permet de combiner un fragment en style trampoline (prenant une continuation de type `unit -> t` et retournant un `t`) avec une telle continuation.

```
type t
val (>>=) : (('a -> t) -> t) -> ('a -> t) -> t
val terminate : unit -> t
type 'a mvar
val put_mvar : 'a mvar -> 'a -> (unit -> t) -> t
val take_mvar : 'a mvar -> ('a -> t) -> t
```

Le type `t` étant abstrait il devient impossible d'obtenir un thread (*ie* une fonction retournant un `t`) autrement qu'en combinant par `>>=` les opérations bloquantes fournies par la bibliothèque. Ainsi la fonction `recv` définie précédemment a le type suivant (où `Data.cmd` est le type des commandes FTP) :

```
val recv : Unix.file_descr -> (Data.cmd -> Muthr.t) -> Muthr.t
```

Le typage ne prévient cependant pas toutes les erreurs, en particulier l'usage inadéquat du séquencement "classique" (avec le point-virgule). Ainsi, si une erreur est détectée ici

```
let rejected m = take_mvar m >>= fun v ->
                 print_int v; terminate (); print_string "toto"
```

c'est uniquement à cause de la valeur de retour, et ceci ne provoque qu'un avertissement :

```
# let accepted m = take_mvar m >>= fun v ->
                   print_int v; terminate (); print_string "toto"; terminate ()
                                ^^^^^^^^^^^^
Warning 10: this expression should have type unit.
```

Représentation des threads Essentiellement, un thread est composé d'une fonction (retournant un `t`) représentant sa continuation. Cependant, pour mettre en œuvre la gestion des exceptions et la temporisation des opérations, il faut associer à chaque thread quelques informations supplémentaires :

- une référence `pending_io` donnant l'état du thread concernant les opérations d'entrées/sorties, pouvant être :
  - actuellement bloqué en lecture sur la socket `skt`,
  - actuellement bloqué en écriture sur la socket `skt`,
  - pas actuellement bloqué sur une opération d'entrée/sortie.
- une pile de contexte indiquant les temporisations et traite-exceptions actifs.

```
type pending_io =
  | FdRead  of Unix.file_descr
  | FdWrite of Unix.file_descr
  | Nofd
```

```
type ctxt_elt =
  | Timeout of bool ref * float * (unit -> t)
  | TryWith of (exn -> t) * (unit -> t)
  | Seq     of (unit -> t)
and context = ctxt_elt list
```

Dans la suite nous dirons souvent simplement "continuation" pour parler de ce qui permet d'exécuter la suite du thread, comprenant donc le contexte et la référence `pending_io`.

Ordonnancement Un seul thread (au mieux) s'exécutant à tout instant, les autres threads se trouvent dans :

- une file `runq` traitée en FIFO par l'ordonnanceur,
- une liste `sleepers` (triée par date de réveil) pour les threads bloqués dans l'opération `sleep`,
- une structure de données `readers`/`writers` pour les opérations d'entrée/sortie en attente,
- un thread bloqué sur une MVar est stocké dans la MVar elle même.

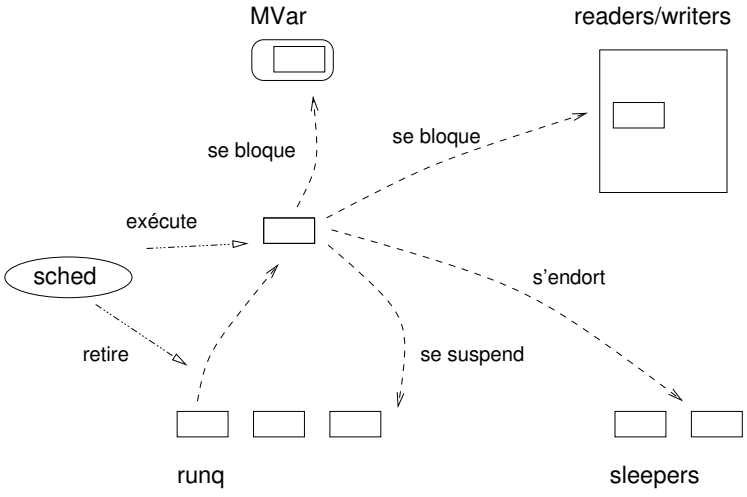

FIGURE 1 – Exécution d'un thread

L'ordonnanceur (`sched`) est une simple boucle infinie qui défile un à un les threads présents dans `runq` et appelle la fonction `run` qui met en place le contexte et `pending_io` et exécute la fonction en

attrapant les exceptions. Comme le montre la figure 1, c'est la fonction elle même, par les appels aux primitives de la bibliothèque, qui se replace dans `runq` ou se place dans les structures de données pour attendre la réalisation d'une opération bloquante.

```
let run (lio, c, f) =
  io := lio;
  ctxt := c;
  try f () with e -> trycatch e c lio

type t = Done | Susp | Pop
```

