Programmation réactive en OCaml

Christophe Deleuze
Laboratoire de conception et d'intégration des Systèmes (LCIS)
50, rue Barthélémy de Laffemas, BP54
26902 Valence Cedex 09
France

Décembre 2009

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.

1 Introduction

1.1 Programmation réactive

Langages synchrones 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 [1]. Les langages synchrones les plus connus sont Estérel [2], Lustre [3] et Signal [4]. Lustre et Signal sont des langages "flot de données", alors qu'Estérel 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 (multi) 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.

Un programme Estérel se compile sous forme d'un automate dans lequel le parallélisme présent dans le programme a été séquentialisé. Aucun parallélisme n'est nécessaire au niveau du support d'exécution et il est facile de s'assurer d'un temps de réaction garanti.

Modèle reactif 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 [5] (sous entendu non synchrone) qui intègre ces notions à un langage "classique", permettant donc les boucles et en ajoutant la possibilité de créer des processus, appelée composition dynamique. Ces idées ont tout d'abord été mises en œuvre dans le langage ReactiveC [6] puis en Java [7] et ont plus récemment abouti à la proposition des *FairThreads* [8].

Bien entendu, il n'est plus possible de faire disparaître le parallélisme dans un automate, un support d'exécution mettant en œuvre un ordonnancement coopératif est nécessaire et il devient impossible en général de garantir un temps de réaction du système.

ReactiveML OCaml [9] est un langage fonctionnel (avec des traits impératifs et orientés objet) de la famille ML. Nous supposons le lecteur raisonnablement familier avec ce langage. Par fonctionnel on entend en particulier que les fonctions sont des *valeurs de première classe*. Elles peuvent être passées en paramètres, retournées comme valeur résultat d'un appel de fonctions, composées, stockées dans des structures de données etc. ReactiveML [10] est une extension de OCaml ajoutant des constructions réactives inspirées d'Estérel. Le compilateur ReactiveML peut générer du code pour différents *runtimes* qui sont essentiellement des machines réactives écrites en OCaml. Il fournit donc des fichiers sources OCaml qui sont destinés à être compilés et liés à une bibliothèque.

1.2 Trampoline

Dans un programme une *continuation* représente "ce qui reste à exécuter" à un point donné [11]. Le style CPS *(continuation passing style)* [12] 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 dont pas de maintenir une pile des appels de fonctions, chaque appel de fonction écrasant l'enregistrement d'activation précédent. L'appel de continuation est ainsi parfois appelé "goto avec argument" [13].

Illustrons ceci sur un exemple. Soit la fonction OCaml *len* ci-dessous qui calcule la longueur d'une liste.

```
let rec len l =
  match l with
  | [] → 0
  | h :: t → 1 + (len t)
```

Appelons c la continuation de l'appel *len l1*. Si *l1* est la liste vide, alors c recevra la valeur 0. Sinon, *l1* est de la forme $h :: t$. Ce qu'il faut faire après avoir calculé la longueur de t, c'est lui ajouter 1 puis donner cette valeur à c. La continuation de *len t* est donc fun $n \rightarrow c\ (1 + n)$. On en déduit la version CPS de notre fonction :

```
let rec len l c =
  match l with
  | [] → c 0
  | h :: t → len t (fun n → c (1 + n))
```

Le style trampoline [14] se base sur le CPS et permet d'exprimer sous forme de fonction des processus utilisant un ordonnancement coopératif. Ceci est illustré par la figure 1 qui montre un exemple de tel processus calculant la factorielle d'un entier. L'appel de *bounce* constitue un point de coopération et son argument est la continuation du processus, qui sera utilisé par l'ordonnanceur (appelé ici *pogostick*) pour réactiver le processus. Dans la figure 2, la factorielle est exécutée concurremment avec un autre processus affichant des points (ordonnanceur *seesaw*).

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.

1.3 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 rédigés en style trampoline, et feront appel 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 par exemple MetaOCaml [15][1]. D'un autre

[1] Cela est en fait aussi possible avec ReactiveML puisque le compilateur génère des sources OCaml, mais est certainement beaucoup moins confortable.

```
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))

(* one thread scheduler *)

let rec pogostick f =
  match f () with
  | Done v → v
  | Doing f → pogostick f

(* give our trampolined fact function to the scheduler *)

let fact n =
  pogostick (fun () → fact_tramp n 1)
```

FIG. 1 – Exemple de style trampoline

```
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)
```

FIG. 2 – Style trampoline, suite

côté, au contraire d'une bibliothèque, un langage spécifique comme ReactiveML peut adapter la syntaxe aux nouveaux concepts introduits[2] et réaliser à la compilation des vérifications sémantiques spécifiques.

