Manuscrit auteur, publié dans "N/P"

hal-00493223, version 1 - 18 Jun 2010

Programmation réactive en OCaml

Christophe Deleuze

Laboratoire de conception et d'intégration des systèmes (LCIS)
50, rue Barthélémy de Laffemas, BP54
F-26902 Valence Cedex 09
christophe.deleuze@lcis.grenoble-inp.fr

RÉSUMÉ. La programmation réactive permet d'écrire des programmes sous forme d'un ensemble de processus qui s'exécutent de manière synchronisée et communiquent par diffusion de signaux. Ce paradigme peut être fourni par des langages spécialisés (parfois basés sur des langages "classiques") ou par des bibliothèques. Le langage ReactiveML est un tel langage réactif basé sur OCaml. Nous décrivons ici une bibliothèque OCaml fournissant les constructions réactives de ReactiveML pour le langage OCaml lui-même. Les processus devront pour cela être rédigés en style trampoline*. Des exemples montrent que le style obtenu est raisonnable et que les performances sont au moins équivalentes.*

ABSTRACT. Reactive programming allows to write programs as a set of processes executing synchronously and communicating by signal broadcasting. This paradigm can be provided by specialized languages (sometimes based on "classic" languages) or by a library. ReactiveML is such a language based on OCaml. We describe an OCaml library that provides ReactiveML reactive constructs for the OCaml language. Processes must be written in trampolined *style. Examples show the resulting style to be acceptable, and performances at least as good.*

MOTS-CLÉS : programmation réactive, ReactiveML, langage fonctionnel, OCaml, style trampoline, bibliothèque.

KEYWORDS: reactive programming, ReactiveML, functionnal language, OCaml, trampolined style, library.

MCours.com

JESA – 43/2009. MSR 2009, pages 757 à 771

hal-00493223, version 1 - 18 Jun 2010

1. Introduction

Programmation réactive

Les langages synchrones ont été introduits dans les années 80 pour programmer des systèmes *réactifs* : systèmes qui interagissent continuellement et en temps réel avec leur environnement (Halbwachs, 1993). En particulier, Estérel (Boussinot *et al.*, 1991) adopte le paradigme impératif. Il permet d'exprimer un programme comme un ensemble de processus concurrents, synchronisés sur une notion d'instant. À chaque instant chacun des processus a l'opportunité de s'éxécuter. La communication entre les processus se fait par diffusion de signaux valués. À chaque instant, un signal est soit présent (et porteur d'une ou plusieurs valeurs) soit absent, et tous les processus en ont la même vision.

Une forte limitation du modèle synchrone est l'impossibilité de créer dynamiquement des processus ou de réaliser des calculs complexes impliquant des boucles ou l'usage de la récursion : c'est à ce prix que l'aspect temps réel du système peut être garanti. Beaucoup d'applications pourraient bénéficier du modèle réactif sans pour autant demander des contraintes temporelles fortes comme les systèmes réactifs synchrones. À partir de cette idée Frédéric Boussinot a introduit le modèle réactif (sous entendu non synchrone) qui reprend les mêmes notions en ajoutant essentiellement la possibilité de créer des processus, appelée composition dynamique (Boussinot, 1996). Ces idées ont tout d'abord été mises en œuvre dans le langage ReactiveC (Boussinot, 1991) puis en Java (Boussinot *et al.*, 2000) et ont plus récemment abouti à la proposition des *FairThreads* (Boussinot, 2006).

OCaml (Leroy, 2008) est un langage fonctionnel (avec des traits impératifs et orientés objet) de la famille ML. Nous supposons le lecteur raisonnablement familier avec ce langage. ReactiveML (Mandel, 2006) est une extension de OCaml ajoutant des constructions réactives inspirées d'Estérel. Le compilateur ReactiveML génère du code OCaml pour différents *runtimes* qui sont essentiellement des machines réactives écrites en OCaml.

Trampoline

Dans un programme une *continuation* représente "ce qui reste à exécuter" à un point donné (Reynolds, 1993). Le style CPS *(continuation passing style)* (Steele *et al.*, 1976) consiste à écrire (ou transformer) un programme de façon à rendre explicite toutes les continuations. Une fonction ne retourne jamais, elle transmet son résultat à sa continuation, qui est une fonction qu'elle a reçue en paramètre. L'exécution ne requiert donc pas de maintenir une pile des appels de fonctions, chaque appel de fonction écrasant l'enregistrement d'activation précédent.