Le type abstrait `t` évoqué précédemment permet au thread d'indiquer à l'ordonnanceur son état actuel : `Done` si le thread est terminé (seule la fonction `terminate` retourne cette valeur), `Susp` si le thread s'est occupé de son propre sort (cas des opérations suspensives évoquées ci-dessus), ou `Pop` si le thread requiert une action de l'ordonnanceur pour poursuivre son exécution dans le contexte sous-jacent (cas des exceptions que nous discutons plus loin).

Entrées/sorties Les sockets sont placées en mode non bloquant. Quand une opération ne peut être effectuée immédiatement la continuation est enregistrée dans une structure de donnée `readers/writers` qui l'associe aux descripteur de fichier et type d'opération demandée. Cette structure permet de constituer les deux listes de descripteurs de fichiers pour lesquels on veut réaliser respectivement une opération de type lecture et écriture.

La file `runq` contient initialement un thread `check_io` qui est exécuté régulièrement et passe ces listes à la primitive système `select` [5]. Quand elle retourne celle-ci fournit les listes de descripteurs pour lesquels une opération respectivement de lecture et d'écriture est possible. Ces listes permettent pour chaque thread concerné de retrouver dans `readers/writers` l'opération précise et la continuation associée et de replacer le thread dans `runq` pour qu'il réalise effectivement l'opération.

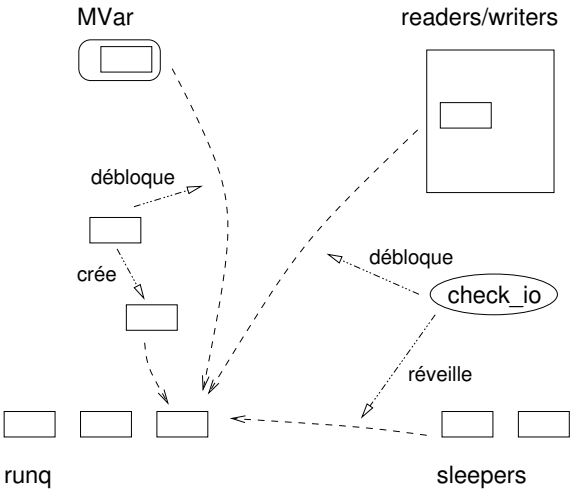

FIGURE 2 – Débloquage d'un thread

Comme montré sur la figure 2, le thread `check_io` gère aussi l'expiration des temporisations en replaçant dans `runq` les threads qui doivent être réveillés. Si `runq` est vide, la temporisation du

5. Sur le principe, ou pourrait facilement remplacer `select` par une alternative plus performance, comme `epoll` sous Linux [4].

`select` est mise conformément au premier thread à réveiller (attente passive) sinon le `select` est non bloquant. Un thread bloqué sur une MVar est (le cas échéant) débloqué par le thread effectuant l'opération attendue.

Gestion des exceptions L'exécution de `trywith f h k` consiste à programmer l'exécution de `f'` dans un contexte augmenté d'un `TryWith(h', k')`. `f'`, `h'` et `k'` permettent de résoudre une petite difficulté de typage : `f` (ou `h`) doit passer un résultat de type quelconque `'a` à `k`, mais l'élément de contexte `TryWith`, stocké dans la liste du contexte, ne peut pas contenir un composant polymorphe. Le passage se fait donc par l'intermédiaire d'une référence partagée.

```
let trywith f h k =                ((’a->t)->t) -> (exn->(’a->t)->t) -> (’a->t)-> t
  let res = ref None
  in
  let f’ () = f   (fun r -> res := Some r; Pop) in
  let h’ e  = h e (fun r -> res := Some r; Pop) in
  let k’ () = match !res with Some r -> k r
  in
  push_context (TryWith(h’, k’)) f’;
  Susp
```