Dans la section suivante, nous présentons les principales constructions de ReactiveML et leurs équivalents dans 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 ()`. Dans la suite nous noterons RML les fragments de code ReactiveML et TML

[2]Quoique OCaml dispose des outils `Camlp4` et `Camlp5` pour étendre sa syntaxe.

(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. On peut noter que l'indentation utilisée pour TML ne reflète pas l'imbrication des paramètres mais suggère au contraire le déroulement séquentiel du 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).

```
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())
```

La construction de boucle infinie loop/end peut être formulée de manière récursive :

```
loop
    i_1;
    ...
    i_n
end
```

```
let rec loop () =
    i_1;
    ...
    i_n;
    loop()
in loop()
```

et son corps rédigé en style trampoline. Ainsi (await immediate est similaire à await sauf que la suite est exécutée dans l'instant où le signal est émis) :

```
loop
    print_string "Awaiting s...";
    await immediate s;
    print_string "Yep!"
end
```

```
let rec loop() =
    print_string "Awaiting s...";
    await_immediate s (fun () →
    print_string "Yep!";
    loop ())
in loop()
```

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 process main =
  await s1;
  run pause3;
  await one s2(n) in ...
```

```
let pause3 k =
  pause (fun () →
  pause (fun () →
  pause k))

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*	*par*	...	...
||	*p1*	run *p1*	*tail_par*
run *p2*);	*p2*	||	*p1*
...	(fun () → ...)	run *p2*	*p2*

Type des signaux Notre implémentation contraint tous les signaux à avoir le même type, ce qui est une limitation forte. 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. Ceci sera illustré en section 3.2. Une version expérimentale levant cette contrainte a également été réalisée. Nous la décrirons brièvement en fin de section 4.

Exceptions Le compilateur ReactiveML vérifie que les exceptions sont gérées de manière instantanée (i.e. à l'intérieur d'un instant). Nous n'avons pas à nous en préoccuper, le style trampoline assurant syntaxiquement cette contrainte.

Constructions de contrôle Trois constructions de contrôle agissent sur l'exécution des processus. do/until est la construction de *préemption* : le processus contenu est préempté (terminé) en fin d'instant si le signal associé a été émis.

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

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

ReactiveML fournit également une construction de *suspension* do/when : le processus suspendu n'est exécuté qu'au cours des instants où un signal donné est émis ; et une construction de *désactivation* control/with qui désactive ou réactive le processus contrôlé à chaque instant où un signal donné est émis.

3 Exemples

3.1 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 [16]. Il est composé des processus (figure 4) :

– *integers* qui génère sur un signal la suite des entiers à partir de 2.
– *filter* qui 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 localement *s*) 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, 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 le temps processeur utilisé et la mémoire allouée dans le tas avant de terminer.
– *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).

Le crible se construit donc comme une chaîne de processus *filter* encadrés par un générateur d'un côté, un récepteur précédé d'un "étendeur" de l'autre, comme illustré figure 3 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.

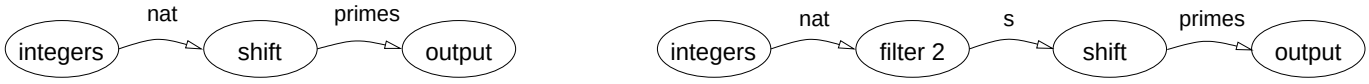


FIG. 3 – Les processus du crible

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.

Chacun sait qu'il existe des façons plus efficaces de mettre en œuvre le crible, mais cet algorithme, par sa simplicité et son élégance, nous a paru pertinent pour cette étude. La version itérative "classique" peut d'ailleurs se voir comme une optimisation rendue possible par l'organisation systématique que prennent les processus.

3.2 Elip

Elip est un logiciel de simulation de protocole de routage pour réseaux ad'hoc, écrit en ReactiveML [17]. 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*)).

La figure 5 montre le début du processus principal de `elip`. Après la création de signaux, plusieurs séries de processus sont lancés en parallèle, la première série (quatre processus) étant soumise à une construction de contrôle control/with avec le signal *suspend*.