Le style trampoline (Ganz *et al.*, 1999) se base sur le CPS pour mettre en œuvre un ordonnancement coopératif entre plusieurs processus. Ceci est illustré par la figure 1 qui montre un exemple de calcul de la factorielle en style trampoline. L'appel de $bounce$ constitue un point de coopération et son argument est la continuation du

processus, qui sera utilisée par l'ordonnanceur (appelé ici *seesaw*) pour le réactiver. Dans cet exemple, la factorielle est exécutée concurremment avec un autre processus.

```
type α thread = Done of α | Doing of (unit → α thread)

let return v = Done v
let bounce f = Doing f

(* factorial function *)

let rec fact_tramp i acc =
  if i = 0 then
    return acc
  else
    bounce (fun () → fact_tramp (i − 1) (acc × i))

(* run two computations concurrently. Return result of whichever terminates first, and
rest of other computation *)

let rec seesaw f1 f2 =
  match f1 () with
  | Done v → v, f2
  | Doing f → seesaw f2 f

let rec dotter () =
  print_char '.'; bounce dotter

(* compute factorial, printing dots at each step *)
let fact n =
  seesaw dotter (fun () → fact_tramp n 1)
```

Figure 1. *Style trampoline*

On voit donc qu'il est possible (et facile) d'exécuter des processus concurrents à partir du moment où chacun d'eux est écrit en style trampoline.

Programmation réactive en OCaml

Nous avons développé une bibliothèque permettant la programmation réactive en OCaml. Celle-ci met en œuvre l'ordonnancement et les communications entre processus, en reprenant le modèle de ReactiveML. Les processus devront être écrits en style trampoline, et feront appels aux fonctions de la bibliothèque.

Fournir ces mécanismes au sein du langage plutôt que par une extension apporte un certain nombre d'avantages. L'intégralité du langage est disponible (ReactiveML ne supporte pas les foncteurs ni les objets), ainsi que les outils associés (débogueur, profileur, environnement de développement). D'autre part la bibliothèque peut être utilisée avec des extensions de OCaml comme MetaOCaml (Taha, 2004). D'un autre côté, au contraire d'une bibliothèque, un langage spécifique comme ReactiveML peut

hal-00493223, version 1 - 18 Jun 2010

adapter la syntaxe aux nouveaux concepts introduits[1] et surtout réaliser à la compilation des vérifications sémantiques spécifiques.

Dans la section suivante, nous présentons les principales constructions de ReactiveML et leurs équivalents pour notre bibliothèque. La section 3 décrit deux exemples de programmes réactifs écrits à l'origine en ReactiveML. La section 4 décrit les grandes lignes de l'implémentation de la bibliothèque. La section 5 présente une évaluation des performances réalisée sur les deux exemples. Conclusion et perspectives terminent l'article.

2. Constructions réactives en style trampoline

ReactiveML fournit un ensemble de constructions syntaxiques pour les processus. La bibliothèque les met en œuvre sous forme de fonctions. Pour pouvoir être manipulé, un processus sera représenté le plus souvent par une fonction "glaçon" *(thunk)*. L'instruction d'exécution run sera donc réalisée simplement par la fonction let $run\ p\ =\ p\ ()$. Nous noterons RML les fragments de code ReactiveML et TML (pour *trampoline* ML) ceux pour la bibliothèque.

Trampoline

Le style trampoline consiste à rendre explicite les continuations aux endroits où le processus est susceptible de se suspendre. Par exemple l'instruction pause indique que le processus attend l'instant suivant pour continuer son exécution. Un autre est l'attente (await) de l'émission d'un signal s : le processus sera relancé au prochain instant suivant l'émission du signal. La valeur du signal est reçue en paramètre par la fonction de continuation. Le processus *demo* ci-dessous (à gauche RML, à droite TML) affiche un message au premier instant puis attend l'émission du signal *s1* et affiche la valeur émise. *term* permet de terminer le processus.

```
let process demo =
    print_string "do nothing now!";
    pause;
    print_string "Waiting for s1.";
    await s1(n) in
      printf "s1 value is: %i\n" n
```

```
let demo () =
    print_string "do nothing now!";
    pause (fun () →
    print_string "Waiting for s1.";
    await s1 (fun n →
    printf "s1 value is: %i\n" n;
    term () ))
```

Certaines constructions nécessitent de "factoriser" le code d'une continuation "lointaine" commune. C'est le cas de l'instruction present qui exécute instantanément (c'est-à-dire dans l'instant courant) la partie c_then si le signal est présent dans l'instant, la partie c_else à l'instant suivant dans le cas contraire (ce retard de réaction à l'absence d'un signal permet d'éviter les problèmes de causalité).