Si `f'` se termine normalement (avec la valeur finale `Pop`), l'ordonnanceur dépile l'élément de contexte et exécute la continuation `k'` dans le contexte initial. Si une exception est lancée, elle est rattrapée par `run` et passée avec le contexte à la fonction `trycatch` qui déroule la pile de contexte juqu'à trouver un élément de la forme `TryWith(h, k)`. Elle encapsule alors `h` dans `tryh` qui permet, si le traite-exception refuse l'exception (une exception `Match_failure` est lancée), de relancer l'exception afin de poursuivre l'exploration de la pile. Le thread est terminé si le fond de la pile est atteint.

`tryh` sera exécuté dans un contexte `Seq k`, indiquant à l'ordonnanceur de poursuivre avec l'exécution de `k` quand `tryh` se termine (avec la valeur finale `Pop`).

```
let rec trycatch e cs lio =
  match cs with
  | Timeout(s,_,_) :: t -> s:=true; trycatch e t lio    (* kill timer *)
  | Seq _          :: t -> trycatch e t lio
  | TryWith(h, k)::t ->
      let tryh () =
         try h e with Match_failure _ -> raise e
      in
      enqueue (lio, (Seq k)::t, tryh);
      Susp
  | [] ->
      let msg = ("Warning: uncaught exception, thread killed! "
             ^ (Printexc.to_string e) ^ "\n")
      in
      output_string stderr msg; flush stderr;
      Done
```

Temporisations La fonction `timeout` a le type suivant, où les arguments sont le délai `d`, l'opération `op` à temporiser (prenant une continuation de type `'a -> t`), la continuation `kexp` à utiliser si le temporisateur expire et la continuation `k` à utiliser si l'opération est réalisée complètement :

```
val timeout : float -> (('a -> t) -> t) -> (unit -> t) -> ('a -> t) -> t
                d            op                kexp          k
```

Cette fonction :

1. empile un nouvel élément de contexte `Timeout(shot, d, k)` où `shot` est une référence sur un booléen faux, et `d` et `k` les arguments mentionnés ci-dessus (plus précisément, la technique montrée pour le `TryWith` est utilisée pour ne pas faire apparaître le type `'a` de `k` dans l'élément de contexte),
2. place dans `runq` l'opération `op` pour exécution dans ce nouveau contexte,
3. place dans `sleepers` un nouveau thread `timer` à exécuter dans le contexte initial après le temps `d`.

La référence `shot` sera mise à vrai par "le premier à tirer". À son réveil, `timer` commencera par examiner la valeur de `shot`. Si celui-ci est toujours faux, l'opération n'est pas terminée : `timer` met `shot` à vrai, nettoie le `pending_io` (si le thread temporisé était bloqué sur une opération d'entrée/sortie, il est alors détruit) et poursuit avec l'exécution de `kexp`. Si `shot` est vrai, `timer` termine immédiatement.

Quand le thread `op` rend la main à l'ordonnanceur, la valeur retournée indique si l'opération est terminée. Si c'est le cas (valeur `Pop`) l'ordonnanceur met `shot` à vrai ce qui annulera l'action du thread `timer`. De plus, à chaque fois qu'il s'apprête à exécuter un thread défilé de `runq` l'ordonnanceur vérifie si son contexte contient un élément `Timeout` dont le composant `shot` est vrai et ne l'exécute pas (le détruisant donc) si c'est le cas.

Opérations combinées MVar et entrées/sorties La réalisation du serveur FTP a montré la nécessité de pouvoir attendre simultanément deux événements, sur une MVar et une socket. Nous décrivons brièvement la réalisation de `take_or_read`.

```
take_or_read mv fd s ofs len ktake kread
```

L'idée est simplement de réagir normalement aux deux événements mais en faisant en sorte que la première continuation à s'exécuter annule la deuxième.

Un thread `mvar_wait` est placé en attente de l'opération `take` dans la MVar, de façon à être placé en file d'exécution dès qu'une valeur est placée dans la MVar. Son rôle est de nettoyer l'entrée de `readers` pour détruire la continuation `kread` puis d'exécuter `ktake`. La continuation placée dans `readers` commence elle par effacer le thread stocké dans la MVar avant d'exécuter `kread`.