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 [ ("-n", Arg.Set_int llast,
               "set last number to try");
              ("-q", Arg.Clear show,
               "quiet : dont display numbers") ]
    (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 [ ("-n", Arg.Set_int last,
             "set last number to try");
            ("-q", Arg.Clear show,
             "quiet : dont display numbers") ]
  (fun _ → ()) "sieve";

start [ sieve ]
```

FIG. 4 – Code pour le crible, en RML et TML

simul_pr.rml

```
let process main =
  Area.make_areas;
  signal suspend in
  signal start in
  signal new_node, kill in
  control
    run (make_preemptible_nodes nb_nodes kill)
    ||
    run (Dynamic.add new_node start)
    ||
    loop
      emit start;
      pause;
      pause;
      pause;
    end
    ||
    if with_stat then run (Stat.stat start)
  with suspend
  ||
  (if with_graphics then
    run (draw_simul draw suspend new_node kill))
  ||
  ...
```

simul_pr.ml

```
let main () =
  Area.make_areas;
  let suspend = signal () in
  let start = signal () in
  let new_node = signal () in
  let kill = signal () in
  tail_parn
    [
     (fun () → controlwith suspend
          (fun () → tail_parn
              [
               make_preemptible_nodes nb_nodes kill;
               (Dynamic.add new_node start);
               let rec loop () =
                 emit start SV_Null;
                 pause (fun () →
                 pause (fun () →
                 pause loop))
               in loop;
               if with_stat then Stat.stat start else term
              ])
          term)
     ;
     (if with_graphics then
       draw_simul draw suspend new_node kill else term)
       ;
       ...
```

FIG. 5 – Code pour le début du processus *main* de `elip`

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" [18] 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.

```
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)
```

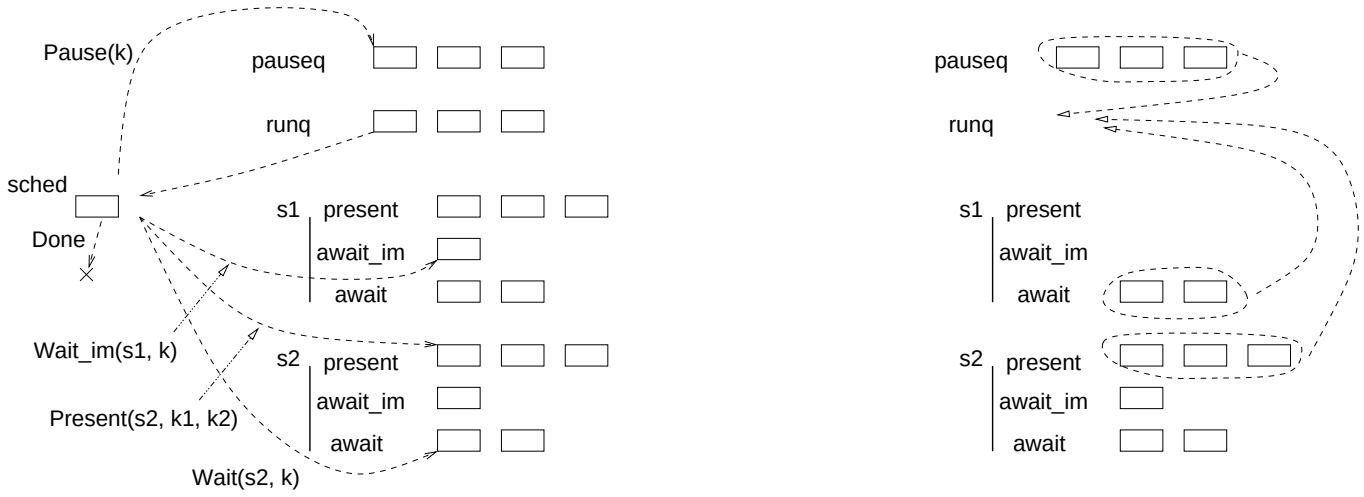
Les fonctions de suspension que nous avons vues précédemment consistent essentiellement à retourner une valeur de type *coop* (*ispst* – pour *is present* – indique si un signal a déjà été émis dans l'instant courant).

```
let term () = Done
let pause p = Pause p
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 6(a)). 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 processus qui ne doivent plus être en attente à l'instant suivant (*await* sur un signal présent, *present* sur un signal absent, figure 6(b))).



(a) En cours d'instant

(b) En fin d'instant

FIG. 6 – 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
  (fun () → k1
      (fun _ → if !you_re_last then k3 () else begin
        you_re_last := true;
        term()
      end)),
  (fun () → k2
      (fun _ → if !you_re_last then k3 () else begin
        you_re_last := true;
        term()
      end))
```

```
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;
    mutable value : sval_t list;
    mutable prev_value : sval_t list;
    mutable used : bool;
    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 ont été émis sur cet instant ou pour lesquels un processus est en attente dans la liste *present*. Ces signaux ont leur champ *used* à true. 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
  | Untilv of signal_t × (sval_t list → coop) × 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 [18] 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).

