Exécution efficace de programmes ReactiveML

Louis Mandel[1,3] & Cédric Pasteur[2,3]

1: Collège de France
2: École normale supérieure
3: INRIA Paris-Rocquencourt

Résumé

ReactiveML est un langage dédié à la programmation de systèmes combinant des parties algorithmiques et réactives. Il s'agit d'une extension de ML avec des constructions pour la concurrence inspirées des langages synchrones. Celles-ci permettent d'obtenir une très grande expressivité, mais leur implantation efficace représente un défi.

Dans cet article, nous présentons l'implantation de ReactiveML, de la compilation à l'implantation du moteur d'exécution en OCaml. Nous décrivons également une implantation parallèle en mémoire partagée du moteur d'exécution utilisant le vol de tâches. L'approche choisie permet d'obtenir une exécution efficace même en présence de structures de contrôle complexes. Elle s'étend simplement au cas parallèle avec des résultats expérimentaux prometteurs.

1. Introduction

Lors du dernier concours ICFP, [1] la tâche consistait à deviner des fonctions en posant des questions à un serveur de jeux. On pouvait envoyer les requêtes suivantes au serveur : demander un nouveau problème ; demander la valeur de la fonction à deviner sur 256 entrées ; soumettre une solution. Lors de la soumission d'une solution, si la fonction proposée n'était pas correcte, le serveur fournissait un contre-exemple. Les concurrents avaient cinq minutes pour résoudre chaque problème et il était possible de faire au plus cinq requêtes toutes les vingt secondes.

Une architecture possible pour la programmation d'un joueur est la suivante : (1) un processus consacré à la génération de fonctions qui satisfont la spécification de la fonction à deviner et (2) un processus chargé de la communication avec le serveur de jeux. Le processus de communication demande périodiquement plus d'informations sur la fonction à deviner et dès qu'une solution est proposée par le processus de génération de fonctions, il soumet cette solution au serveur de jeux.

On observe ici la description d'un système qui contient à la fois des parties algorithmiques (génération de fonctions) et réactives (synchronisation des processus et communications avec le serveur). Le langage ReactiveML est dédié à la programmation de ce type de systèmes. Il combine l'expressivité d'un langage à la ML (ici un sous-ensemble d'OCaml) avec des constructions de programmation synchrone pour parler du temps et des événements.

Par rapport aux bibliothèques [2] de threads préemptifs comme le module `Thread` d'OCaml, [3] de threads coopératifs comme Lwt, [4] Async, [5] Muthreads [6] ou de programmation événementielle comme Equeue, [7] ReactiveML propose des constructions de communication et de synchronisation plus

1. `http://icfpc2013.cloudapp.net`
2. Nous citons ici des bibliothèques OCaml, mais la situation est similaire dans la plupart des langages généralistes.
3. `http://caml.inria.fr/pub/docs/manual-ocaml-4.01/libthreads.html`
4. `http://ocsigen.org/lwt`
5. `https://ocaml.janestreet.com/ocaml-core/latest/doc/async_core`
6. `http://christophe.deleuze.free.fr/muthreads`
7. `http://www.camlcity.org/archive/programming/equeue.html`

expressives. Elles permettent en particulier de préempter et de suspendre l'exécution d'un processus à l'émission d'un signal. Ces constructions réactives sont héritées du langage synchrone Esterel [4, 2] qui est dédié à la programmation du contrôle de systèmes embarqués temps-réel critiques.

Dans cet article, nous présentons l'implantation de ReactiveML. Comme habituellement dans les langages fonctionnels, la concurrence est réalisée à l'aide de continuations [20]. L'originalité de l'approche vient de l'utilisation d'une structure de données annexe pour la gestion de l'activation des processus en présence de préemption et de suspension.

Nous commençons par présenter intuitivement la sémantique de ReactiveML à travers des exemples (partie 2). La partie 3 présente la compilation de ReactiveML qui est fondée sur un langage intermédiaire à base de continuations. Le moteur d'exécution est ensuite décrit sans préemption ni suspension (partie 4), puis étendu avec ces constructions (partie 5). La partie 6 présente une implantation parallèle du moteur d'exécution. Enfin, nous concluons par une discussion sur les travaux similaires (partie 7).

2. Présentation de ReactiveML

Reprenons l'exemple du concours ICFP 2013. Les fonctions à deviner prennent un argument qui est un entier 64 bits et le corps de la fonction est composé des constantes entières 0 et 1, de variables, d'un test à zéro, d'opérations unaires, binaires et d'un itérateur. Comme en OCaml, l'arbre de syntaxe abstraite de ce langage peut être représenté à l'aide de types structurés de la façon suivante :

```
type program = { input : ident; expr : expr }
and ident = { name : string; mutable value : Int64.t; }
and expr =
  | Const of Int64.int64
  | Var of ident
  | If_Zero of expr * expr * expr
  ...
```

On peut ensuite définir des fonctions de manipulation de ces valeurs :

```
let rec eval_expr e = match e with
  | Const c -> c
  | Var v -> v.value
  | If_Zero (e1, e2, e3) ->
      if eval_expr e1 = Int64.zero then eval_expr e1 else eval_expr e2
  ...
```

La particularité de ReactiveML vient de ses constructions réactives. On définit ci-dessous un processus `guesser` qui, à partir de la description `pb` d'un problème (un ensemble d'opérateur et une taille de terme) émet sur le signal `guess` des programmes qui respectent la spécification lue sur le signal `spec`. La fonction `random_program` génère aléatoirement des valeurs de type `program` à partir de la définition du problème à résoudre et la fonction `check` vérifie si un programme respecte une spécification.

```
let process guesser pb spec guess =
  loop
    let f = random_program pb in
    if check f (last ?spec) then emit guess f;
    pause
  end
```

ReactiveML est fondé sur le modèle synchrone où le temps est défini comme une succession d'instants logiques. Le mot clé `process` indique que l'exécution de `guesser` peut durer plusieurs instants et

l'expression `pause` marque l'attente de l'instant suivant. La communication entre les processus se fait par des signaux transportant des valeurs. Ici, le corps du processus est une boucle infinie `loop`/`end` qui génère aléatoirement une fonction `f`, récupère sur le signal `spec` avec l'expression `last ?spec` les couples d'entrées/sorties que cette fonction doit respecter et émet `f` sur le signal `guess` si elle convient.

De façon similaire au processus `guesser`, on peut définir un processus `increase_spec` qui, à chaque instant, demande au serveur de jeux plus d'informations sur la fonction à deviner (le module `Webapi` se charge des communications réseaux).

```
let process increase_spec pb spec =
  for i = 0 to Array.length inputs_array - 1 do
    let inputs = inputs_array.(i) in
    let outputs = Webapi.eval pb.id inputs in
    emit spec (inputs, outputs);
    pause
  done
```