En fait les choses sont un peu plus compliquées car les deux opérations peuvent être déclenchées "simultanément" : une des deux opérations est déclenchée (c'est à dire placée dans `runq`) puis la deuxième l'est également avant que la première soit effectivement exécutée. Pour cela l'opération proprement dite n'est pas effectuée immédiatement mais le sera à la prochaine exécution du thread.

4. Expressivité

Les fonctions primitives fournies par la bibliothèque sont volontairement en nombre réduit : création de thread, MVars, entrée/sortie, MVar et entrée/sortie combinées, temporisation, gestion des exceptions.

Il est intéressant que ces primitives aient été suffisantes pour développer de "vraies" applications. La plus compliquée de ces primitives, la gestion combinée MVar et entrées/sorties n'a d'ailleurs été utilisée que dans le serveur FTP. En particulier on aurait pu penser que le modèle de communication fourni pour les MVars est en l'état très limité, mais il peut être étendu par combinaison des primitives.

Avec les MVars Un premier exemple est le test non bloquant de l'état d'une MVar qui peut se définir à l'aide d'une temporisation :

```
let timed_take d mv k =
  timeout d
    (take_mvar mv)
    (fun () -> k None)
    (fun v -> k (Some v))

let try_take mv k = timed_take 0. mv k
```

Une MVar ne se prête pas facilement à une communication bidirectionnelle, mais une MVar de requête peut transporter à la fois une donnée et la MVar à utiliser pour transmettre la réponse. On peut ainsi mettre en œuvre un mécanisme de type client-serveur, suivant un schéma classique en Erlang :

```
let rpc_send mv v k =
  let ret = make_mvar () in
  put_mvar mv (v,ret) >>= fun () ->
  take_mvar ret >>= k
```

```
let rpc_recv mv f k =
  take_mvar mv >>= fun (v,ret) ->
  put_mvar ret (f v) >>= k
```

Répétitions Le style indirect interdit l'utilisation de boucles impératives. On peut toujours utiliser la récursion mais une autre possibilité intéressante est de définir des fonctions de répétition d'opérations. Puisque en style indirect les séquences d'opérations sont représentées par des fonctions enchaînées, il parait naturel de pouvoir manipuler ces fonctions pour répéter des opérations.

Dans des applications réseau en particulier il est souvent utile de recommencer une opération qui ne s'est pas terminée de manière satisfaisante (retransmission d'un message qui n'a pas reçu de réponse par exemple). On montre ci-dessous la définition de la fonction `retry`, qui essaie au maximum `n` fois d'exécuter `op`, qui doit retourner (passer à sa continuation) une valeur de type `'a option`, valant `Some v` pour un succès avec le résultat `v` et `None` pour un échec non définitif (devant déclencher un nouvel essai). Pour un échec définitif, `op` doit lancer l'exception `Failed`. Les continuations `kfail` et `k` seront utilisées respectivement dans le cas d'un échec final et d'un succès.

```
val retry : int -> (('a option -> t) -> t) -> (unit -> t) -> ('a -> t) -> t
```

```
let retry n op kfail k =
  let rec loop i k =
    if i = 0 then raise Failed
    else
      op >>= fun res ->
      match res with
      | Some r -> k (Some r)
      | None -> loop (i-1) k
  in
  trywith
    (fun k -> loop n k)
    (fun Failed k -> k None)
    (fun r -> match r with Some r -> k r | None -> kfail ())
```

D'autres fonctions similaires sont définies pour réessayer pendant un temps déterminé, pour chaque élément d'une liste etc.

5. Parallélisme

Les applications concurrentes devraient pouvoir bénéficier du parallélisme désormais communément disponible avec les processeurs multi-coeurs. Les threads OCaml (système ou de la machine virtuelle) ne pouvant s'exécuter simultanément sur plusieurs processeurs, une solution reposerait sur l'utilisation de plusieurs processus. Nous proposons ici une extension de muthreads permettant d'assurer la communication entre threads (au sens de la bibliothèque) s'exécutant dans des processus différents.

Modèle L'idée est de permettre de partager des MVars entre deux processus, au moyen de l'interface suivante :

```
val startn : int -> (unit -> t) -> unit
val rspawn : int -> ('a mvar -> t) -> 'a mvar -> unit
val rspawn2 : int -> ('a mvar -> 'b mvar -> t) -> 'a mvar -> 'b mvar -> unit
```

`startn n main` duplique le processus courant en `n` exemplaires et lance le thread `main` dans le processus initial.