Polymorphisme des opérations sur les signaux Une version expérimentale permet d'utiliser directement des signaux portant des types différents. Le type des signaux est est alors paramétré et devient α *signal_t*. Ce paramètre se transmet aux types *coop* et *process* ce qui pose problème avec les listes de processus, ainsi qu'avec *used_signals*. Enfin, *contextelement* devient lui aussi paramétré ce qui empêche la constitution de *ctxt* comme une liste de *contextelement*.

Nous ne décrivons ici que très succintement les modifications qui consistent essentiellement en :
- la réalisation des opérations liées à la suspension d'un processus en attente d'un signal par les fonctions de suspension (*await* et consorts) elles-mêmes, le signal n'apparait alors plus dans le type *coop*,
- le stockage dans *used_signals* des *actions* à exécuter sur les signaux en fin d'instant plutôt que les signaux eux-même (la liste contient des valeurs de type $unit \rightarrow unit$ plutôt que *signal_t*),
- pour les constructions de contrôle n'utilisant pas la valeur du signal, le signal est transtypé par *Obj.magic* en un *unit signal_t* ce qui permet de former les listes de *contextelement*,
- pour *Untilv*, elle contient maintenant une fermeture qui teste la présence du signal et retourne le cas échéant (type *option*) le code du processus continuation (fermeture contenant la valeur du signal).

5 Performances

Nous comparons les performances, en OCaml code-octet et natif, de trois *runtime* de RML (Lco_ct, Lco_ct_class et Lk[3]) 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 et RML 1.07.1.

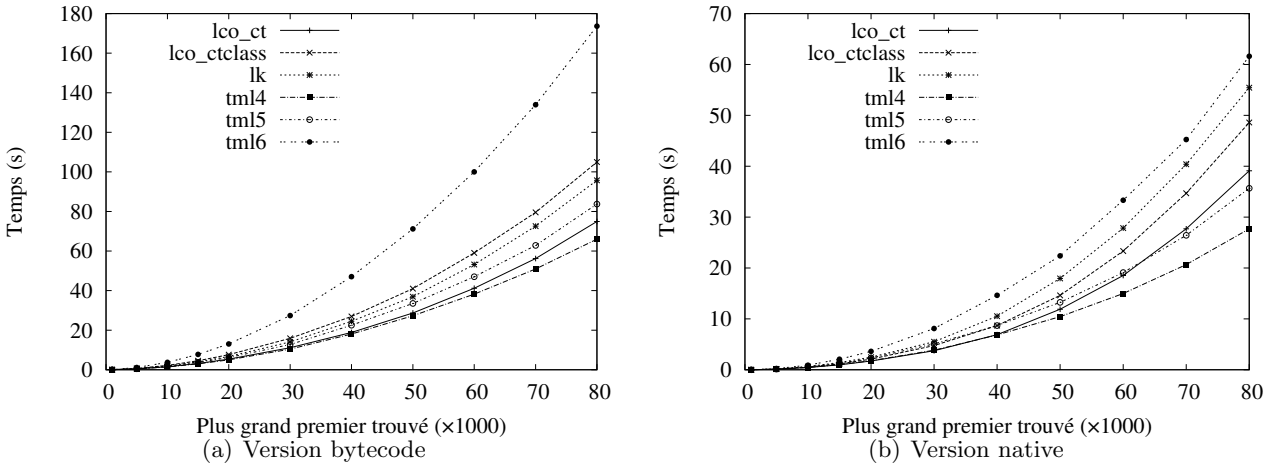
Le temps de calcul a été mesuré par la commande `time` et l'occupation mémoire a été fournie par le module *Gc* de OCaml.

Sieve Les figures 7 et 8 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.

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 %.