On peut maintenant définir le processus `communicator` qui gère les communications avec le serveur. Il est composé de deux parties qui s'exécutent en parallèle. La première, lignes 2 à 7, est une boucle qui attend sur le signal `guess` une fonction `f` calculée par le processus `guesser` et demande au serveur de jeux si cette fonction est une solution au problème. Si c'est le cas, le signal `finished` est émis. Sinon, le contre-exemple fourni par le serveur de jeux est envoyé sur le signal `spec`. La seconde partie du processus, lignes 9 à 18, demande périodiquement plus d'informations au serveur sur la fonction à deviner en exécutant le processus `increase_spec` (la construction `run` marque l'exécution de processus). Pour limiter le nombres de requêtes envoyées au serveur, le processus `increase_spec` est exécuté dans une construction `do`/`when` qui exécute son corps uniquement aux instants où le signal de contrôle (`tick`) est émis. La boucle ligne 15 gère l'émission du signal `tick` toutes les quatre secondes (le processus `wait d` ne fait rien pendant `d` secondes puis termine). Cette boucle est comprise dans une construction `do`/`until` qui préempte l'exécution lorsque le signal `guess` est émit. Enfin, cette construction `do`/`until` est elle même comprise dans une boucle pour réémettre le signal `tick` périodiquement. Lors de la déclaration du signal `tick` (ligne 10), il faut donner une valeur par défaut et une fonction de combinaison. En effet, la valeur d'un signal est le résultat de l'itération (un *fold*) de la fonction de combinaison sur l'ensemble des valeurs émises sur le signal en partant de la valeur par défaut.

```
1  let process communicator pb guess spec finished =
2    loop
3      await guess (f) in
4      match Webapi.guess pb.id f with
5      | Guess_win -> emit finished ()
6      | Guess_mismatch (input, output, _) -> emit spec ([|input|], [|output|])
7    end
8    ||
9    begin
10     signal tick default () gather (fun x y -> ()) in
11     do run increase_spec pb spec when tick done
12     ||
13     loop
14       do
15         loop emit tick (); run wait 4.0 end
16       until guess -> run wait 4.0 done
17     end
18   end
```

Enfin, le processus `solve` essaye de résoudre un problème donné en paramètre en moins de cinq minutes. Pour cela, il exécute le processus `communicator` et le processus `guesser` en parallèle. Pour gérer la terminaison, ces deux processus sont exécutés dans une construction `do/until` contrôlée par le signal `finished`. Ce signal est soit émis par le processus `communicator`, soit par la troisième branche parallèle du processus `solve` au bout de cinq minutes.

```
let process solve pb =
  signal guess default dummy_prog gather (fun x y -> x) in
  signal spec memory [] gather (fun x y -> x :: y) in
  signal finished default () gather (fun x y -> ()) in
  do
    run communicator pb guess spec finished
    ||
    await spec; run guesser pb spec guess
    ||
    run wait (5. *. 60.); emit finished ()
  until finished done
```

On peut constater que la valeur par défaut du signal `spec` est introduite avec le mot clé `memory` au lieu de `default`. Cela signifie qu'au lieu de partir de la valeur par défaut pour calculer la valeur du signal, il faut partir de la dernière valeur du signal. Ainsi, le signal `spec` ne fait qu'accumuler les informations sur la fonction à deviner.

3. Compilation

La compilation de ReactiveML passe par un langage intermédiaire appelé $\mathcal{L}_k$ fondé sur l'utilisation de continuations. Ce langage a été introduit dans [16]. La principale opération pour la traduction de ReactiveML en $\mathcal{L}_k$ est une transformation CPS (*Continuation Passing Style*) partielle. Pour chaque opération potentiellement bloquante comme l'instruction `pause` ou l'attente d'un signal, on crée une continuation contenant la suite du programme. Le reste du code demeure inchangé. Nous nous intéressons dans un premier temps au langage sans préemption ni suspension. Nous verrons dans la partie 5 comment ajouter ces structures de contrôle.

ReactiveML On considère pour cette traduction un noyau non minimal de ReactiveML (sans préemption ni suspension) défini par :

$$\begin{aligned} e ::= {} & x \mid c \mid (e,e) \mid \lambda x.e \mid e\ e \mid \texttt{rec}\ x = e \mid \texttt{process}\ e \mid \texttt{run}\ e \mid \texttt{pause} \\ & \mid \texttt{let}\ x = e\ \texttt{and}\ x = e\ \texttt{in}\ e \mid e; e \mid \texttt{signal}\ x\ \texttt{default}\ e\ \texttt{gather}\ e\ \texttt{in}\ e \\ & \mid \texttt{emit}\ e\ e \mid \texttt{await immediate}\ e \mid \texttt{await}\ e(x)\ \texttt{in}\ e \mid \texttt{present}\ e\ \texttt{then}\ e\ \texttt{else}\ e \end{aligned}$$

Il s'agit d'un lambda-calcul avec appel par valeur, étendu avec la création (`process`) et le lancement (`run`) de processus, l'attente du prochain instant (`pause`), la définition parallèle (`let/and`), la déclaration de signaux (`signal`), l'émission d'un signal (`emit`), l'attente immédiate d'un signal (`await immediate`), l'attente de la valeur d'un signal (`await`) et le test de présence d'un signal (`present`). L'expression `await immediate` s termine instantanément lorsque le signal s est émis. L'expression `present` s `then` e_1 `else` e_2 exécute e_1 instantanément si le signal s est présent ou e_2 à l'instant suivant s'il est absent. Le délai de réaction à l'absence, hérité du langage ReactiveC [6], permet d'éviter les contradictions sur la présence des signaux. À partir de ce noyau, on peut encoder la plupart des autres constructions du langage. On notera _ les variables qui n'apparaissent pas libres dans le corps du `let` et () l'unique valeur de type `unit` :

$$e_1 \,||\, e_2 \triangleq \texttt{let}\ \_ = e_1\ \texttt{and}\ \_ = e_2\ \texttt{in}\ ()$$

$$\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \triangleq \texttt{let}\ x = e_1\ \texttt{and}\ \_ = ()\ \texttt{in}\ e_2$$

$$\texttt{let}\ f\ x_1\ \ldots x_p = e_1\ \texttt{in}\ e_2 \triangleq \texttt{let}\ f = \lambda x_1.\ldots\lambda x_p.e_1\ \texttt{in}\ e_2$$
$$\texttt{let rec}\ f\ x_1\ \ldots x_p = e_1\ \texttt{in}\ e_2 \triangleq \texttt{let}\ f = (\texttt{rec}\ f = \lambda x_1.\ldots\lambda x_p.e_1)\ \texttt{in}\ e_2$$
$$\texttt{let process}\ f\ x_1\ \ldots x_p = e_1\ \texttt{in}\ e_2 \triangleq \texttt{let}\ f = \lambda x_1.\ldots\lambda x_p.\texttt{process}\ e_1\ \texttt{in}\ e_2$$
$$\texttt{let rec process}\ f\ x_1\ \ldots x_p = e_1\ \texttt{in}\ e_2 \triangleq \texttt{let}\ f = (\texttt{rec}\ f = \lambda x_1.\ldots\lambda x_p.\texttt{process}\ e_1)\ \texttt{in}\ e_2$$

Ce noyau est non minimal car il contient par exemple la construction $e_1; e_2$ qui peut être encodée comme $\texttt{let}\ \_ = e_1\ \texttt{in}\ e_2$. On peut en effet proposer une implantation directe de cette construction bien plus efficace que son encodage.

