ReactiveML : un langage fonctionnel pour la programmation réactive

Louis Mandel[*] — **Marc Pouzet**[**]

[*] *Institut National de Recherche en Informatique et en Automatique*
Domaine de Voluceau, 78153 Le Chesnay, France
louis.mandel@inria.fr

[**] *Laboratoire de Recherche en Informatique*
Université Paris-Sud, Bât. 490, 91405 Orsay, France
marc.pouzet@lri.fr

RÉSUMÉ. *La programmation de systèmes réactifs tels que les simulateurs de systèmes dynamiques ou les jeux vidéo est une tâche difficile. Les techniques classiques pour programmer ces systèmes sont fondées sur l'utilisation de bibliothèques de threads ou de programmation événementielle. Nous introduisons ici le langage* REACTIVEML *comme une alternative à ces pratiques. Le langage est une extension de OCaml fondée sur le modèle réactif synchrone de Boussinot. Ce modèle reprend des principes du synchrone tels que la composition parallèle déterministe et la communication par diffusion. Il les combine à des mécanismes de création dynamique de processus. Cet article présente le langage, son système de type et sa sémantique.*

ABSTRACT. *The programming of reactive systems such as simulators of dynamic systems or video games is a difficult task. We introduce the* REACTIVEML *language as an alternative to classical techniques such as event loops or thread-based programming.* REACTIVEML *is an extension of OCaml founded on the synchronous reactive model of Boussinot. This model combines the synchronous model which provides parallel composition and instantaneous communications with dynamic creation of processes. This paper presents the language, its type system and its formal semantics.*

MOTS-CLÉS : *programmation réactive synchrone, programmation fonctionnelle, concurrence, typage, sémantique formelle.*

KEYWORDS: *synchronous reactive programming, functional programming, concurrency, typing, formal semantics.*

2[e] soumission à *TSI*, le 20 novembre 2006

1. Introduction

Dans cet article, nous nous intéressons à la programmation de systèmes *réactifs*. À la différence d'un système transformationnel qui lit une entrée, effectue un calcul et produit un résultat, un système réactif se caractérise par son interaction permanente avec un environnement extérieur. Certains de ces systèmes sont dits *temps-réel* lorsque le temps de réaction est imposé par l'environnement extérieur. C'est le cas du système de commande de vol d'un avion : il doit pouvoir répondre en permanence aux sollicitations du pilote et est soumis aux lois de la physique qui, elle, n'attend pas ! Un système de fenêtrage d'ordinateur, au contraire, n'est pas soumis à de telles contraintes. Lorsque l'utilisateur demande l'ouverture d'une fenêtre, le système essaye de répondre le plus vite possible, mais aucun temps de réponse n'est garanti. Ces systèmes interactifs sont plus expressifs que les systèmes temps-réel car ils peuvent évoluer dynamiquement en fonction des sollicitations de l'environnement. De nouveaux processus peuvent être créés ou supprimés en cours d'exécution.

Deux modèles sont principalement utilisés pour la programmation de ces systèmes : les *boucles événementielles* et les *processus légers* (ou *threads*).

Les threads (Kleiman *et al.*, 1996) tel qu'on peut les trouver dans les distributions de JAVA ou OCAML proposent un modèle de concurrence où les processus sont exécutés indépendamment et les communications entre les processus se font par mémoire partagée. Dans ce modèle, l'ordonnancement est préemptif et les synchronisations se font avec des primitives de synchronisation (verrous et sémaphores). L'utilisation de threads est intéressante lorsqu'il y a peu de communications et de synchronisation entre processus. Elle s'avère cependant difficile pour deux raisons principales (Ousterhout, 1996). L'accès à la mémoire partagée doit être protégé afin d'éviter les incohérences de données, introduisant par la même des possibilités d'interblocages. De plus, la non reproductibilité des exécutions rend la mise au point des programmes délicate.

La programmation événementielle (Ferg, 2006) (*event-driven programming*) est une autre approche pour la programmation de systèmes interactifs. Ce modèle est basé sur un ordonnancement coopératif où chaque processus contrôle le moment où il peut être interrompu. Dans ce modèle, des actions sont attachées à des événements. Une boucle événementielle récupère les événements et déclenche successivement les actions qui leur sont associées. Chaque action doit être de courte durée afin de ne pas bloquer la boucle d'événements. C'est un des inconvénients de ce modèle (von Behren *et al.*, 2003).