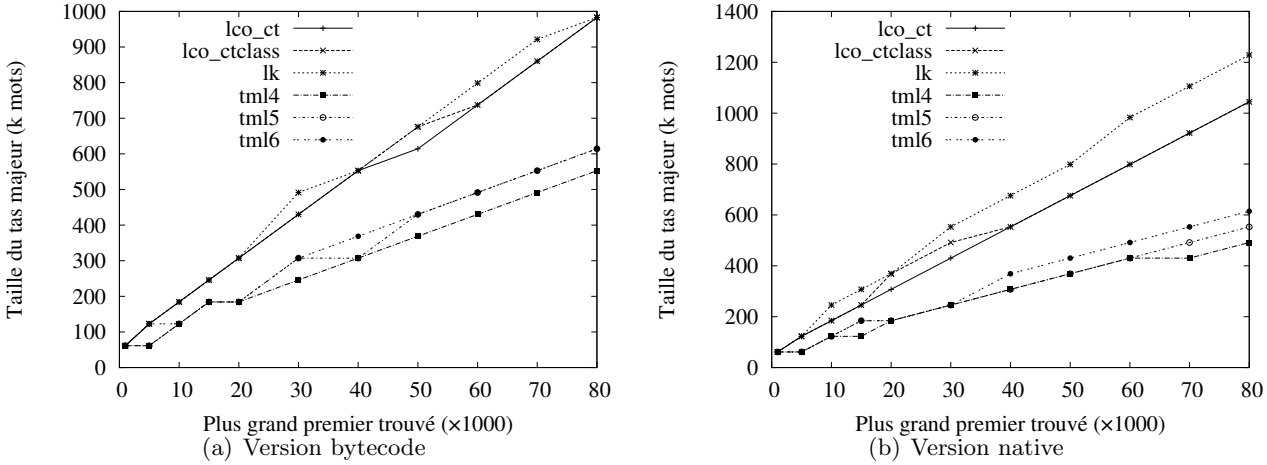
Elip Nous distinguons deux variantes de `elip` avec et sans nœuds préemptibles. La mise en œuvre de la préemption des nœud implique l'imbrication des constructions de contrôle, seul TML6 est alors utilisable (figure 11). Pour la version sans préemption, les constructions de contrôle sont sur un seul niveau et nous pouvons comparer les performances de TML5 et TML6 (figures 9 et 10). Les simulations ont été exécutées sur

[3]Lco et Lk sont deux façons d'encoder les processus. Les variantes de Lco correspondent à des implémentations différentes des primitives réactives.



(a) Version bytecode (b) Version native

FIG. 7 – Crible, temps d'exécution



(a) Version bytecode (b) Version native

FIG. 8 – Crible, espace mémoire

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`).

Globalement, la différence entre RML et TML est ici beaucoup moins nette. En ce qui concerne l'occupation mémoire, les implémentations sont toutes extrêmement proches[4]. 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 *runtime* RML ont le même comportement alors que des différences étaient perceptibles avec le crible.

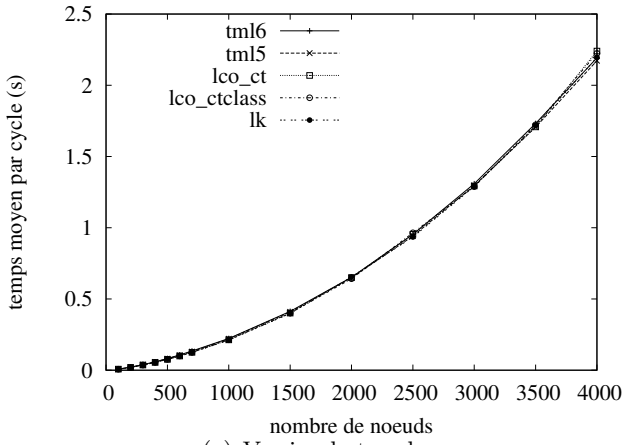
Par contre on peut relever un léger avantage en temps pour TML particulièrement dans les versions natives.

Impact de *par* **et** *tail_par* Sur le crible, une partie du gain en mémoire est due à l'utilisation de la fonction *tail_par* au lieu de *par*. RML n'a pas d'équivalent du *tail_par*. Le *runtime* Lco_n tente d'appliquer automatiquement cette optimisation mais n'a pas d'effet important sur le crible[5].

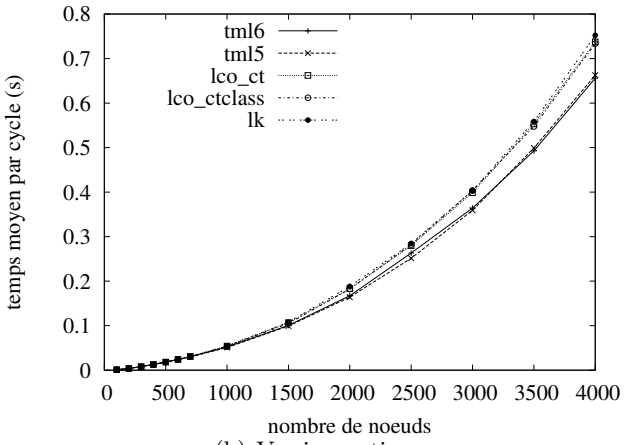
Le tableau 1 compare la taille du tas majeur en kilo mots pour les différentes réalisations du crible, dans le cas d'une recherche jusqu'à 80000. On voit que l'usage de *par* augmente la consommation mémoire de TML qui reste cependant sensiblement sous celle de RML.

[4]La présence ou pas de la préemption n'a aucun effet.