Le langage intermédiaire $\mathcal{L}_k$ Le langage $\mathcal{L}_k$ est défini formellement par :

$$\begin{aligned}
k ::= {} & \texttt{end} \mid \kappa \mid e_i.k \mid \texttt{present}\ e_i\ \texttt{then}\ k\ \texttt{else}\ k \mid \texttt{run}\ e_i.k && \text{(continuations)}\\
& \mid \texttt{signal}\ x\ \texttt{default}\ e_i\ \texttt{gather}\ e_i\ \texttt{in}\ k \mid \texttt{await immediate}\ e_i.k \mid \texttt{await}\ e_i(x)\ \texttt{in}\ k\\
& \mid \texttt{split}\ (\lambda x.(k,k)) \mid \texttt{join}\ x\ i.k \mid \texttt{def}\ x\ \texttt{and}\ x\ \texttt{in}\ k \mid \texttt{bind}\ \kappa = k\ \texttt{in}\ k\\
e_i ::= {} & x \mid c \mid (e_i, e_i) \mid \lambda x.e_i \mid \texttt{rec}\ x = e_i \mid \texttt{process}\ \Lambda\kappa.k && \text{(expressions instantanées)}\\
& \mid \texttt{signal}\ x\ \texttt{default}\ e_i\ \texttt{gather}\ e_i\ \texttt{in}\ e_i \mid \texttt{emit}\ e_i\ e_i
\end{aligned}$$

Le langage $\mathcal{L}_k$ distingue les continuations k et les expressions instantanées e_i qui correspondent aux expressions ML. La continuation `end` marque la fin du programme, alors que κ est une variable qui peut être substituée par une continuation k. L'expression $e_i.k$ évalue l'expression instantanée e_i puis donne sa valeur à la continuation k. Les expressions `split`, `join` et `def` servent à encoder la composition parallèle synchrone. La construction `split` commence l'exécution du parallèle, le `join` synchronise la terminaison des deux branches et la construction `def` récupère les valeurs retournées par les branches avant d'exécuter la continuation. La variable x introduite dans le `split` et utilisée par le `join` est une variable partagée par les deux branches qui permet de les synchroniser. Le paramètre i, qui ne peut valoir que 1 ou 2, différencie la branche gauche ($i = 1$) de la branche droite ($i = 2$) de la composition parallèle. La construction `bind` permet de nommer les continuations. Les expressions instantanées e_i sont similaires à ReactiveML. La principale différence est que la définition d'un processus prend en argument sa continuation κ introduite par le lieur Λ.

Identification des expressions instantanées Avant de pouvoir traduire le code ReactiveML en $\mathcal{L}_k$, le compilateur doit distinguer les expressions instantanées et les expressions réactives. On utilise pour cela un système de types simple qui permet de garantir certaines propriétés de bonne formation des expressions. En particulier, on souhaite interdire l'utilisation d'expressions réactives comme `pause` à l'intérieur des fonctions. On ne peut les utiliser qu'à l'intérieur d'un processus. Ainsi, la transformation CPS ne concerne que les processus, alors que les fonctions sont conservées telles quelles. Cela permet d'améliorer les performances du code généré, puisque le code OCaml n'est pas modifié.

Cette analyse est définie figure 1 par un jugement de la forme $k \vdash e$ où $k \in \{0, 1\}$. Le prédicat $0 \vdash e$ signifie que e est une expression instantanée (on dit aussi combinatoire). $1 \vdash e$ signifie que e est une expression réactive (on dit aussi séquentielle ou à mémoire en reprenant la terminologie des circuits numériques synchrones). Nous ne discuterons pas des choix faits dans la définition de ce prédicat, puisque cela est déjà fait dans [16]. Nous pouvons tout de même rappeler les points les plus importants :

- $k \vdash e$ signifie que $0 \vdash e$ et $1 \vdash e$. L'expression peut donc être utilisée dans n'importe quel contexte, aussi bien dans une expression instantanée que dans une expression réactive. C'est par exemple le cas des variables, des constantes ou encore de l'application.
- Le corps d'une fonction doit être une expression instantanée (Abs), alors que celui d'un processus peut être une expression réactive (ProcAbs).

$$\frac{}{k \vdash x} \qquad \frac{}{k \vdash c} \qquad \frac{0 \vdash e_1 \qquad 0 \vdash e_2}{k \vdash (e_1, e_2)} \qquad (\textsc{Abs})\frac{0 \vdash e_1}{k \vdash \lambda x.e_1} \qquad \frac{0 \vdash e_1 \qquad 0 \vdash e_2}{k \vdash e_1\ e_2} \qquad \frac{0 \vdash e_1}{k \vdash \texttt{rec}\ x = e_1}$$

$$(\textsc{ProcAbs})\frac{1 \vdash e_1}{k \vdash \texttt{process}\ e_1} \qquad \frac{0 \vdash e_1}{1 \vdash \texttt{run}\ e_1} \qquad \frac{}{1 \vdash \texttt{pause}} \qquad \frac{k \vdash e_1 \qquad k \vdash e_2 \qquad k \vdash e_3}{k \vdash \texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e_3}$$

$$\frac{k \vdash e_1 \qquad k \vdash e_2}{k \vdash e_1; e_2} \qquad \frac{k \vdash e_1 \qquad k \vdash e_2 \qquad k \vdash e}{k \vdash \texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e} \qquad \frac{0 \vdash e_s \qquad 0 \vdash e_1}{k \vdash \texttt{emit}\ e_s\ e_1}$$

$$\frac{0 \vdash e}{1 \vdash \texttt{await immediate}\ e} \qquad \frac{0 \vdash e_1 \qquad 1 \vdash e_2}{1 \vdash \texttt{await}\ e_1(x)\ \texttt{in}\ e_2} \qquad \frac{0 \vdash e_s \qquad 1 \vdash e_1 \qquad 1 \vdash e_2}{1 \vdash \texttt{present}\ e_s\ \texttt{then}\ e_1\ \texttt{else}\ e_2}$$

FIGURE 1 – Bonne formation des expressions

Traduction de ReactiveML vers $\mathcal{L}_k$ La traduction vers $\mathcal{L}_k$ est définie par une fonction $C_k[e]$ paramétrée par une continuation k. Elle prend en argument une expression ReactiveML e et renvoie une continuation dans le langage $\mathcal{L}_k$. Cette fonction est définie sur la figure 2. Elle utilise la fonction $C[e]$ de traduction des expressions instantanées. Celle-ci ne prend pas en paramètre de continuation puisque la transformation CPS ne concerne que les expressions réactives. Commentons ces fonctions de traduction.

- Si l'expression e est instantanée, c'est-à-dire si $0 \vdash e$, alors on ne doit pas lui appliquer la transformation CPS. On la traduit donc en une expression instantanée $C[e]$, que l'on évalue avant de donner sa valeur à la continuation k.
- Il n'y a pas de séquence en $\mathcal{L}_k$, puisque celle-ci est encodée avec les continuations. Dans le cas de $e_1; e_2$, on traduit d'abord e_2 en lui donnant la continuation k. On traduit ensuite e_1 en utilisant la traduction de e_2 comme continuation.
- Pour le `present`, on utilise la construction `bind` pour définir la continuation partagée par les deux branches. Ce partage est nécessaire pour ne pas recopier la continuation dans les deux branches, ce qui pourrait aboutir à une explosion exponentielle de la taille du code généré.
- La traduction de la construction `let/and/in` utilise la construction `split` pour déclencher l'exécution des deux branches. La continuation de chacune des branches est une instruction `join` qui attend la terminaison de l'autre branche. La continuation des `join` est la continuation κ qui est une instruction `def` qui récupère instantanément les valeurs calculées par les deux branches. On utilise la construction `bind` pour partager κ entre les deux branches.

La traduction des expressions instantanées, définie sur la figure 2b, consiste à paramétrer les processus par une continuation κ, puis à traduire le corps du processus en utilisant la fonction $C_\kappa[.]$, c'est-à-dire en utilisant κ comme continuation du corps du processus. Les autres règles appliquent cette transformation de façon structurelle dans toute l'expression.