Pour répondre à ces faiblesses, Frédéric Boussinot introduit dans les années 90 le *modèle réactif synchrone* (Boussinot *et al.*, 1996; Boussinot, 1991). Ce modèle permet de concilier les principes des langages de programmation synchrones conçus à l'origine pour la programmation de systèmes temps-réel avec la possibilité de créer dynamiquement des processus. Le modèle réactif synchrone se base sur le modèle synchrone d'ESTEREL (Berry *et al.*, 1987; Berry, 1998; Benveniste *et al.*, 2003). Dans ce modèle, le temps est défini logiquement comme une succession de réactions à des signaux d'entrée. Au cours d'une réaction le statut d'un signal est présent ou absent (il ne peut pas évoluer). La réaction du système à son environnement est supposée instantanée.

Cette hypothèse de réaction instantanée peut cependant introduire des incohérences temporelles ou *boucles de causalité* comme le montre ce programme ESTEREL :

```
signal s in present s then nothing else emit s end; end
```

Dans ce programme, il n'est pas possible de donner un statut cohérent au signal local `s`. Si l'on suppose que `s` est absent, la branche `else` est exécutée et `s` est émis. `s` devrait donc être présent et absent dans le même instant ! Si l'on fait l'hypothèse que `s` est présent, la branche `then` est exécutée et `s` n'est pas émis. Cette hypothèse conduit donc aussi à une incohérence. Le rôle de l'analyse de causalité d'ESTEREL est donc de rejeter statiquement ce type de programme.

L'idée centrale du modèle *réactif synchrone* vient de l'observation qu'il est possible d'éliminer les problèmes de causalité à condition de réagir avec retard au test d'absence d'un signal. Les programmes sont causaux par construction et il devient possible de concilier les principes du synchrone avec la possibilité de créer dynamiquement des processus. Il en résulte un modèle bien adapté à la programmation de systèmes mêlant un très grand nombre de processus fortement synchronisés (jeux, problèmes de simulation, etc.).

Nous présentons ici le langage REACTIVEML, une extension d'OCAML basée sur le modèle réactif. La première partie présente le langage à partir d'exemples. La seconde partie décrit la sémantique du langage. Elle commence par la définition du noyau du langage (section 3). La section 4 présente les règles de typage. Puis, dans les sections 5 et 6, les sémantiques grands pas et petits pas sont présentées. La preuve d'équivalence entre ces deux sémantiques est donnée section 7. Nous terminons avec la présentation d'autres travaux faits autour de la programmation réactive (section 8) et nous discutons des choix faits lors de la conception du langage (section 9).

2. Une introduction à REACTIVEML

REACTIVEML est une extension de OCAML intégrant une notion de temps logique représenté comme une succession d'instants. Dans le langage, les comportements temporels sont définis par des processus alors que toutes les expressions OCAML sont instantanées.

Comparons la fonction OCAML `fact` avec le processus `pfact` qui exécute un appel récursif par instant de tel sorte que `pfact n` soit exécuté en `n` instants :

```
let rec fact n =
  if n <= 1 then 1
  else
    let v = fact (n-1) in
    n * v
val fact : int -> int
```

```
let rec process pfact n =
  pause;
  if n <= 1 then 1
  else
    let v = run (pfact (n-1)) in
    n * v
val pfact : int -> int process
```

Le mot-clé `process` introduit le processus `pfact`, indiquant que son exécution peut

s'effectuer sur plusieurs instants logiques. Ce processus s'instancie en écrivant simplement `run (pfact (n-1))`.

Le temps est introduit dans ce processus par l'instruction `pause`. Cette instruction retarde d'un instant l'exécution de sa continuation. Ainsi, à chaque appel à `pfact`, un instant passe.

2.1. *Les communications*

Les processus communiquent entre eux par diffusion instantanée de signaux. À chaque instant, un signal est soit présent, soit absent et tous les processus observent le même statut. Ainsi, dans l'exemple suivant, lorsque `s` est émis, toutes les expressions parallèles le voient présent. Les signaux `s1` et `s2` sont tous les deux émis.