`rspawn i thr mv` permet de lancer le thread `thr` dans le processus `i` et de lui passer en paramètre une MVar qui sera *partagée* avec la MVar désignée par `mv` sur le processus demandant le `rspawn`.

`rspawn2 i thr mv1 mv2` fait de même mais en partageant deux MVars avec le thread lancé dans le processus `i`.

Mise en œuvre Quand `startn` duplique les processus il leur attribue un *pid* (de 0 pour le processus initial à n-1) et crée un maillage de canaux de communications (chaque processus peut échanger des messages avec chaque autre, désigné par son *pid*). Ces canaux permettent d'échanger des données OCaml sérialisées par les fonctions du module `Marshal` de la bibliothèque standard. En particulier, des fermetures peuvent être transmises, ce qui permet de désigner une fonction à exécuter.

Toute MVar possède deux champs mutables : `id`, égal à -1 si elle n'est pas partagée, et `rpid` (*remote pid*) contenant le *pid* du processus avec lequel elle est partagée, le cas échéant. Quand la MVar doit être partagée (opération `rspawn`) l'initiateur alloue un numéro globalement unique pour l'`id` et le met à jour ainsi que le `rpid`. Le message envoyé au processus destination contient la fermeture à exécuter et l'`id` de la MVar. Le récepteur crée une MVar avec l'`id` spécifié et le `rpid` de l'émetteur, puis lance le thread avec la MVar en paramètre.

Quand le contenu d'une MVar est modifié (suite à une opération `take_mvar` ou `put_mvar`) si la MVar est partagée (`id` différent de -1) un message est envoyé au processus correspondant à son `rpid` pour synchroniser les deux copies de la MVar. Ce message contient l'`id` de la MVar, le type d'opération `Set` ou `Del` et la valeur à placer si l'opération est `Set`.

Chaque processus gère un tableau indicé sur les `id` de MVars partagées et contenant pour chacune une fonction de modification de la MVar. Le récepteur décode l'`id` et l'opération et passe l'opération ainsi que le canal de communication à la fonction correspondante à la MVar. Celle-ci peut alors modifier le contenu de la MVar, en lisant la valeur dans le canal si l'opération est un `Put`. [6]

Une MVar partagée étant référencée depuis la fonction de modification, le gestionnaire de mémoire ne peut pas détecter qu'elle n'est plus utilisée par aucun thread. Il est pourtant nécessaire de pouvoir les libérer, et de recycler les `id` qui sont en nombre limité (tableau). La solution actuellement utilisée est de déclencher la destruction de la MVar partagée quand le thread distant (créé par `rspawn`) se termine. Un message est envoyé au processus créateur qui peut ainsi recycler l'`id`.

6. Stocker la fonction de modification plutôt que la MVar elle même permet de préserver le polymorphisme. De même, la lecture de la valeur dans le canal permet de ne pas faire apparaître son type dans le type de la fonction.

État Les idées décrites ici ont été mises en œuvre : le serveur FTP peut notamment répartir les threads DTP sur différents processus mais il n'a pas été testé très intensivement, aussi cette mise en œuvre doit-elle être considérée comme expérimentale.

Une idée que nous n'avons pas encore expérimentée pourrait permettre d'utiliser le ramasse-miette. Si la fonction de modification utilise un pointeur faible vers la MVar, le ramasse miette collectera la MVar si aucun autre lien n'est présent. Cependant ici, on veut vérifier qu'aucun autre lien n'est présent *aussi* dans l'autre processus qui partage la MVar. Grâce à la fonction `finalize` du module `Gc` on pourrait intercepter l'action du ramasse miette, et informer le processus partenaire. Si les deux ramassent miette sont déclenchés de manière synchronisée et décident tous deux de collecter la MVar, on peut les laisser agir et recycler les `id`.

6. Travaux similaires

Au moins deux bibliothèques OCaml sont actuellement disponibles pour fournir la concurrence légère : Lwt [16] et Async [9]. Toutes deux sont développées depuis des années et utilisées en production, muthreads n'a pas l'ambition de les "concurrencer" (c'est le cas de le dire !)