1. Quoique OCaml dispose des outils Camlp4 et Camlp5 pour étendre sa syntaxe.

hal-00493223, version 1 - 18 Jun 2010

MCours.com

present s then c_then else c_else; c_after ...	let $after$ () = c_after ... in $present\ s$ (fun () → c_then; $after$()) (fun () → c_else; $after$())

Processus imbriqués

Pour un processus jouant le rôle de "sous-programme" pour un autre, on n'utilisera pas un glaçon mais une fonction prenant sa continuation en paramètre (cas de $pause3$ ci-dessous). await one récupère *une* seule des valeurs émises sur le signal.

let process $pause3$ = pause; pause; pause	let $pause3\ k$ = $pause$ (fun () → $pause$ (fun () → $pause\ k$))
let process $main$ = await $s1$; run $pause3$; await one $s2(n)$ in ...	let $main$ () = $await\ s1$ (fun _ → $pause3$ (fun () → $await_one\ s2$ (fun n → ...)))

Création de processus

La "composition parallèle" (opérateur || de ReactiveML) est réalisée par deux fonctions par et $tail_par$ selon qu'il y a une continuation ou pas. Les processus $p1$ et $p2$ sont ici des fonctions glaçons pour $tail_par$ et des fonctions prenant leur continuation en paramètre pour par.

RML	TML	RML	TML
(run $p1$ || run $p2$); ...	par $p1$ $p2$ (fun () → ...)	... run $p1$ || run $p2$	... $tail_par$ $p1$ $p2$

Type des signaux

Notre implémentation contraint tous les signaux à avoir le même type, ce qui est une limitation forte[2]. On peut la contourner en définissant, pour un programme donné, un type somme pour tous les types de valeurs portées par des signaux. La bibliothèque se présente sous forme d'un foncteur prenant le type des signaux en paramètre. Nous reviendrons sur ce point en section 3.

2. Suite aux remarques de relecteurs nous avons développé une version de TML levant cette contrainte. Elle est cependant plus complexe et encore trop peu mature pour être décrite ici.

hal-00493223, version 1 - 18 Jun 2010

Constructions de contrôle

Trois constructions de contrôle agissent sur l'exécution des processus. Elles sont réalisées en TML sous forme d'une fonction. Par exemple, do/until est la construction de *préemption* : le processus contenu est préempté (terminé) en fin d'instant si le signal associé a été émis. Les deux autres constructions (*suspension* et *désactivation*) sont traitées de la même façon.

```
do
    print_string "Wait for s...";
    await s;
    print_string "Arrived!"
until s'
print_string "Left do/until!";
...
```

```
dountil s'
    (fun () →
        print_string "Wait for s...";
        await s (fun _ →
        print_string "Arrived!"))
    (fun () →
        print_string "Left do/until!";
        ...
```

3. Exemples

Sieve

Le crible d'Ératosthène est un exemple classique de l'approche réactive d'abord proposé dans le contexte d'un langage flot de données (Kahn *et al.*, 1977). Le crible se construit comme une chaîne de processus *filter* encadrés par un générateur (*integers*) d'un côté, un récepteur (*output*) précédé d'un "étendeur" (*shift*) de l'autre, comme illustré figure 2 qui montre l'état initial et après la création du filtre sur 2. Les nombres progressent d'une étape de la chaîne à chaque instant.

Figure 2. *Les processus du crible*

Détaillons chacun des processus (figure 3) :

– *integers* génère sur un signal la suite des entiers à partir de 2.

– *filter* filtre les multiples de son paramètre : il reçoit des entiers sur son signal d'entrée et réemet ceux qu'il ne filtre pas sur son signal de sortie. La boucle infinie de RML est ici exprimée par la récursion.

– *shift* reçoit des entiers sur son signal d'entrée, crée un nouveau signal (appelé s localement) et insère un processus *filter* dans la chaîne à chaque nouveau nombre reçu (qui sera premier par construction de la chaîne). On utilise ici la fonction $tail_par$.