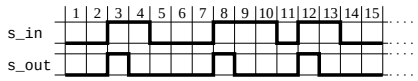
```
    await immediate s; emit s1
 || pause; emit s
 || await immediate s; emit s2
```

Une caractéristique importante du modèle réactif de Boussinot est d'introduire un retard lors du test d'absence d'un signal. Dans le programme suivant, le message `Present` est affiché à l'instant où `s` est présent alors que `Previously absent` n'est affiché qu'à l'instant suivant.

```
let process present_absent s =
  loop
    present s then (print_string "Present"; pause)
    else print_string "Previously␣absent"
  end
val present_absent : ('a, 'b) event -> unit process
```

Voyons enfin l'exemple du détecteur de front haut. Le comportement du processus `edge` est d'émettre le signal `s_out` quand `s_in` est présent et qu'il était absent à l'instant précédent :

```
let process edge s_in s_out =
  loop
    present s_in then pause
    else (await immediate s_in;
          emit s_out)
  end
val edge : ('a, 'b) event -> (unit, 'c) event -> unit process
```



Tant que `s_in` est présent, `s_out` n'est pas émis. Quand `s_in` devient absent, `s_out` n'est toujours pas émis, mais le contrôle passe par la branche `else`. À l'instant suivant, le processus se met en attente de l'émission de `s_in`. Maintenant, quand `s_in` est présent, `s_out` est émis (`s_in` était nécessairement absent à l'instant précédent).

2.2. *Les structures de contrôle*

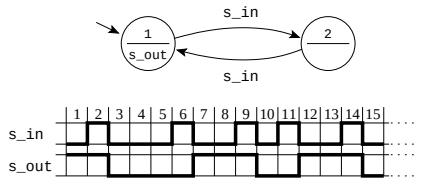
On introduit maintenant les deux principales structures de contrôle du langage : la construction `do` e `when` s permet de suspendre l'exécution de e lorsque s est absent et `do` e `until` s `done` interrompt définitivement l'exécution de e lorsque s est présent. On illustre ces deux constructions avec un processus `suspend_resume` qui contrôle les instants où un processus est exécuté. Pour reprendre l'analogie donnée dans (Berry, 1993), la construction `do/until` correspond à la commande UNIX `Ctrl-c` alors que le processus `suspend_resume` a le comportement du couple de commande `Ctrl-z` et `fg`.

Considérons d'abord le processus `sustain` qui maintient l'émission d'un signal `s` à chaque instant :

```
let process sustain s = loop emit s; pause end
val sustain : (unit, 'a) event -> unit process
```

Le processus `switch` est un autre opérateur typique. Il est paramétré par deux signaux, `s_in` et `s_out`. Son comportement consiste à maintenir l'émission de `s_out` tant que `s_in` est absent. Lorsque la présence de `s_in` est détectée, le processus suspend l'émission de `s_out` jusqu'à ce que `s_in` soit émis à nouveau. [1] Le processus retourne alors dans son état initial.

```
let process switch s_in s_out =
  loop
    do
      run (sustain s_out)
    until s_in done;
    await s_in
  end
val switch : ('a, 'b) event -> (unit, 'c) event -> unit process
```



On définit maintenant le processus `suspend_resume`. Il est paramétré par un signal `s` et un processus `p`. Ce processus commence l'exécution de `p`. A chaque émission de `s` il suspend puis reprend l'exécution de `p` alternativement. Ce processus s'implante en composant en parallèle (1) un `do/when` qui exécute le processus `p` seulement lorsque le signal `active` est présent et (2) un processus `switch` qui contrôle l'émission de `active` avec le signal `s`.

```
let process suspend_resume s p =
  signal active in
  do run p when active
  ||
  run (switch s active)
val suspend_resume : ('a, 'b) event -> 'c process -> unit process
```

1. `await s` $\stackrel{def}{=}$ `await immediate s ; pause`

2.3. *Signaux valués et multi-émission*

Les signaux peuvent transporter des valeurs. L'attente d'un signal valué est notée : `await` *<signal>* `(`*<pattern>*`) in` *<expression>*. Cette construction `await/in` lie la variable à la valeur du signal. Le corps de cette construction (*<expression>*) est toujours exécuté à l'instant suivant l'émission.

La valeur précédente (respectivement le statut) d'un signal `s` peut être accédée en écrivant `pre ?s` (respectivement `pre s`).