Contrairement à muthreads, toutes deux sont basées sur la monade de promesse (bien que Lwt appelle une promesse "thread", et Async "deferred"). Une promesse est une valeur mandataire pour une valeur qui n'est pas nécessairement disponible immédiatement, elle peut être *prête* (la valeur est alors disponible) ou *bloquée*. Un intérêt des promesses est qu'en plus d'assurer le séquencement, elles constituent un moyen de communication entre les opérations bloquantes. On peut aussi définir des opérations sur les promesses. Par exemple Async définit les opérations suivantes qui permettent de créer une promesse qui devient prête respectivement quand deux promesses, toutes les promesses d'une liste, une promesse quelconque dans une liste, deviennent prêtes :

```
val both : 'a t -> 'b t -> ('a * 'b) t
val all : 'a t list -> 'a list t
val any : 'a t list -> 'a t
```

Exceptions Le traitement des exceptions de Lwt est assez différent de celui de muthreads. Une promesse peut représenter une valeur à venir ou la valeur elle même mais aussi l'échec du calcul de cette valeur, sous la forme `Fail e`. Une telle valeur peut être générée directement par l'appel de la fonction `fail` ou par une exception OCaml "traditionnelle". Pour cela l'opérateur `>>=` rattrape systématiquement les exceptions et évalue la promesse à `Fail e` le cas échéant.

`catch f g` définit `g` comme traite-exception pour `f`. Si une exception `e` a été lancée pendant l'évaluation de `f`, la promesse retournée a la forme `Fail e`, et `catch` passe sa valeur à `g`. Dans le cas contraire, `catch` retourne simplement la promesse. Cette fonction joue ainsi le rôle de notre fonction `trywith` mais sans gestion explicite d'une pile de traite-exceptions, se reposant sur la pile de OCaml.

C'est la formulation des threads en style indirect qui motive les fonctions `trywith` de muthreads et `catch` de Lwt : celles-ci ont simplement pour but de rétablir la fonctionnalité de la construction `try/with`. Async propose la notion de *moniteur*, un mécanisme plus général prenant en compte l'aspect concurrent d'une application. Un moniteur est un contexte permettant de notifier d'éventuels *listeners* de la survenue d'une exception. Les moniteurs sont organisés en arbre dont chaque branche forme la "pile de gestion des exceptions" d'un traitement asynchrone. Il est ainsi possible à un *listener* d'être notifié des exceptions survenues dans un ensemble de traitements asynchrones.

Performances Le tableau 1 montre le temps d'exécution de l'application `thread-ring` du *Computer Language Benchmarks Game* [2], avec muthreads, Lwt, Async, les threads de la machine virtuelle

OCaml et ceux fournis pas le système. L'application consiste en un anneau de 501 threads se passant un jeton décrémenté à chaque étape jusqu'à arriver à 0. La communication se fait à l'aide de MVars (réalisées à partir de promesses suivant le modèle décrit dans [3] pour Lwt et Async). Ici le jeton avait pour valeur initiale 1 000 000. La machine utilisée est un Core 2 Duo à 2.53 GHz avec un noyau Linux 2.6.32 en architecture amd64. Le compilateur est OCaml 3.12.1.

	muthr	lwt	async	threads VM	threads natifs
code-octet	3,4	2,3	12,5	10	14,5
natif	1,1	0,4	0,8		12

TABLE 1 – Temps d'exécution de `thread-ring` (secondes)

On ne peut évidemment pas tirer de conclusions définitives de ce genre de *benchmarks*, mais on peut constater que les performances des trois bibliothèques sont relativement proches et bien meilleures que celles des threads "classiques", avec cependant une contre performance étonnante pour Async en code-octet. Signalons que Async ne repose pas sur la bibliothèque standard OCaml mais sur la bibliothèque `Core` [8] développée par Jane Street.

7. Conclusion et perspectives

La bibliothèque muthreads propose un modèle simple de concurrence légère en OCaml. Si les threads sont légères, la bibliothèque l'est aussi. La mise en œuvre d'applications de taille significative a montré que les fonctions fournies par la bibliothèque semblent adéquates. Ce développement a aussi montré l'importance de pouvoir détecter certaines erreurs par le typage, aspect qui avait été initialement négligé.