Traduction en OCaml La traduction de $\mathcal{L}_k$ en OCaml est immédiate. On associe à chaque construction de $\mathcal{L}_k$ une fonction OCaml (un combinateur). On représente une continuation par une valeur de type `'a step = 'a -> unit`, qui est une fonction attendant une valeur de type `'a`. On représente les expressions instantanées par des glaçons de type `unit -> 'a`. Ainsi, la construction $e.k$ est représentée par le combinateur `rml_compute` défini par :

```
let rml_compute e k = (fun _ -> k (e ()))
val rml_compute : (unit -> 'a) -> 'a step -> 'b step
```

Le combinateur prend en argument une expression instantanée `e` et une continuation `k`. Il renvoie une fonction de transition qui évalue l'expression `e` en lui donnant la valeur `()`, puis donne le résultat à la

$$C_k[e] = C[e].k \quad \text{si} \quad 0 \vdash e \qquad C_k[\texttt{run}\ e] = \texttt{run}\ C[e].k \qquad C_k[e_1; e_2] = C_{C_k[e_2]}[e_1]$$

$$C_k[\texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2] = \texttt{bind}\ \kappa = k\ \texttt{in}\ \texttt{present}\ C[e]\ \texttt{then}\ C_\kappa[e_1]\ \texttt{else}\ C_\kappa[e_2]$$

$$C_k[\texttt{await immediate}\ e] = \texttt{await immediate}\ C[e].k$$

$$C_k[\texttt{await}\ e_1(x)\ \texttt{in}\ e_2] = \texttt{await}\ C[e_1](x)\ \texttt{in}\ C_k[e_2]$$

$$C_k[\texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e] = \texttt{signal}\ x\ \texttt{default}\ C[e_1]\ \texttt{gather}\ C[e_2]\ \texttt{in}\ C_k[e]$$

$$C_k[\texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e] = \texttt{bind}\ \kappa = (\texttt{def}\ x_1\ \texttt{and}\ x_2\ \texttt{in}\ C_k[e])\ \texttt{in}\ \texttt{split}\ (\lambda y.(C_{\texttt{join}\ z\ 1.\kappa}[e_1], C_{\texttt{join}\ z\ 2.\kappa}[e_2]))\ \text{où } z \text{ frais}$$

(a) Traduction des expressions réactives

$$C[x] = x \qquad C[c] = c \qquad C[(e_1, e_2)] = (C[e_1], C[e_2]) \qquad C[e_1\ e_2] = C[e_1]\ C[e_2]$$

$$C[\lambda x.e] = C[\lambda x.C[e]] \qquad C[\texttt{rec}\ x = e] = \texttt{rec}\ x = C[e] \qquad C[\texttt{process}\ e] = \texttt{process}\ \Lambda\kappa.C_\kappa[e]$$

$$C[\texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e] = \texttt{signal}\ x\ \texttt{default}\ C[e_1]\ \texttt{gather}\ C[e_2]\ \texttt{in}\ C[e]$$

$$C[\texttt{emit}\ e_1\ e_2] = \texttt{emit}\ C[e_1]\ C[e_2]$$

(b) Traduction des expressions instantanées

FIGURE 2 – Traduction de ReactiveML vers $\mathcal{L}_k$

continuation `k`. Les combinateurs correspondant aux autres expressions ont une forme similaire :

```
val rml_pause : unit step -> 'a step
val rml_present : (unit -> ('a, 'b) event) -> unit step -> unit step -> 'c step
val rml_await_immediate : (unit -> ('a, 'b) event) -> unit step -> 'c step
val rml_await_all : (unit -> ('a, 'b) event) -> ('b -> unit step) -> 'c step
...
```

La suite de l'article décrit l'implantation de ces différents combinateurs.

4. Implantation du moteur d'exécution

Nous décrivons maintenant l'implantation du moteur d'exécution en OCaml. On trouvera une présentation formelle de la sémantique de $\mathcal{L}_k$ qui suit les mêmes principes dans la partie 7.2 de [16].

Principe de l'exécution Le moteur d'exécution de ReactiveML est un ordonnanceur de tâches. Il utilise un ordonnancement coopératif, où chaque processus doit volontairement rendre la main à l'ordonnanceur pour laisser les autres processus s'exécuter. On dispose d'un ensemble $\mathcal{C}$ de continuations à exécuter dans laquelle l'ordonnanceur vient piocher (on représente cet ensemble par une liste). Certains combinateurs, comme par exemple la composition parallèle, ajoutent des continuations dans cette liste. Il existe également une seconde liste appelée *next* qui contient les processus à exécuter à l'instant suivant. On y ajoute typiquement la continuation de la construction `pause`. L'exécution d'un instant suit l'algorithme suivant :

1. On exécute un instant du programme. Pour cela, on exécute tous les processus dans la liste $\mathcal{C}$ jusqu'à ce qu'elle soit vide.

2. On exécute ensuite la fin de l'instant. On réveille alors les processus qui testent la présence d'un signal absent et ceux qui attendent la valeur d'un signal. On transfère ensuite les processus contenus dans la liste *next*, en attente de l'instant suivant, jusque dans la liste $\mathcal{C}$ des continuations à exécuter. On commence l'exécution de l'instant suivant en revenant à la première étape.

Ces deux phases sont présentes dans la sémantique opérationnelle de ReactiveML [17]. Elle est décrite sous la forme de deux réductions, correspondant respectivement à l'exécution de l'instant et à la fin de l'instant.

Le moteur d'exécution présente deux caractéristiques importantes qui viennent s'ajouter à cet algorithme de base :

— Pour obtenir un interprète efficace, il est capital de faire de *l'attente passive* des signaux. Cela signifie qu'un processus en attente d'un signal ne doit être activé que lorsque le signal est présent : il ne doit pas être réveillé pour vérifier la présence du signal plusieurs fois par instant, ni aux instants où le signal est absent. Pour éviter l'attente active, on associe de façon classique à chaque signal une liste des continuations en attente de ce signal. On réveille ces continuations uniquement lorsque le signal est émis.

— L'autre composante du moteur d'exécution, qui est aussi la plus complexe, est la gestion des préemptions (`do`/`until`) et suspensions (`do`/`when`). L'utilisation d'une liste de continuations à exécuter fait que l'on perd complètement la structure du programme. Cela permet d'obtenir une exécution efficace, mais on a besoin d'une autre structure de données pour gérer l'activation des processus, puisque tous les processus ne sont pas forcément actifs à un instant donné. Nous décrirons cette structure que l'on appelle *arbre de contrôle* dans la partie 5.

Combinateurs L'implantation des différentes fonctionnalités du langage peut se faire à partir d'un nombre limité de primitives d'ordonnancement, définies dans un module appelé `Runtime`. La séparation entre la définition des combinateurs et celle du moteur d'exécution permet de clarifier l'implantation en séparant les différents problèmes (*separation of concerns*). Elle autorise aussi le partage de code entre les différentes versions du moteur d'exécution, notamment entre l'implantation séquentielle et l'implantation parallèle que nous décrirons dans la partie 6.

La figure 3 présente un extrait de l'interface du module `Runtime`. On rappelle que le type d'une continuation qui attend une valeur de type `'a` est `'a step = 'a -> unit`. On définit plusieurs combinateurs correspondant aux différentes opérations dont on a besoin :

— `on_current_instant` exécute une continuation à l'instant courant. Cela revient à l'ajouter dans la liste $\mathcal{C}$.

— `on_next_instant` exécute une continuation à l'instant suivant. Cela correspond typiquement à l'opérateur `pause` et revient à ajouter la continuation dans la liste *next*.