Il est possible d'émettre plusieurs valeurs au cours d'un instant : c'est la multi-émission. Pour cela, il est nécessaire de définir la manière dont seront combinées les valeurs émises durant cet instant. Pour cela, on écrira :

`signal` *<name>* `default` *<value>* `gather` *<function>* `in` *<expression>*

Le programme suivant permet de définir un signal `sum` contenant la somme de toutes les valeurs émises au cours d'un instant. Les valeurs émises sont combinées avec la fonction + et la valeur par défaut `0` :

```
signal sum default 0 gather (+) in ...
sum : (int,int) event
```

Dans ce cas, le programme `await sum(x) in print_int x` attend le premier instant où `x` est présent. Puis, à l'instant suivant, il affiche la somme de toutes les valeurs émises sur `x`.

Le type des valeurs émises sur un signal et le type de la valeur combinée peuvent être différents. Cette information est contenue dans le type inféré pour ce signal. Si τ_1 est le type des valeurs émises sur le signal `s` et τ_2 celui de la combinaison, alors `s` a le type (τ_1, τ_2) `event`. Dans ce cas, la valeur par défaut doit avoir le type τ_2 et la fonction de combinaison $\tau_1 \rightarrow \tau_2 \rightarrow \tau_2$.

Dans l'exemple suivant, le signal `s` collecte toutes les valeurs émises pendant un instant :

```
signal s default [] gather fun x y -> x :: y in ...
s : ('a, 'a list) event
```

Ici, la valeur par défaut est la liste vide et la fonction de combinaison ajoute chaque valeur émise à la liste des valeurs déjà émises. La notation `signal s in ...` est un raccourci pour cette fonction de combinaison.

2.4. *Aspects dynamiques, ordre supérieur et échappement de portée*

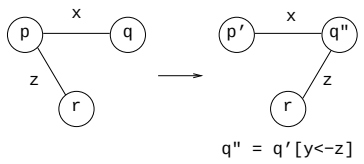
La possibilité d'écrire des processus récursifs ou d'ordre supérieur permet de décrire des systèmes reconfigurables dynamiquement. Considérons le processus `replace` paramétré par un signal `s` et un processus `p`. Ce processus permet de remplacer un processus `p` en cours d'exécution par un processus `p'` à chaque fois que le

signal s est émis. Les processus étant des valeurs de première classe, la définition du processus p' qui doit remplacer p est envoyé sur le signal s.

```
let rec process replace s p =
  do
    run p
  until s(p') -> run (replace s p') done
val replace : ('a, 'b process) event -> 'b process -> 'b process
```

En REACTIVEML, les signaux sont également des valeurs de première classe et peuvent être émis sur un signal, permettant par la même de retrouver des traits propres au π-calcul. Nous l'illustrons sur l'exemple du chapitre 9.3 de (Milner, 1999). Trois processus p, q et r sont exécutés en parallèle. D'une part p et q peuvent communiquer en utilisant un signal x et d'autre part p et r communiquent par le signal z. Les processus p et q peuvent être définis de telle sorte que q et r puissent communiquer en utilisant z.

```
let process p x z =
  emit x z; run (p' x)
val p :
  ('a, 'b) event -> 'a -> unit process
```




```
let process q x = await x(y) in run (q' y)
val q : ('a, ('b, 'c) event) event -> unit process

let process r z = ...
val r : ('a, 'b) event -> unit process

let process mobility x =
  run (q x) || signal z in (run (p x z) || run (r z))
val mobility :
  (('a, 'a list) event, ('b, 'c) event) event -> unit process
```

2.5. *Le crible d'Ératosthène*

Terminons cette présentation du langage avec l'exemple du crible d'Ératosthène tel qu'on peut le trouver dans (Kahn, 1974). C'est un exemple classique de l'approche réactive (cf. (Boussinot, 2003) par exemple). Il reprend l'utilisation des signaux, de la composition parallèle et de la création dynamique. Le programme est défini figure 1.

Le processus `integers` émet sur `s_out` la suite des entiers naturels à partir de la valeur de `n` et le processus `filter` supprime les multiples d'un nombre premier.