– *output*, placé en fin de chaîne, est chargé de l'affichage du nombre premier qui a passé tous les filtres. Il termine le programme quand le nombre reçu atteint une valeur limite. L'appel à $Mem.exit$ affiche la mémoire allouée dans le tas avant de terminer.

hal-00493223, version 1 - 18 Jun 2010

MCours.com

sieve.rml

```
let llast = ref 0
let show = ref true

let rec process integers n s_out =
  emit s_out n;
  pause;
  run (integers (n + 1) s_out)

let process filter prime s_in s_out =
  loop
    await one s_in(n) in
    if (n mod prime) ≢ 0 then
        emit s_out n
  end

let rec process shift s_in s_out =
  await one s_in(prime) in
  emit s_out prime;
  signal s in
  run (filter prime s_in s)
  ||
  run (shift s s_out)

let process output s_in =
  loop
    await one s_in (n) in
    if n ≥ !llast then Mem.exit ();
    if !show then begin
      print_int n;
      print_string " ";
      flush stdout
    end
  end

let process sieve =
  Arg.parse [ ... ]
    (fun _ → ()) "sieve";
  signal nat in
  signal primes in
  run (integers 2 nat)
  ||
  run (shift nat primes)
  ||
  run (output primes)
```

sieve.ml

```
module TML =
  Tml.Gen_TML(struct type t =int end)

open TML

let last = ref 0
let show = ref true

let rec integers n s_out () =
  emit s_out n;
  pause (integers (n + 1) s_out)

let rec filter prime s_in s_out () =
  await_one s_in (fun n →
  if (n mod prime) ≢ 0 then
    emit s_out n;
  run (filter prime s_in s_out))

let rec shift s_in s_out () =
  await_one s_in (fun prime →
  emit s_out prime;
  let s = signal () in
  tail_par (filter prime s_in s)
           (shift s s_out))

let rec output s_in () =
  await_one s_in (fun n →
  if n ≥ !last then Mem.exit ();
  if !show then begin
    print_int n;
    print_string " ";
    flush stdout
  end;
  run (output s_in))

let sieve () =
  let nat = signal () in
  let primes = signal () in
  tail_parn [ integers 2 nat;
    shift nat primes; output primes ]

Arg.parse [ ... ]
  (fun _ → ()) "sieve";

start [ sieve ]
```

Figure 3. *Code pour le crible, en RML et TML*

hal-00493223, version 1 - 18 Jun 2010

– $sieve$, processus initial qui amorce le système. On remarque l'utilisation du $tail_parn$, similaire à $tail_par$ mais prenant une liste de processus en paramètre. $start$ démarre les processus passés en paramètre (en RML le processus initial est fixé à la compilation).

Tous les signaux portant des entiers, le type passé au foncteur est simplement *int*. On note que la formulation des processus comme glaçons amène souvent une notation élégante et proche de celle de ReactiveML.

Elip

Elip est un logiciel de simulation de protocole de routage pour réseaux ad'hoc, écrit en ReactiveML (Mandel *et al.*, 2005). Chaque nœud du réseau est implémenté par un processus. Il a pu très facilement être porté sur TML. Sur les 4300 lignes de code, seules 270 concernent les définitions de processus (au nombre de 11) et doivent donc être converties en style trampoline. Aucune difficulté notable n'a été rencontrée.

Le programme utilise 10 signaux, dont 4 non valués. Les autres portent des articles (types `position` et `node`), une paire de flottants, un caractère ou un entier. Le module paramètre du foncteur sera donc le suivant :

```
module SV =
  struct
    type t = SV_Null (* signaux non valués *)
      | SV_Pos of position
      | SV_Node of node
      | SV_Click of float × float
      | SV_Key of char
      | SV_Int of int
  end
```

L'émission d'un signal devra utiliser le constructeur correspondant, par exemple $emit\ click\ (SV_Click\ (pos_x,\ pos_y))$.

4. Réalisation de la bibliothèque

Nous décrivons ici les grandes lignes d'une version simplifiée de la bibliothèque. Le code détaillé est disponible sous forme d'un document "litterate programming" (Deleuze, 2009) montrant la mise en œuvre progressive des fonctionnalités. La bibliothèque complète comprend environ 500 lignes de code.