— `on_eoi` exécute une continuation à la fin de l'instant. On l'utilise par exemple pour attendre la fin de l'instant pour connaître la valeur d'un signal.

— `on_event` exécute la continuation immédiatement après l'émission du signal, ce qui correspond à la construction `await immediate`.

— `on_event_or_next` correspond à la construction `present`. Il prend donc en entrée deux continuations. Il exécute la première immédiatement si le signal est émis et la seconde à la fin de l'instant dans le cas contraire.

La définition de certains combinateurs devient immédiate une fois que l'on dispose de ces primitives. Par exemple, la construction `await immediate` $e.k$ est traduite par le combinateur `rml_await_immediate_expr`, qui utilise la primitive `Runtime.on_event` (`expr_evt()` déclenche l'évaluation de `expr_evt` qui doit renvoyer un signal). De la même façon, la primitive `on_next_instant` permet d'implémenter simplement l'attente de l'instant suivant avec `pause` :

```
let rml_await_immediate expr_evt k _ = Runtime.on_event (expr_evt()) k ()
let rml_pause k _ = Runtime.on_next_instant k
```

```
module type Runtime = sig
  ...
  (* [on_current_instant f] execute f a l'instant courant *)
  val on_current_instant : unit step -> unit
  (* [on_next_instant f] execute f a l'instant suivant *)
  val on_next_instant : unit step -> unit
  (* [on_eoi f] execute f pendant la fin de l'instant *)
  val on_eoi : unit step -> unit
  (* [on_event ev f] execute f a l'emission de evt *)
  val on_event : ('a, 'b) event  -> unit step ->
  (* [on_event_or_next ev f_ev f_n] execute f_ev si le signal est emis,
     sinon execute f_n a l'instant suivant *)
  val on_event_or_next : ('a, 'b) event -> unit step -> unit step -> unit
  ...
```

FIGURE 3 – Interface du module `Runtime`

Les autres constructions s'obtiennent en composant plusieurs primitives. On peut, par exemple, implémenter un combinateur appelé `on_event_at_eoi` qui exécute une continuation `f` à la fin de l'instant où le signal `evt` est émis :

```
let on_event_at_eoi evt f = Runtime.on_event evt (fun () -> Runtime.on_eoi f)
```

Lorsque le signal `evt` est émis, on ajoute `f` dans la liste des continuations à exécuter à la fin de l'instant. On peut maintenant définir le combinateur `rml_await_all`, qui correspond à la construction `await` $e(x)$ `in` k et qui permet de récupérer la valeur émise sur un signal :

```
let rml_await_all expr_evt k _ =
  let evt = expr_evt () in
  let await_eoi _ =
    let v = Runtime.Event.value evt in
    Runtime.on_next_instant (fun () -> k v)
  in
  on_event_at_eoi evt await_eoi
```

Ce combinateur prend en entrée un signal `expr_evt` et une continuation `k` qui attend la valeur du signal. On utilise le combinateur `on_event_at_eoi`, que l'on vient de définir, pour exécuter la fonction `await_eoi` à la fin de l'instant où le signal `evt` est émis. Cette fonction lit la valeur du signal en utilisant le module `Runtime.Event` qui regroupe les fonctions relatives aux signaux. Plus précisément, on appelle la fonction `value` qui renvoie la valeur du signal à l'instant courant. On demande enfin l'exécution de la continuation `k` à l'instant suivant, en lui donnant comme argument la valeur `v` du signal.

Composition parallèle synchrone Nous allons maintenant décrire l'implantation de la composition parallèle synchrone (`let` $x = e$ `and` $x = e$ `in` e en ReactiveML), qui correspond aux constructions `split`, `join` et `def` en $\mathcal{L}_k$. Le principe est le suivant :

- L'instruction `split` crée un compteur qui est une référence initialisée au nombre de branches. Elle crée aussi une référence par branche pour stocker le résultat de cette branche.
- Les instructions `join` et `def` sont implémentées par le même combinateur. Celui-ci décrémente le compteur, puis stocke le résultat de son argument, qui est la valeur de la branche, dans la référence associée à la branche. Si le compteur atteint zéro, alors toutes les branches ont terminé. Le combinateur appelle alors sa continuation avec un n-uplet formé des valeurs des différentes branches.

Pour obtenir une exécution efficace, le moteur d'exécution définit également un opérateur de composition parallèle n-aire qui reprend les mêmes principes. On peut aller encore plus loin et partager les points de synchronisation de façon dynamique. L'idée est que si la continuation d'une composition parallèle est une instruction `join` d'un autre parallèle, alors on peut utiliser le même compteur pour les deux parallèles. Pour cela, chaque combinateur prend maintenant en argument un point de synchronisation, qui est soit `None`, soit `Some jp` où `jp` est le compteur associé à la composition parallèle dans laquelle est lancé le processus. Lorsque l'on exécute une instruction `split`, on réutilise le compteur donné en argument et on en crée un nouveau si l'argument vaut `None`. Grâce à cette approche, on peut n'utiliser qu'un seul compteur pour synchroniser un nombre dynamique de processus.

Gestion des signaux Pour obtenir une exécution efficace, il est important de ne pas faire d'attente active des signaux, c'est-à-dire que l'attente d'un signal ne doit rien coûter tant que le signal est absent. On associe pour cela deux listes d'attente à chaque signal. Un signal est ainsi un triplet `(n, wa, wp)` où `n` est un enregistrement contenant le statut et les valeurs émises sur le signal, `wa` contient les processus en attente de l'émission du signal et `wp` contient les processus qui testent la présence du signal avec la construction `present`. On réveille les processus dans ces deux listes au moment de l'émission d'un signal. On implémente le combinateur `on_event` par :

```
let on_event (n, wa, wp) f =
  if Event.status n then f () else wa := f :: !wa
```

On teste tout d'abord la présence du signal, c'est-à-dire s'il a déjà été émis au cours de l'instant. Si c'est le cas, alors on exécute la fonction `f`. Sinon, on la stocke dans la liste `wa` des processus en attente de l'émission du signal.

On procède de façon similaire pour la primitive `on_event_or_next`, qui correspond à la construction `present` :

```
let on_event_or_next (n, wa, wp) f_ev f_n =
  let act () = if is_eoi () then next := f_n :: !next else f_ev () in
  if Event.status n then f_ev ()
  else (wp := act :: !wp; weoi := wp :: !weoi)
```

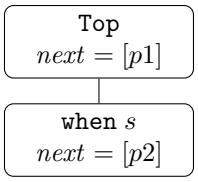
Si le signal est présent, alors on lance la fonction `f_ev` immédiatement. Sinon, on ajoute la fonction `act` dans la liste `wp`. On ajoute ensuite cette liste dans une liste globale appelée `weoi` (qui est donc une liste de références vers des listes). On réveille tous les processus dans ces listes au moment de la fin de l'instant. La fonction `act` teste si l'on est en train d'exécuter la fin de l'instant avec la fonction `is_eoi`. Si ce n'est pas le cas, alors cela signifie que le processus a été réveillé car le signal a été émis. On lance donc immédiatement `f_ev`. Sinon, cela signifie que le signal n'a pas été émis au cours de l'instant et il est donc absent. On ajoute donc `f_n` à la liste *next* des processus à exécuter à l'instant suivant. On peut remarquer qu'il n'est pas nécessaire d'enlever `wp` de la liste `weoi` en cas d'émission car dans ce cas, `wp` contient la liste vide.