Dans le processus `filter`, l'expression `await s_in([n]) in ...` attend qu'exactement une valeur soit émise sur le signal `s_in`. Cette valeur est alors nommée `n`.

```
let rec process integers n s_out =
  emit s_out n; pause; run (integers (n+1) s_out)
val integers : int -> (int, 'a) event -> 'b process

let process filter prime s_in s_out =
  loop
    await s_in([n]) in if n mod prime <> 0 then emit s_out n
  end
val filter :
  int -> ('a, int list) event -> (int, 'b) event -> unit process

let rec process shift s_in s_out =
  await s_in([prime]) in emit s_out prime;
  signal s in run (filter prime s_in s) || run (shift s s_out)
val shift : (int, int list) event -> (int, 'a) event -> unit process

let process output s_in =
  loop await s_in ([prime]) in print_int prime end
val output : ('a, int list) event -> unit process

let process sieve =
  signal nat, prime in
  run (integers 2 nat) || run (shift nat prime) || run (output prime)
val sieve : unit process
```

Figure 1. *Crible d'Ératosthène*

Le processus `shift` crée dynamiquement un nouveau processus `filter` à chaque fois qu'un nouveau nombre premier est découvert. La création dynamique se fait par combinaison de la récursion et de la composition parallèle.

Enfin, le processus `output` affiche les nombres premiers et le processus `sieve` est le processus principal.

Nous arrêtons ici cette présentation. Des exemples complets sont disponibles à l'adresse `http://moscova.inria.fr/~mandel/rml`.

3. Le noyau du langage

La sémantique du langage est définie sur un noyau fonctionnel étendu avec des constructions réactives. La syntaxe des expressions (e) est définie dans la figure 2. Les constantes (c) sont soit des valeurs de base comme des entiers, des booléens ou des flottants, soit des opérateurs :

$$c \quad ::= \quad \texttt{true} \mid \texttt{false} \mid \texttt{()} \mid \texttt{0} \mid \ldots \mid \texttt{+} \mid \texttt{-} \mid \ldots$$

Les motifs (p) utilisés dans la construction `do/until` sont des variables, ou des multi-ensembles à un élément :

$$p \quad ::= \quad x \mid \{x\}$$

e	::=	$x \mid c \mid (e,e) \mid \lambda x.e \mid e\ e \mid \texttt{rec}\ x = e$	expressions ML
		$\mid \texttt{process}\ e$	définition de processus
		$\mid \texttt{let}\ x = e\ \texttt{and}\ x = e\ \texttt{in}\ e$	composition parallèle
		$\mid \texttt{signal}\ x\ \texttt{default}\ e\ \texttt{gather}\ e\ \texttt{in}\ e$	déclaration de signal
		$\mid \texttt{present}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	test de présence
		$\mid \texttt{emit}\ e\ e$	émission
		$\mid \texttt{run}\ e$	exécution d'une définition de processus
		$\mid \texttt{pre}\ e \mid \texttt{pre}\ ?e$	statut et valeur précédents d'un signal
		$\mid \texttt{do}\ e\ \texttt{until}\ e(p)\ \texttt{->}\ e$	préemption
		$\mid \texttt{do}\ e\ \texttt{when}\ e$	suspension

Figure 2. *Syntaxe abstraite du noyau de* REACTIVEML

$\texttt{let process}\ f\ x = e_1\ \texttt{in}\ e_2$	$\stackrel{def}{=}$	$\texttt{let}\ f = \lambda x.\texttt{process}\ e_1\ \texttt{in}\ e_2$
$\texttt{emit}\ e$	$\stackrel{def}{=}$	$\texttt{emit}\ e\ ()$
$\texttt{let}\ x_1 = e_1\ \texttt{in}\ e$	$\stackrel{def}{=}$	$\texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = ()\ \texttt{in}\ e \quad x_2 \notin fv(e)$
$e_1 \texttt{||} e_2$	$\stackrel{def}{=}$	$\texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ ()$
$e_1 \texttt{;} e_2$	$\stackrel{def}{=}$	$\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \quad x \notin fv(e)$
$\texttt{pause}$	$\stackrel{def}{=}$	$\texttt{signal}\ x\ \texttt{in present}\ x\ \texttt{then}\ ()\ \texttt{else}\ ()$
$\texttt{await immediate}\ s$	$\stackrel{def}{=}$	$\texttt{do}\ ()\ \texttt{when}\ s$
$\texttt{await}\ s(p)\ \texttt{in}\ e$	$\stackrel{def}{=}$	$\texttt{do loop pause end until}\ s(p)\ \texttt{->}\ e$
$\texttt{loop}\ e\ \texttt{end}$	$\stackrel{def}{=}$	$\texttt{run}\ ((\texttt{rec}\ loop = \lambda x.\texttt{process}\ (\texttt{run}\ x\texttt{;run}\ (loop\ x)))\ e)$
$\texttt{signal}\ s\ \texttt{in}\ e$	$\stackrel{def}{=}$	$\texttt{signal}\ s\ \texttt{default}\ \emptyset\ \texttt{gather}\ \lambda x.\lambda y.\{x\} \uplus y\ \texttt{in}\ e$