[5]Une compilation modulaire rend ceci difficile car le compilateur peut difficilement exclure à priori que le processus ne sera pas appelé dans un contexte nécessitant une synchronisation.

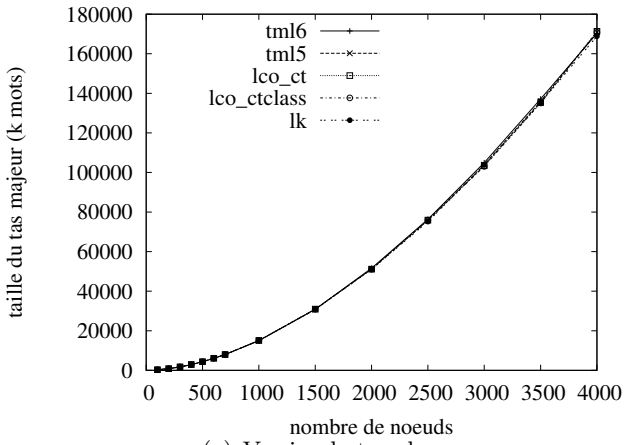


(a) Version bytecode

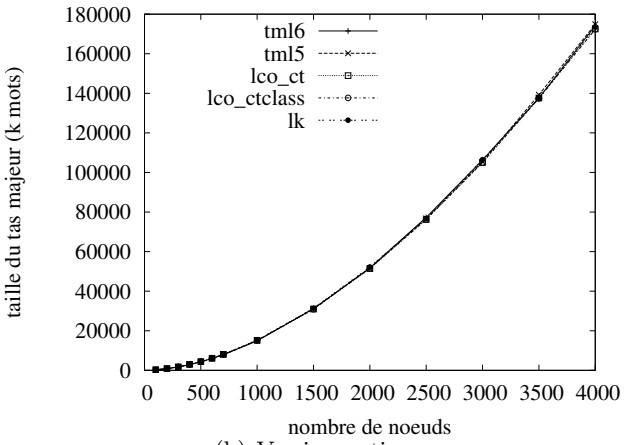


(b) Version native

FIG. 9 – Elip sans préemption, temps d'exécution

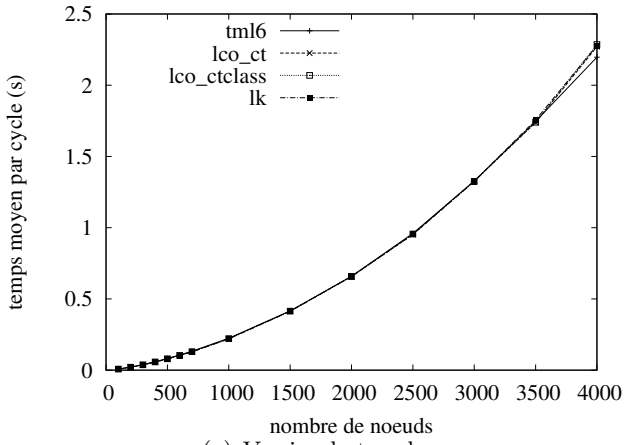


(a) Version bytecode

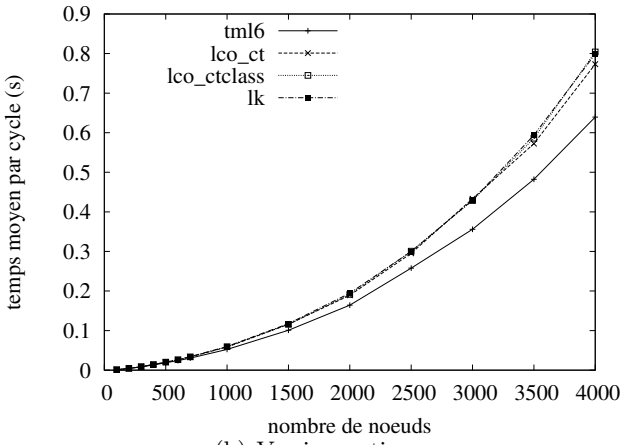


(b) Version native

FIG. 10 – Elip sans préemption, espace mémoire



(a) Version bytecode



(b) Version native

FIG. 11 – Elip avec préemption, temps d'exécution

	Lco_ct	Lco_ctclass	Lk	Lco_n	TML4	4 par	TML5	5 par	TML6	6 par
natif	1020	1020	1200	960	480	660	540	660	600	720
code-octet	960	960	960	960	540	660	600	720	600	780

TAB. 1 – Taille du tas pour le crible après 80000 nombres (k-mots)

Coût des constructions de contrôle Les mesures effectuées sur le crible suggeraient un impact important des constructions de contrôle. Les courbes ne montrent pourtant aucune différence pour Elip entre TML5 et TML6. L'utilisation de TML4 avec des constructions de contrôle "vides" (que nous ne montrons pas sur les figures) donne exactement les mêmes résultats que TML5 et TML6. Encore une fois, nous pensons que les différences étaient visibles sur le crible à cause de l'extrême légèreté des processus.