5. Suspension et préemption

Nous avons vu dans la partie précédente les bases de l'implantation du moteur d'exécution de ReactiveML. Nous allons maintenant voir comment prendre en compte le reste du langage, et en particulier la suspension et la préemption. Ces deux structures de contrôle différencient ReactiveML des autres bibliothèques de concurrence coopérative mais rendent aussi l'implantation bien plus complexe. Nous nous contentons ici de présenter les grands principes de l'implantation. On trouvera une présentation plus formelle dans la partie 7.3 de [16] qui présente une sémantique de $\mathcal{L}_k$ avec préemption et suspension.

Principe L'utilisation d'une liste $\mathcal{C}$ de continuations à exécuter fait que l'on perd complètement la structure du programme. Cela permet d'obtenir une exécution efficace, mais on a besoin d'une autre structure de données pour gérer l'activation des processus, puisque tous les processus ne sont pas forcément actifs à un instant donné. Cette structure s'appelle *arbre de contrôle*, puisqu'il s'agit d'un arbre n-aire traduisant l'imbrication des préemptions et suspensions dans le programme. Chaque nœud de cet arbre correspond à une construction `do/until` ou `do/when` dans le programme. Considérons par exemple le processus suivant :

```
let process control_tree s p1 p2 =
  emit s; pause; run p1
  ||
  do
    pause; run p2
  when s done
```



Le processus `control_tree` prend en entrée un signal `s` et deux processus `p1` et `p2`. Il lance `p1` et `p2` au second instant, mais `p2` n'est activé qu'aux instants où le signal `s` est présent. La figure sur la droite montre l'arbre de contrôle associé à l'exécution de ce programme à la fin du premier instant de l'exécution de `control_tree`. Sa racine notée `Top` correspond aux processus toujours actifs, alors que le nœud `when` s est associé à la suspension. Chaque nœud de l'arbre de contrôle contient une liste *next* des continuations à exécuter au prochain instant où le corps de la structure de contrôle est actif, c'est-à-dire au prochain instant où le signal `s` est présent dans le cas de la suspension. La liste *next* que l'on a évoquée précédemment est celle qui est associée au nœud `Top` et qui correspond aux processus activés à tous les instants. Ainsi, à la fin du premier instant, la liste *next* de la racine de l'arbre contient le processus `p1`, alors que celle associée à la suspension contient `p2`. On transfère alors les processus depuis la liste *next* de la racine dans la liste $\mathcal{C}$ des processus à exécuter, mais pas ceux du nœud `when` s. On ne le fera qu'à la prochaine émission du signal `s`, lorsqu'on saura que le corps de la suspension est activé.

Dans l'implantation, tous les combinateurs prennent désormais en argument le nœud de contrôle correspondant au contexte dans lequel s'exécute le code. La fonction de traduction $C_k[e]$ (figure 2) est modifiée pour ajouter ces arguments supplémentaires [16]. De même, les primitives `on_next_instant`, `on_event` et `on_event_or_next` de la figure 3 prennent en argument un nœud de l'arbre de contrôle. Par exemple, `on_next_instant ctrl f` exécute `f` à l'instant suivant où le nœud de contrôle `ctrl` est activé, en l'ajoutant dans la liste *next* du nœud `ctrl`.

Implémentation de `do/until` En ReactiveML, la préemption est faible. Cela signifie que l'on ne préempte le corps de la suspension qu'à la fin de l'instant si le signal de préemption est présent. On vérifie donc à la fin de chaque instant si ce signal est présent. Si c'est le cas, alors on désactive le nœud correspondant de l'arbre de contrôle et on ajoute la continuation de la préemption dans la liste *next* du nœud parent. Une alternative est d'utiliser la primitive `on_event` pour attendre l'émission du signal de préemption. On enregistre alors un processus pour désactiver le nœud à la fin de l'instant où le signal est présent.

Implémentation de `do/when` Dans le cas du `do/when`, le nœud de contrôle est activé au moment de l'émission du signal de suspension. On transfère alors les processus depuis sa liste *next* dans la liste $\mathcal{C}$ des processus à exécuter. On active également les nœuds enfants dans l'arbre de contrôle. Pendant la fin de l'instant, on ne doit parcourir un nœud associé à une suspension et ses enfants que si le signal de suspension a été émis pendant l'instant. On désactive alors le nœud pour l'instant suivant et on se remet en attente du signal de suspension.

Attente passive avec preemption et suspension Nous avons vu dans la partie 4 que lorsqu'un processus attend un signal, on le place dans la file d'attente attachée au signal et on le réveille uniquement lorsque le signal est émis. Mais si le processus s'exécute sous une suspension, il ne faut le lancer que si le signal de suspension est présent à cet instant. Dans le cas contraire, tout se passe comme si le processus n'avait pas vu l'émission et il doit rester en attente de la prochaine émission du signal. Illustrons ce problème avec le processus suivant :

```
1 let process await_when =
2   signal act, s in
3   do
4     await immediate s; print_endline "Recu!"
5   when act done
6   || loop pause; emit act; pause end
7   || emit s; pause; emit s
```

On attend l'émission de `s` pour afficher immédiatement le message `"Recu!"`, mais uniquement aux instants où le signal `act` est présent, c'est-à-dire aux instants pairs (voir ligne 6). La première émission de `s`, dans la troisième branche du parallèle, a lieu au cours du premier instant, au cours duquel le corps de la suspension n'est pas actif. On ne doit donc pas réveiller le processus en attente de `s`. On émet ensuite une seconde fois `act` au cours du deuxième instant. Comme le signal de contrôle est cette fois présent, on affiche le message `"Recu!"`.

Une première solution à ce problème, qui est utilisée dans les premières versions de ReactiveML, consiste à utiliser l'attente active lorsque l'on attend un signal sous une suspension ou une préemption active. Une autre approche permet de gérer l'attente passive dans tous les cas. On suit l'algorithme suivant pour lancer une fonction `f` lorsque le signal `evt` est émis et que le nœud de contrôle `ctrl` est actif :

1. Si le signal est présent à l'instant où on démarre l'attente, alors on peut lancer directement `f`. En effet, on sait que le nœud `ctrl` est actif puisque l'on est en train d'exécuter un processus dans ce contexte.

2. Sinon, on met une fonction dans la liste d'attente du signal. Celle-ci effectue les étapes suivantes à l'émission du signal :
   — Si le nœud de contrôle est actif, alors on lance `f`.
   — Sinon, on attend à la fois la fin de l'instant et l'activation du nœud de contrôle `ctrl` qui peut avoir lieu plus tard dans l'instant. En effet, dans le cas d'une expression `do e when s done`, on active le nœud de contrôle correspondant à la suspension lorsque `s` est émis, ce qui peut se passer plus tard dans l'instant. Si le nœud `ctrl` est activé avant la fin de l'instant, alors on lance `f`. Sinon, on sait que le nœud n'est pas actif, ce qui signifie qu'il faut attendre la prochaine émission du signal `evt`. On recommence donc au début de l'étape 2.

Les deux approches ont leurs avantages. Par exemple, l'attente active est plus efficace si le signal `s` est présent à tous les instants mais que `act` est rarement présent. À l'opposé, la version avec attente passive est plus efficace si le signal de suspension `act` est toujours présent et si `s` est rarement émis.

6. Exécution parallèle

Nous cherchons maintenant à répondre à la question de la parallélisation du moteur d'exécution de ReactiveML. La version que nous avons décrite précédemment est une implantation séquentielle de la concurrence avec un ordonnancement coopératif. On souhaite maintenant exécuter un programme ReactiveML sur plusieurs cœurs pour gagner en efficacité, mais sans changer le programme et de façon transparente pour l'utilisateur.