Principes

Les processus sont des fonctions glaçons retournant des valeurs de type $coop$ indiquant la raison de la suspension et la continuation éventuelle.

hal-00493223, version 1 - 18 Jun 2010

```
type process = unit → coop
and coop =
  | Done
  | Pause of process
  | Wait of signal_t × awproc
  | Wait_im of signal_t × awproc
  | Present of signal_t × process × process
```

awproc (pour *awaiting process*) est le type des processus attendant l'arrivée d'un signal (sans se préoccuper des valeurs éventuelles portées), l'intégralité des valeurs émises sur le signal ou une seule des valeurs émises. *sval_t* est le type somme des valeurs de signaux, paramètre du foncteur.

```
and awproc =
  | No of (unit → coop)
  | One of (sval_t → coop)
  | All of (sval_t list → coop)
```

Les fonctions de suspension que nous avons vues précédemment consistent essentiellement à retourner une valeur de type *coop* (*ispst* indique si un signal a déjà été émis dans l'instant courant).

```
let term () = Done
let pause k = Pause k
let await s k = Wait(s, All k)
let await_one s k = Wait(s, One k)
let await_immediate s k = if ispst s then k () else Wait_im(s, No k)
let await_immediate_one s k =
  if ispst s then k (List.hd s.value) else Wait_im(s, One k)
let present s k1 k2 = if ispst s then k1 () else Present(s, k1, k2)
```

Les processus sont stockés dans diverses listes :

– *runq* : processus en attente d'exécution sur l'instant,

– *pauseq* : processus en attente de l'instant suivant,

– à chaque signal sont associées trois listes *present*, *await_im* et *await* contenant les processus en attente sur ce signal.

Au cours de chaque instant l'ordonnanceur (*sched*) lance un à un les processus de la liste *runq* et récupère les continuations (dans des valeurs de type *coop*) qu'il place dans la liste appropriée (*pauseq* si le processus a terminé l'instant, une liste d'attente associée à un signal si le processus attend ce signal, voir figure 4a). Si un processus émet un signal, les processus en attente immédiate (*await_im* et *present*) sur ce signal sont immédiatement replacés dans *runq*.

L'instant se termine quand l'ordonnanceur trouve *runq* vide. Il exécute alors la fonction *next_instant* qui transfère le contenu de *pauseq* dans *runq* ainsi que les

hal-00493223, version 1 - 18 Jun 2010

processus qui ne doivent plus être en attente à l'instant suivant (*await* sur un signal présent, *present* sur un signal absent, figure 4b).

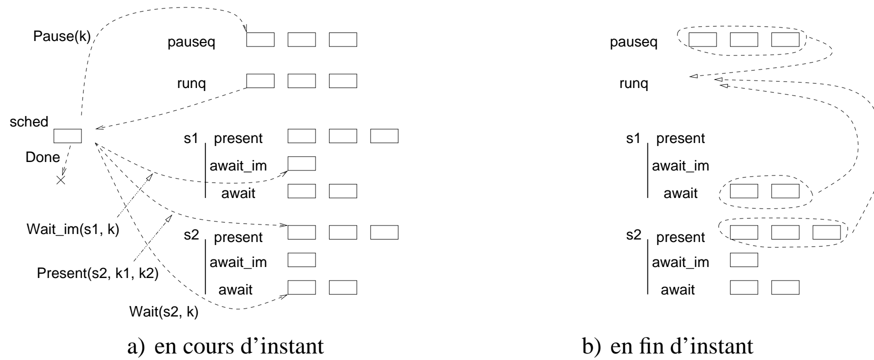


Figure 4. *Utilisation des listes de processus*

Création de processus

Les fonctions de création de processus (*par* et ses variantes) se basent sur la fonction *spawn* qui ajoute un processus dans *runq* et sur *par_help* qui met en place la synchronisation des deux processus. *parn* et *tail_parn* sont réalisées de façon similaire.

```
let par_help k1 k2 k3 =
  let you_re_last = ref false in
  let test _ =
    if !you_re_last then k3 () else begin
        you_re_last := true;
        term()
    end
  in
  (fun () → k1 test),
  (fun () → k2 test)
```

```
let spawn k = runq := k :: !runq

let par e1 e2 k =
  let k1, k2 = par_help e1 e2 k
  in
  spawn k1;
  k2 ()

let tail_par k1 k2 =
  spawn k1;
  k2 ()
```

Signaux

Un signal est une structure contenant des champs mutables dont les deux derniers instants où le signal a été émis (permet de tester le status du signal à l'instant courant et à l'instant précédent), les deux dernières (listes de) valeurs émises (de type *sval_t*, le type fourni en paramètre au foncteur), une fonction de collecte des valeurs émises au cours d'un instant et les trois listes de processus en attente.

```
and signal_t = {
    mutable last_emit : int;
    mutable prev_emit : int;
```

hal-00493223, version 1 - 18 Jun 2010

MCours.com

```
    mutable value : sval_t list;
    mutable prev_value : sval_t list;
    gather : (sval_t list → sval_t list);
    mutable await : awproc list;
    mutable await_im : awproc list;
    mutable present : (process × process) list
  }
```

Pour limiter les traitements en fin d'instant la fonction $next_instant$ ne considère que les signaux présents dans la liste $used_signals$. Ces signaux *utilisés* sont ceux qui ont été émis sur cet instant ou pour lesquels un processus est en attente dans la liste $present$. Cette liste est remise à vide pour l'instant suivant.