La formulation des threads en style indirect n'a pas été perçue comme une difficulté ou une gêne particulière. Quelques habitudes permettent d'alléger les notations. Par exemple on a souvent intérêt à donner aux fonctions un paramètre supplémentaire final de type `unit`. Ainsi la dernière phase du thread PI du serveur FTP est réalisée par la fonction `update_stats` définie comme

```
and update_stats dtr () = ...
```

ce qui permet de réduire le "bruit" en "économisant" quelques `fun () ->` (comparer au code montré au paragraphe **MVar et entrées/sorties** de la section 2 avec celui-ci) :

```
take_or_recv res skt
  (fun r -> rep skt r >>= fun () -> update_stats dtr) (* DTP is done  *)
  (fun m ->                                           (* cmd received *)
        ...
        rep skt 226 >>= fun () ->    (* from PI: successful abort *)
        update_stats dtr
```

Une autre possibilité serait d'utiliser un peu de sucre syntaxique. Cela pourrait surtout être intéressant pour les constructions de type `retry` qui peuvent devenir assez difficile à lire. Lwt propose quelques extensions syntaxiques basées sur le préprocesseur Camlp4.

Un opérateur similaire à `>>=` peut permettre d'insérer visuellement un calcul "pur" (non bloquant) dans une chaîne d'opérations bloquantes. La fonction `recv` décrite au début de la section 2 pourrait s'écrire :

```
val (/>>) : ('a -> 'c) -> ('c -> ('b -> t) -> t) -> 'a -> ('b -> t) -> t
let (/>>) f k = fun a -> k (f a)

let recv skt k =
  let s = String.create 512 in
  read skt s 0 512 >>= parse cmd />> k
```

Perspectives En dehors des fonctionnalités qui ne sont pas disponibles actuellement mais qui ne posent pas de problème particulier (fonctions de haut niveau pour les entrées/sorties par exemple), nous identifions les deux points suivants comme particulièrement intéressants à traiter :

- Les idées décrites dans la section sur le parallélisme pourraient être approfondies. En particulier le problème de la gestion mémoire et du recyclage des `id` mais aussi la possibilité de partager automatiquement une MVar qui serait transmise dans une MVar partagée (l'idée étant de rendre le plus proche possible le comportement des MVars locales et partagées). La mise en œuvre actuelle devrait aussi pouvoir facilement être étendue au cas de processus distribués sur différentes machines.
- Un inconvénient de l'ordonnancement coopératif est la nécessité de prévoir explicitement un découpage de longs traitements sans opération bloquante ("compute-bound"). Lwt permet de lancer un ensemble de threads système sur lesquels des traitements peuvent être *détachés*. Une autre approche que nous avons commencé à explorer reposerait sur la capture de continuations [7]. Cependant pour agir de manière préemptive ces captures doivent être asynchrones, donc être exécutées depuis un traite-signal ce qui pose quelques difficultés intéressantes...

Le code source de la bibliothèque et des applications est disponible sur la page `http://christophe.deleuze.free.fr/muthreads/`.

Références

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

[2] The computer language benchmarks game. web site. `http://shootout.alioth.debian.org/`

[3] Christophe Deleuze. Concurrence et continuations en OCaml. In *23è Journées Francophones des Langages Applicatifs*, pages 60–74, Carnac, February 2012.

[4] *epoll - I/O event notification facility.* Linux Programmer's Manual, section 7.

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

[6] Jesse James Garrett. Ajax : A new approach to web applications. blog post, February 2005. `http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications`

[7] Oleg Kiselyov. Delimited control in OCaml, abstractly and concretely system description. Technical report, March 2010. Also on FLOPS 2010. `http://okmij.org/ftp/Computation/caml-shift.pdf`

[8] Yaron Minsky. Core has landed. web page, May 2008. `https://ocaml.janestreet.com/?q=node/27`

[9] Yaron Minsky. Announcing Async. web page, October 2011. `https://ocaml.janestreet.com/?q=node/100`

[10] Node.js. Software library. `http://nodejs.org`

[11] John Ousterhout. Why threads are a bad idea (for most purposes). Presentation at Usenix 96, January 1996.

[12] 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. `http://research.microsoft.com/en-us/um/people/simonpj/papers/marktoberdorf/`

[13] J. Postel and J. Reynolds. *File Transfer Protocol (FTP).* RFC 959, IETF, October 1985.

[14] 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.

[15] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of HotOS IX*, Lihue, Kauai, Hawaii, May 2003.

[16] 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.