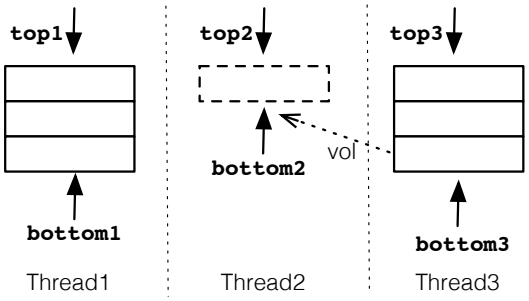


FIGURE 4 – Vol de tâches

Outils et langage ReactiveML est bâti sur un noyau d'OCaml (sans objets, variants polymorphes ni modules). Malheureusement, nous ne pouvons pas utiliser la version standard d'OCaml pour implémenter le moteur d'exécution parallèle. En effet, bien qu'il soit possible de créer des threads en OCaml (avec le module `Thread` de la librairie standard), un verrou global empêche l'exécution de plusieurs threads OCaml en parallèle [8]. Nous avons donc choisi d'utiliser le langage F# [9] pour mener à bien cette expérience. Il est en effet très proche d'OCaml et quasiment compatible au niveau du source avec le sous-ensemble d'OCaml que génère le compilateur ReactiveML.

Implémentation On a vu que le moteur d'exécution de ReactiveML est centré autour d'une liste $\mathcal{C}$ de continuations dans laquelle l'ordonnanceur vient piocher des tâches à exécuter. Le principe de la version parallèle en mémoire partagée est d'avoir plusieurs threads qui viennent piocher des continuations dans la liste $\mathcal{C}$ et les exécutent. On ne parallélise que l'exécution au cours d'un instant. En particulier, la fin de l'instant reste purement séquentielle et tous les threads se synchronisent à la fin de chaque instant.

Pour représenter la liste $\mathcal{C}$, on utilise une structure de données concurrente pour réaliser du *vol de tâches* ou *work stealing* [14]. La figure 4 représente le principe de l'algorithme. Chaque thread dispose de sa propre liste de continuations à exécuter, qui est une file à double entrée ou *deque*, symbolisée par un sommet noté `top` et un pointeur `bottom` indiquant le bas de la file. Lorsque la liste d'un thread est vide, comme dans le cas du second thread de la figure, il va « voler » des tâches à exécuter dans la file d'un autre thread, ici le troisième thread. L'utilisation d'une file à double entrée permet d'éviter dans la majorité des cas les conflits entre le voleur, qui accède au bas `bottom` de la file, et le propriétaire de la file qui accède au sommet `top` de la file. Cet algorithme peut s'implémenter de façon très efficace sans verrous avec des opérations atomiques comme le *compare-and-swap* [10]. Nous utilisons la classe `System.Collections.Concurrent.ConcurrentBag` de la librairie standard .NET qui implémente cette structure de données.

Puisque les threads communiquent par mémoire partagée, on peut réutiliser l'essentiel du code de la version séquentielle du moteur d'exécution. Il faut toutefois gérer les accès concurrents aux différentes structures de données du moteur d'exécution. Le principe est d'associer un verrou à chaque structure partagée. On peut cependant faire mieux :

- On associe à chaque composition parallèle synchrone un compteur du nombre de branches qui ont terminé leur exécution. Comme il s'agit d'entiers, on utilise des opérations atomiques pour décrémenter ces compteurs, de la classe `System.Threading.Interlocked`.
- On peut utiliser des structures de données sans verrous [14] pour les listes partagées. On utilise par exemple une file sans verrous de la classe `ConcurrentStack` pour les listes *next* des nœuds de l'arbre de contrôle.

8. Voir par exemple la partie 19.10.2 *Parallel execution of long-running C code* de `http://caml.inria.fr/pub/docs/manual-ocaml/manual033.html`
9. `http://research.microsoft.com/en-us/projects/fsharp/`

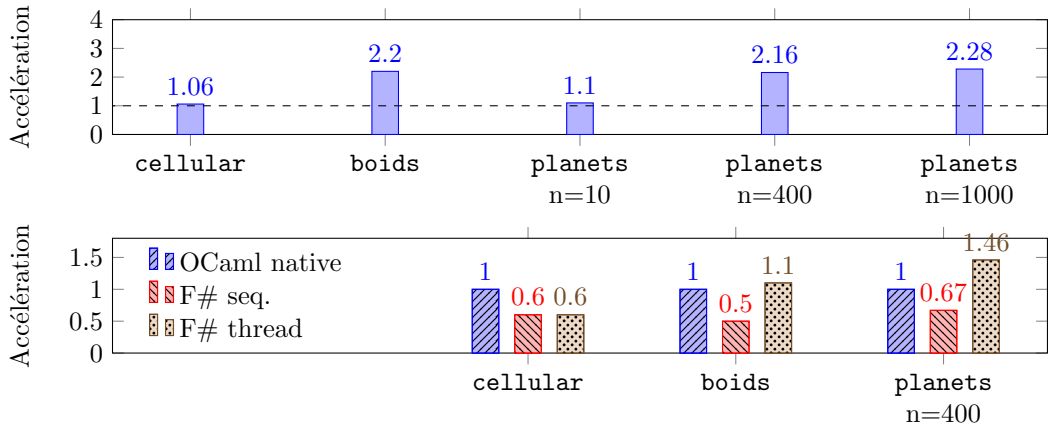


FIGURE 5 – Résultats expérimentaux du moteur d'exécution parallèle en F#
(4 threads, 2 processeurs avec 2 cœurs chacun, F# 3.0)

— On peut se passer de verrous s'il n'y a pas de risques de course critique (*data race*). On parle de course critique si une lecture et une écriture ou deux écritures sur une même location mémoire ne sont pas ordonnées par des verrous. Dans le cas de la dernière valeur d'un signal, elle n'est modifiée que pendant la fin de l'instant, qui est séquentielle, et on ne fait que lire la valeur pendant l'instant. Il n'y a donc pas de risque de course critique.

On utilise des verrous pour les nœuds de l'arbre de contrôle, mais ceci ne pose pas de problème car les modifications de ces nœuds sont rares. L'autre cas où l'on utilise encore un verrou est pour l'implantation des signaux. En effet, au moment de l'émission d'un signal, on doit, de façon atomique, changer le statut du signal et récupérer les processus en attente du signal pour les réveiller. Il faut utiliser un verrou pour garantir qu'aucun autre processus ne puisse voir le signal absent sans être dans la liste des processus en attente.

Résultats expérimentaux La figure 5 montre les performances du moteur d'exécution parallèle écrit en F#. On mesure le temps d'exécution total de plusieurs instants de chaque programme. Le premier graphe montre les résultats pour plusieurs exemples typiques de ReactiveML, tirés de la distribution du compilateur. On mesure ici l'accélération par rapport à la version séquentielle F#, c'est-à-dire le rapport entre la durée du calcul pour la version séquentielle et la durée du calcul dans la version parallèle. Plus la barre est élevée, plus la version parallèle est rapide. Les deux premiers exemples sont des simulations d'automates cellulaires [7] et de *boids* [19], où l'on simule une nuée d'oiseaux en vol. Les trois dernières barres correspondent à une simulation du problème des n-corps dans laquelle on fait varier le nombre de planètes afin de tester plusieurs rapports calcul/synchronisation. On voit que l'on obtient de bons résultats avec une version parallèle deux fois plus rapides environ, sauf dans le cas de la simulation d'automates cellulaires. Cela peut s'expliquer par le fait que ce programme fait très peu de calculs pour chaque processus et aussi par le fait qu'il alloue énormément de mémoire (on alloue une continuation à chaque appel de `pause`), ce qui pose problème puisque le garbage collector de .NET bloque tous les threads.