Figure 3. *Définitions de constructions* REACTIVEML *dans le noyau du langage*

Les valeurs (v) sont des constantes (c), des noms de signaux (n), des paires de valeurs (v, v), des abstractions ($\lambda x.e$) ou des définitions de processus ($\texttt{process}\ e$) :

$$v \quad ::= \quad c \mid n \mid (v,v) \mid \lambda x.e \mid \texttt{process}\ e$$

Notons que ce noyau permet de traduire facilement les constructions vues dans l'introduction (figure 3).

Dans la définition de `signal/in`, $\emptyset$ désigne le multi-ensemble vide et $\uplus$ est l'union de multi-ensembles (si $m_1 = \{v_1, ..., v_n\}$ et $m_2 = \{v'_1, ..., v'_k\}$ alors $m_1 \uplus m_2 = \{v_1, ..., v_n, v'_1, ..., v'_k\}$). [2]

Le codage de l'instruction `pause` utilise le retard introduit par la réaction à l'absence d'un signal. Comme x est absent, l'exécution de la branche `else` de l'instruction `present` a lieu à l'instant suivant.

4. Sémantiques statiques

4.1. *Identification des expressions instantanées*

Nous distinguons les expressions *instantanées* — ici les expressions OCAML — des expressions *réactives* qui s'exécutent sur plusieurs instants. Les expressions réactives nécessiteront ensuite une compilation particulières alors que les expressions instantanées resteront inchangées.

Nous présentons ici les règles de bonne formation permettant d'effectuer cette séparation. Une expression e est *bien formée* lorsqu'elle vérifie le prédicat $k \vdash e$ défini figure 4 pour $k \in \{0, 1\}$. 0 est le contexte des expressions instantanées alors que 1 est celui des expressions réactives.

Nous notons simplement $k \vdash e$ lorsque $0 \vdash e$ et $1 \vdash e$ sont vérifiés. Cela signifie que les variables et les constantes peuvent être utilisées dans tous les contextes. Une fonction $(\lambda x.e)$ peut également être utilisée dans tous les contextes alors que son corps doit nécessairement être instantané. Pour la définition d'un processus (`process` e), le corps peut être réactif. Toutes les expressions ML sont bien formées dans tous les contextes, mais les expressions comme `run` ou `present` dont l'exécution peut se faire sur plusieurs instants ne peuvent être utilisées que dans des processus. Nous pouvons remarquer qu'aucune règle ne permet de conclure qu'une expression est typée uniquement dans un contexte 0 et pas dans un contexte 1. Par conséquent, toutes les expressions instantanées peuvent être utilisées dans des processus.

4.2. *Typage*

Le système de type de REACTIVEML est une extension conservative du système de type de ML de Milner (Milner, 1978) garantissant que tous les programmes ML bien typés restent bien typés et gardent le même type. Le langage de type est :

$$\begin{array}{lcl} \sigma & ::= & \forall \alpha_1, \ldots, \alpha_n.\tau \\ \tau & ::= & T \mid \alpha \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \tau\,\texttt{process} \mid (\tau, \tau)\,\texttt{event} \\ T & ::= & \texttt{int} \mid \texttt{bool} \mid \ldots \\ H & ::= & [x_1 : \sigma_1; \ldots; x_k : \sigma_k] \end{array}$$

2. Dans l'implantation, les valeurs sont collectées dans une liste et non dans un multi-ensemble. Cela permet d'avoir une syntaxe plus légère pour le filtrage et d'utiliser les fonctions de la bibliothèque standard.