Taille du code généré Les tableaux 2, 3 et 4 montrent pour chaque fichier de Elip contenant des définitions de processus respectivement la taille des fichiers cmo (code-octet), la taille des fichiers .o (code natif) et le nombre d'instructions de la machine virtuelle générées pour TML et les deux encodages de RML. On observe que le code généré avec TML est sensiblement (sans être radicalement) plus compact. En fait le gain est certainement plus important si l'on considère uniquement le code des processus (les chiffres donnés incluent aussi tout le code des fonctions instantanées définies dans ces fichiers).

fichier	TML	Lk	Lk/TML (%)	Lco	Lco/TML (%)
draw.cmo	12884	15265	118	15344	119
dynamic.cmo	2166	2837	131	2473	114
node.cmo	2447	2799	114	2973	121
rml_graphics.cmo	1048	1183	113	1231	117
simul.cmo	4082	5876	144	5934	145
stat.cmo	5630	6581	117	6710	119
total	28257	34541	122	34665	123

Tab. 2 – Taille des cmo

fichier	TML	Lk	Lk/TML (%)	Lco	Lco/TML (%)
draw.o	22584	24036	106	25008	111
dynamic.o	4784	6164	129	5328	111
node.o	5192	5676	109	6100	117
rml_graphics.o	2636	2992	113	2900	110
simul.o	9300	12788	137	13036	140
stat.o	10816	11168	103	11416	105
total	55312	62824	114	63788	115

Tab. 3 – Taille des .o

fichier	TML	Lk	Lk/TML (%)	Lco	Lco/TML (%)
draw.ml	1760	2138	120	2039	116
dynamic.ml	227	376	166	269	118
node.ml	321	369	115	384	120
rml_graphics.ml	86	119	138	115	134
simul.ml	497	753	151	665	134
stat.ml	972	994	102	995	102
total	3863	4749	123	4467	116

Tab. 4 – Nombre d'instructions de la VM

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.

Style Le style trampoline repose sur les fonctions de première classe et nous permet de "plaquer" les processus dans la catégorie syntaxique des fonctions. En l'absence de fonctions de première classe, le code

des processus doit être contenu dans des structures de données et le programmeur se trouve contraint de manipuler ce qui est essentiellement l'arbre abstrait des processus. C'est le cas par exemple de la bibliothèque Sugar Cubes pour Java [7]. Dans ce cas un langage spécialisé se justifie pleinement par le confort syntaxique qu'il offre. Les exemples que nous avons présentés nous conduisent à penser que le style trampoline est raisonnablement utilisable avec un peu de pratique, et que l'aspect syntaxique à lui seul ne justifie pas pleinement une extension de langage.

Notons que l'on peut aussi passer du style trampoline au style monadique en définissant simplement un opérateur infixe "bind" par let (>>=) *inst k* = *inst k*. On obtient alors une formulation très similaire aux systèmes de threads basés sur les monades comme `Lwt` [19]. 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 [20]. 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 primitive classique de capture de continuation `call/cc` est fournie par une bibliothèque [21] mais celle ci est présentée comme une implémentation "très naïve" souffrant de performances "terribles", il ne serait donc pas raisonnable de l'utiliser. La bibliothèque caml-shift [22] implémente la capture de continuations partielles[6] mais elle n'est disponible qu'en code-octet et les performances sont sensiblement moins bonnes : quelques tests sur le crible ont montré un triplement de l'occupation mémoire et du temps de calcul.

La programmation par événement est une autre approche possible pour la gestion d'un ordonnancement coopératif au niveau de l'application. Il y a en fait de fortes similarités avec le style trampoline puisque le *handler* placé en attente d'un événement peut être vu comme une continuation. La bibliothèque OCamlnet [23] propose en particulier un module `equeue` mais elle impose un ordonnancement naïf (les événements sont présentés systématiquement à tous les *handler* jusqu'à ce que l'un deux l'accepte) qui ne serait pas adapté à notre contexte.

Performances Les tests que nous avons réalisés ont montré que les performances sont au minimum honorables, et parfois avantageuses en mémoire. Du point de vue des performances cette bibliothèque pourrait être préférable à ReactiveML pour des applications utilisant un très grand nombre de processus très simples.