Le second graphe mesure l'efficacité du moteur d'exécution par rapport à la version OCaml native, qui est cette fois la référence. On peut voir que la version F# séquentielle est toujours plus lente, souvent par un facteur assez grand. Par exemple, elle est deux fois plus lente dans le cas des automates cellulaires ou des boids. L'utilisation du parallélisme permet le plus souvent de combler ce retard, mais n'offre pas des gains importants par rapport à la version OCaml native de référence. Cela relativise donc l'utilité pratique de cette version du moteur d'exécution, en dehors des expérimentations sur la parallélisation du langage.

7. Travaux similaires

Les travaux les plus proches sont ceux autour de Junior [13] et des SugarCubes [9] qui sont des bibliothèques fondées sur le même modèle de concurrence [6]. L'implantation de ReactiveML a d'ailleurs été en particulier inspirée par l'implantation Simple de Junior par Laurent Hazard [12]. Une des différences entre ReactiveML et les autres implantations du modèle réactif est que ReactiveML déstructure le parallélisme et introduit une notion d'arbre de contrôle. Les autres implantations sont des plongements profonds où une représentation explicite de l'arbre de syntaxe abstraite des termes est interprétée. Par ailleurs, la dernière version des SugarCubes [21] supporte l'exécution de programmes OpenCL [10], ce qui n'est pas le cas de ReactiveML.

Par rapport aux Fair Threads [8], qui sont aussi fondés sur le modèle réactif, ReactiveML ne propose pas de support pour la programmation de systèmes globalement asynchrones localement synchrones (GALS), mais dans ce langage, la partie synchrone est moins expressive. En particulier, il n'y a pas de structure hiérarchique dans le parallélisme.

C. Deleuze décrit dans [11] l'implantation d'une bibliothèque de concurrence coopérative reprenant les constructions de ReactiveML. La principale différence avec notre approche concerne la gestion de la préemption et de la suspension. On associe à chaque processus une pile décrivant son contexte d'évaluation, qui correspond aux parents dans l'arbre de contrôle. Cette approche est moins efficace puisqu'il faut reparcourir la pile associée à chaque processus à chaque fin d'instant. Les résultats expérimentaux présentés dans cet article montre que l'ajout de la préemption et la suspension a un impact assez important sur les performances. C'est pourquoi nous n'avons pas cherché ici à comparer l'efficacité de ReactiveML avec celle d'autres bibliothèques de concurrence coopérative en OCaml comme Lwt [22] ou Async, qui ne proposent pas la même expressivité.

Alors que ReactiveML utilise un moteur d'exécution pour obtenir un ordonnancement dynamique, la concurrence d'Esterel est compilée vers du code séquentiel avec ordonnancement statique, après une analyse de causalité préalable pour rejeter les programmes incorrects. Les premières versions du langage compilent les programmes Esterel vers des automates [4], dont la taille est exponentielle en la taille du programme source. La traduction d'Esterel en circuits [1] utilisée dans les versions suivantes permet d'éviter cette explosion combinatoire et de générer aussi bien un programme qu'un circuit à partir du même programme Esterel. D'autres approches [18] se spécialisent dans la génération de code séquentiel et offrent de meilleures performances. Plus récemment, le langage HipHop [5] intègre Esterel dans Hop, qui est un langage *multi-tier* pour la programmation de services Web. Son moteur d'exécution traduit les constructions Esterel en un arbre de syntaxe abstraite (AST) qui est ensuite interprété dynamiquement suivant la sémantique constructive de [3].

8. Conclusion

Nous avons présenté l'implantation du langage ReactiveML. Elle est basée sur l'utilisation de continuations et d'un arbre de contrôle pour gérer l'activation des processus. Son efficacité repose à la fois sur sa transformation CPS partielle qui laisse le code instantané inchangé, le plongement superficiel dans OCaml qui permet un accès en temps constant aux informations du runtime (signaux, points de synchronisation, nœuds de l'arbre de contrôle) et à la déstructuration du parallélisme. Cette approche s'étend à une exécution parallèle basée sur le vol de tâches avec de bonnes performances.

Un axe de recherche pour améliorer les performances du langage est de compiler les parties statiques des programmes en s'inspirant de ce qui est fait en Esterel. Une première approche remplaçant l'ordonnancement dynamique par un ordonnancement statique choisi à la compilation a été expérimentée [15].

10. `http://www.khronos.org/opencl/`

Remerciements Nous souhaitons remercier les rapporteurs des JFLA pour leur relecture attentive, Frédéric Boussinot pour ses conseils et Johannes Kanig avec qui cela a été un plaisir de participer au concours ICFP.

Références

[1] G. Berry. Esterel on hardware, mechanized reasoning and hardware design, 1992.

[2] G. Berry. Preemption in concurrent systems. In *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 72–93, 1993.

[3] G. Berry. The constructive semantics of pure Esterel, 1996.

[4] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[5] G. Berry, C. Nicolas, and M. Serrano. Hiphop: a synchronous reactive extension for Hop. In *Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients*, pages 49–56. ACM, 2011.

[6] F. Boussinot. Reactive C: an extension of C to program reactive systems. *Software: Practice and Experience*, 21(4):401–428, 1991.

[7] F. Boussinot. Reactive Programming of Cellular Automata. RR 5183, INRIA, May 2004. `http://hal.inria.fr/inria-00071405`.

[8] F. Boussinot. Fairthreads: Mixing cooperative and preemptive threads in C. *Concurrency and Computation: Practice and Experience*, 18(5):445–469, April 2006.

[9] F. Boussinot and J.-F. Susini. The SugarCubes tool box: A reactive Java framework. *Software Practice and Experience*, 28(4):1531–1550, 1998.

[10] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28. ACM, 2005.

[11] C. Deleuze. Programmation réactive en OCaml. *Journal Européen des Systèmes Automatisés*, 43(7-8-9):757–771, 2009. 15 pages.

[12] L. Hazard. Simple. An efficient implementation of Junior.

[13] L. Hazard, J.-F. Susini, and F. Boussinot. The Junior reactive kernel. RR 3732, INRIA, 1999. `http://hal.inria.fr/inria-00072933`.

[14] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[15] L. Jachiet. Compilation de ReactiveML. Master's thesis, École normale supérieure, 2013.

[16] L. Mandel. *Conception, Sémantique et Implantation de ReactiveML : un langage à la ML pour la programmation réactive*. PhD thesis, Université Paris 6 - Pierre et Marie Curie, 2006.

[17] L. Mandel and M. Pouzet. ReactiveML, un langage fonctionnel pour la programmation réactive. *Technique et science informatiques*, 27(9-10):1097–1128, 2008.

[18] D. Potop-Butucaru, S.A. Edwards, and G. Berry. *Compiling Esterel*, volume 86. Springer, 2007.

[19] C.W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *ACM SIGGRAPH Computer Graphics*, volume 21, pages 25–34. ACM, 1987.

[20] J.C. Reynolds. The Discoveries of Continuations. *Lisp and Symbolic Computation*, 6(3/4):233–248, 1993.

[21] J.-F. Susini. Les SugarCubes v5. Research report, CNAM, 2013.

[22] J. Vouillon. Lwt: a cooperative thread library. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 3–12. ACM, 2008.