Constructions de contrôle

Chaque processus est associé à une pile de contexte indiquant les conditions dans lesquelles il peut être suspendu, préempté ou désactivé ainsi que la continuation à lancer le cas échéant.

```
and contextelement =
   | Until of signal_t × process
   | Control of signal_t × process
   | When of signal_t × awproc
and ctxt = contextelement list
```

L'implémentation réelle est un peu plus complexe, le nombre de contextes de préemption ou de désactivation étant maintenu avec la liste des contextes pour être accessible sans parcours de la liste.

La présence des contextes complique la gestion des signaux. En effet, un processus soumis à un do/until qui doit être préempté en fin d'instant peut se trouver dans n'importe quelle liste. La notion de signaux *utilisés* est étendue et de nouvelles listes sont mises en place pour limiter le plus possible le coût des opérations de fin d'instant. Nous renvoyons le lecteur à (Deleuze, 2009) pour les détails.

À titre d'exemple, la fonction $dountil\ s\ e\ k$ a pour effet de placer en tête de $runq$ un processus e dans un contexte préempté par s et avec la continuation k, puis de terminer (avec une nouvelle valeur de coopération $Next$). L'ordonnanceur va relancer depuis $runq$ le processus dans le nouveau contexte. Quand e se termine, le sommet de sa pile de contexte est dépilé et sa continuation lancée.

```
let dountil s e k =
  let cel = !current_context in
  put_in_runq ((cel @ [Until(s, k)]), e); Next
```

Les processus suspendus sont placés dans la liste $await_im$ du signal correspondant. En fin d'instant la pile de contexte de chaque processus est examinée et les

opérations nécessaires sont effectuées (préemption, désactivation ou réactivation – éventuellement sur plusieurs niveaux).

5. Performances

Nous comparons les performances, en OCaml code-octet et natif, des trois *runtimes* de RML (Lco_ct, Lco_ct_class et Lk) avec trois versions de notre bibliothèque TML :

– TML4 version sans constructions de contrôle

– TML5 constructions de contrôle non imbriquées

– TML6 constructions de contrôle imbriquées.

Les mesures décrites ci-dessous ont été effectuées sur des machines équipées d'un processeur Intel Core 2 Duo à 2,33 Ghz et 2 Go de mémoire, utilisant un noyau linux 2.6.24, OCaml 3.09.2, RML 1.07.1. Le temps de calcul a été mesuré par la commande `time` et l'occupation mémoire a été fournie par le module Gc de OCaml.

Sieve

Les figures 5 et 6 montrent les performances pour l'application `sieve`. Le temps d'exécution est avantageux pour TML avec les versions incomplètes de la bibliothèque, particulièrement en cas de la compilation native. Les constructions de contrôle induisent clairement un coût important. En code-octet la situation est moins claire, en particulier TML6 est plus lent que tous les *runtimes* RML. Il est possible que la gestion (assez complexe) des constructions de contrôle imbriquées par TML ne soit pas la plus efficace possible.