Perspectives 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* [8] propose un modèle de programmation concurrente inspiré du modèle réactif, et reposant sur des passages de jetons entre threads système. Si le modèle de programmation est plus simple, les inconvénients d'une implémentation système (usage de ressources) persistent. 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.

Le code de la bibliothèque (source OCaml et littéraire) et celui des exemples est disponible à [24]. Les extraits de code apparaissant dans ce document ont été formatés avec l'outil ocamlweb (modifié pour traiter ReactiveML).

Références

[1] Halbwachs (Nicolas). – *Synchronous Programming of Reactive Systems.* – Kluwer Academic Publishers, 1993.

[6] Aussi appelées continuations composables ou continuations délimitées.

[2] Boussinot (F.) et de Simone (R.). – The ESTEREL language. *Proceedings of the IEEE*, vol. 79, n 9, Sep 1991, pp. 1293–1304.

[3] Halbwachs (N.), Caspi (P.), Raymond (P.) et Pilaud (D.). – The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, vol. 79, n 9, Sep 1991, pp. 1305–1320.

[4] LeGuernic (P.), Gautier (T.), Le Borgne (M.) et Le Maire (C.). – Programming real-time applications with signal. *Proceedings of the IEEE*, vol. 79, n 9, Sep 1991, pp. 1321–1336.

[5] Boussinot (Frédéric). – *La programmation réactive. Application aux systèmes communicants.* – Masson et CNET-ENST, 1996.

[6] Boussinot (Frédéric). – Reactive C : An extension of C to program reactive systems. *Software : Practice and Experience*, vol. 21, n 4, avril 1991, pp. 401–428.

[7] Boussinot (Frédéric) et Susini (Jean-Ferdy). – Java threads and sugarcubes. *Software : Practice and Experience*, vol. 30, n 5, avril 2000, pp. 545–566.

[8] Boussinot (Frédéric). – FairThreads : mixing cooperative and preemptive threads in C. *Concurrency and Computation : Practice and Experience*, vol. 18, n 5, avril 2006, pp. 445–469. – Also available as a research report.

[9] INRIA. – The Caml language. `http://caml.inria.fr`

[10] Mandel (Louis) et Pouzet (Marc). – Reactive ML. `http://www.reactiveml.org`

[11] Reynolds (John C.). – The discoveries of continuations. *LISP and Symbolic Computation*, vol. 6, n 3–4, 1993, pp. 233–247. `citeseer.ist.psu.edu/reynolds93discoveries.html`

[12] Steele (Guy L) et Sussman (Gerald J). – *Lambda : The Ultimate Imperative.* – Rapport technique n AIP-353, Cambridge, MA, USA, Massachusetts Institute of Technology, 1976.

[13] Thielecke (Hayo). – Continuations, functions and jumps. *Logic Column 8, SIGACT News*, juillet 1999.

[14] Ganz (Steven E.), Friedman (Daniel P.) et Wand (Mitchell). – Trampolined style. *In : International Conference on Functional Programming*, pp. 18–27. `http://citeseer.ist.psu.edu/ganz99trampolined.html`

[15] Taha (Walid) et al. – MetaOCaml : A compiled, type-safe, multi-stage programming language. – Web page. `http://www.metaocaml.org`

[16] Kahn (G.) et MacQueen (D. B.). – Coroutines and networks of parallel processes. *In : Information processing*, pp. 993–998. – Toronto, août 1977.

[17] Mandel (Louis) et Benbadis (Farid). – Simulation of mobile ad hoc network protocols in ReactiveML. *In : Proceedings of Synchronous Languages, Applications, and Programming (SLAP'05).* – Edinburgh, Scotland, avril 2005.

[18] Deleuze (Christophe). – *Programmation réactive en OCaml – Implémentation de la bibliothèque TML.* – Rapport technique, LCIS, 2009.

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

[20] Friedman (D. P.). – Applications of continuations. – Invited Tutorial, Fifteenth Annual ACM Symposium on Principles of Programming Languages, janvier 1988. `http://www.cs.indiana.edu/~dfried/appcont.pdf`

[21] Leroy (Xavier). – OCaml-callcc : call/cc for OCaml. – OCaml library. `http://pauillac.inria.fr/~xleroy/software.html`

[22] Sabry (Amr), chieh Shan (Chung) et Kiselyov (Oleg). – Native delimited continuations in (byte-code) OCaml. – OCaml library, 2008. `http://okmij.org/ftp/Computation/Continuations.html#caml-shift`

[23] Stolpmann (Gerd). – Ocamlnet. `http://projects.camlcity.org/projects/ocamlnet.html`

[24] Deleuze (Christophe). – Bibliothèque TML. – Page web. `http://christophe.deleuze.free.fr/tml/`