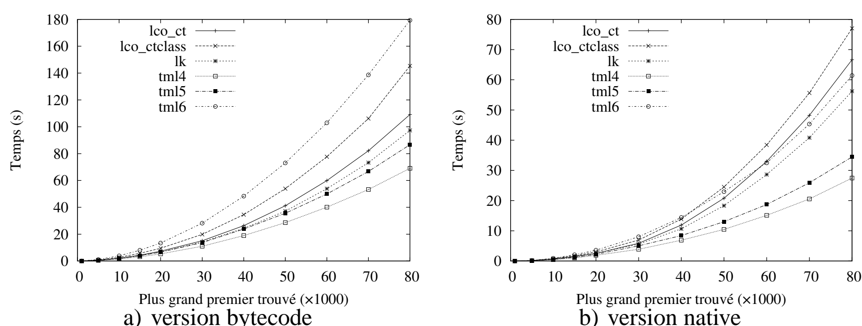
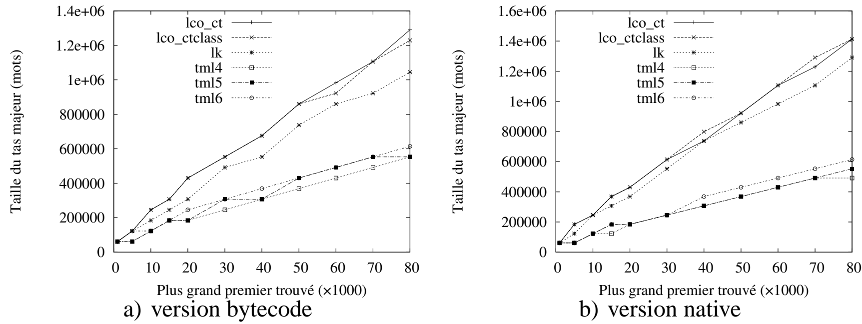


Figure 5. *Crible, temps d'exécution*

L'occupation mémoire est par contre nettement à l'avantage de TML. Que cela soit en code-octet ou en natif, on observe un gain d'au moins 50 %. Les figures ne le montrent pas mais le gain est un peu plus faible si l'on utilise la fonction *par* au lieu de

hal-00493223, version 1 - 18 Jun 2010

$tail_par$ (dont RML n'a pas d'équivalent) mais reste conséquent (24 % en code-octet et 43 % en natif).

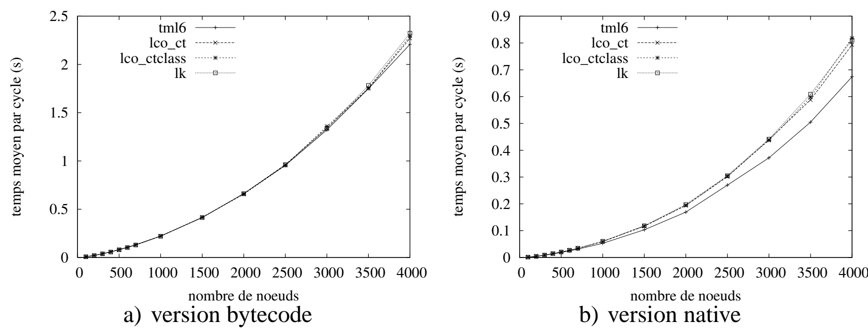


a) version bytecode b) version native

Figure 6. *Crible, espace mémoire*

Elip

`elip` contient des instructions de contrôle imbriquées, seul TML6 est alors utilisable (figure 7). Globalement, la différence entre RML et TML est ici beaucoup moins nette. En ce qui concerne l'occupation mémoire, nous ne montrons pas les graphes par manque de place, les implémentations sont extrêmement proches. Nous supposons qu'ici l'avantage de TML est noyé dans la masse de mémoire allouée par l'application elle même, alors que dans le cas de `sieve` l'application elle même n'allouait que très peu de mémoire, la différence entre les implémentations apparaissait alors très nettement. Cette interprétation est confirmée par le fait que tous les *runtimes* RML ont le même comportement alors que des différences étaient perceptibles avec le crible. Par contre on peut relever un léger avantage en temps pour TML dans le cas de compilation native.



a) version bytecode b) version native

Figure 7. *Elip, temps d'exécution*

hal-00493223, version 1 - 18 Jun 2010

hal-00493223, version 1 - 18 Jun 2010

Les simulations ont été exécutées sur 500 cycles avec une densité constante de 20, une graine du générateur aléatoire égale à 0, et sans l'interface graphique (`-n` *`nodes`* `-N 500 -D 20 -seed 0 -nox`).

6. Conclusion et perspectives

Le but que nous nous étions fixé était de reproduire les fonctionnalités de ReactiveML en OCaml pur et de voir si le style et les performances étaient raisonnables.

Les exemples que nous avons présentés nous conduisent à penser que le style trampoline est raisonnablement utilisable avec un peu de pratique. Notons que l'on peut aussi adopter un style monadique en définissant simplement un opérateur infixe "bind" par let $(>>=)\ inst\ k\ =\ inst\ k$. On obtient une formulation très similaire aux systèmes de threads basés sur les monades comme `Lwt` (Vouillon, 2008). Le processus $integers$ du crible, par exemple, pourra alors s'écrire :

```
let rec integers n s_out () =
  emit s_out n;
  pause >>= integers (n + 1) s_out
```

Le style trampoline consiste à faire apparaître explicitement les continuations aux endroits où elles sont nécessaires. Une autre approche consiste à utiliser les continuations implicites en réalisant des captures de continuation. De cette façon la formulation des opérations bloquantes peut se faire en style direct (de la forme let $v\ =\ await\ s$ in ...) et les boucles n'ont pas besoin d'être transformées en fonctions récursives. La bibliothèque caml-shift (Sabry *et al.*, 2008) implémente la capture de continuations partielles[3] mais elle n'est disponible qu'en code octet et les performances sont sensiblement moins bonnes : sur le crible, nous avons observé un triplement de l'occupation mémoire et du temps de calcul.

Les tests ont montré que les performances sont au minimum honorables, et parfois avantageuses en mémoire. Cette bibliothèque pourrait être préférable à ReactiveML pour des applications utilisant un très grand nombre de processus très simples.

De nombreux travaux s'intéressent à la gestion de la concurrence sans support du système, ou plus généralement à l'ordonnancement coopératif. En particulier *FairThreads* (Boussinot, 2006) propose un modèle de programmation concurrente inspiré du modèle réactif, et reposant sur des passages de jetons entre threads système. Une implémentation trampoline pourrait être très avantageuse en terme de ressources, notamment pour des applications composées d'un très grand nombre de processus fortement couplés.

Les extraits de code apparaissant dans ce document ont été formatés avec l'outil ocamlweb (modifié pour traiter ReactiveML).

3. Aussi appelées continuations composables ou continuations délimitées.

7. Bibliographie

Boussinot F., « Reactive C: An extension of C to program reactive systems », *Software: Practice and Experience*, vol. 21, n° 4, p. 401-428, avril, 1991.

Boussinot F., *La programmation réactive. Application aux systèmes communicants*, Masson et CNET-ENST, 1996.

Boussinot F., « FairThreads: mixing cooperative and preemptive threads in C », *Concurrency and Computation: Practice and Experience*, vol. 18, n° 5, p. 445-469, avril, 2006. Also available as a research report.

Boussinot F., de Simone R., « The ESTEREL language », *Proceedings of the IEEE*, vol. 79, n° 9, p. 1293-1304, septembre, 1991.

Boussinot F., Susini J.-F., « Java threads and SugarCubes », *Software: Practice and Experience*, vol. 30, n° 5, p. 545-566, avril, 2000.

Deleuze C., Programmation réactive en OCaml – Implémentation de la bibliothèque TML, Technical report, LCIS, 2009.

Ganz S. E., Friedman D. P., Wand M., « Trampolined Style », *International Conference on Functional Programming*, p. 18-27, 1999.

Halbwachs N., *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, 1993.

Kahn G., MacQueen D. B., « Coroutines and networks of parallel processes », *Information processing*, Toronto, p. 993-998, août, 1977.

Leroy X., *The Objective Caml system – Documentation and user's manual*, release 3.11, INRIA. 2008.

Mandel L., Conception, Sémantique et Implantation de ReactiveML : un langage à la ML pour la programmation réactive, PhD thesis, Université Paris 6, mai, 2006.

Mandel L., Benbadis F., « Simulation of Mobile Ad hoc Network Protocols in ReactiveML », *Proceedings of Synchronous Languages, Applications, and Programming (SLAP'05)*, Edinburgh, Scotland, avril, 2005.

Reynolds J. C., « The Discoveries of Continuations », *LISP and Symbolic Computation*, vol. 6, n° 3–4, p. 233-247, 1993.

Sabry A., chieh Shan C., Kiselyov O., « Native delimited continuations in (byte-code) OCaml », OCaml library, 2008.

Steele G. L., Sussman G. J., Lambda: The Ultimate Imperative, Technical Report n° AIP-353, Massachusetts Institute of Technology, Cambridge, MA, USA, 1976.

Taha W., « A Gentle Introduction to Multi-stage Programming », *Domain-Specific Program Generation*, Springer Berlin / Heidelberg, p. 30-50, 2004.

Vouillon J., « Lwt: a cooperative thread library », *ML '08: Proceedings of the 2008 ACM SIGPLAN workshop on ML*, ACM, New York, NY, USA, p. 3-12, 2008.

hal-00493223, version 1 - 18 Jun 2010

MCours.com