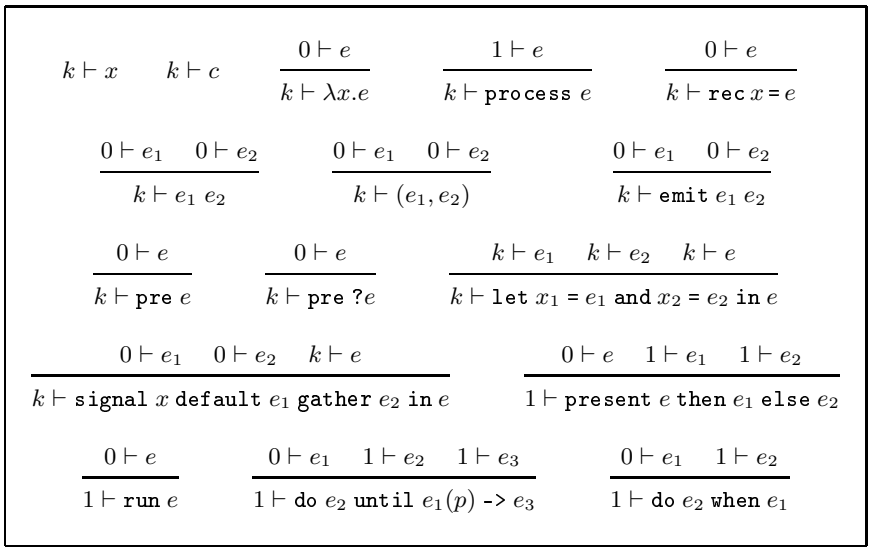


Figure 4. *Séparation des expressions instantanées*

où T est un type de base, α une variable de type, τ un type, σ un schéma de type et H est un environnement de typage qui associe un schéma de type aux variables.

Le prédicat de typage $H \vdash e : \tau$ qui se lit «l'expression e a le type τ dans l'environnement de typage H» est défini figure 5. TC est l'environnement qui définit le type des constantes :

$$TC = [\texttt{true} : \texttt{bool}; \texttt{fst} : \forall \alpha, \beta.\ \alpha \times \beta \rightarrow \alpha; ...]$$

Comme pour le typage des références en ML, il faut veiller à ne pas généraliser le type des expressions qui créent des signaux. Par exemple, le type de `x` ne doit pas être généralisé dans l'expression suivante :

```
let x = signal s in s in emit x 1; emit x true
```

Donner le type $\forall \alpha.(\alpha, \alpha\ \texttt{multiset})\ \texttt{event}$ à `x` conduirait en effet à accepter ce programme qui est pourtant incorrect.

Ce problème est résolu en utilisant la technique de Wright (Wright, 1995) utilisées pour typer les programmes avec références. Les expressions sont distinguées sur des critères syntaxiques. Ainsi, les expressions non-expansives e_{ne} (dont les déclarations de signaux et les applications ne font pas partie) sont définies par :

$$\begin{aligned} e_{ne} ::= {} & x \mid c \mid (e_{ne}, e_{ne}) \mid \lambda x.e \mid \texttt{process}\ e \mid \texttt{emit}\ e_{ne}\ e_{ne} \mid \texttt{pre}\ e_{ne} \mid \texttt{pre}\ ?e_{ne} \\ & \mid \texttt{let}\ x = e_{ne}\ \texttt{and}\ x = e_{ne}\ \texttt{in}\ e_{ne} \mid \texttt{present}\ e_{ne}\ \texttt{then}\ e_{ne}\ \texttt{else}\ e_{ne} \\ & \mid \texttt{do}\ e_{ne}\ \texttt{until}\ e_{ne}(p)\ \texttt{->}\ e_{ne} \mid \texttt{do}\ e_{ne}\ \texttt{when}\ e_{ne} \end{aligned}$$

$$\frac{\tau \leq H(x)}{H \vdash x : \tau} \qquad \frac{\tau \leq TC(c)}{H \vdash c : \tau} \qquad \frac{H \vdash e_1 : \tau_1 \quad H \vdash e_2 : \tau_2}{H \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{H[x : \tau_1] \vdash e : \tau_2}{H \vdash \lambda x.e : \tau_1 \rightarrow \tau_2}$$

$$\frac{H[x : \tau] \vdash e : \tau}{H \vdash \texttt{rec}\, x = e : \tau} \qquad \frac{H \vdash e : \tau}{H \vdash \texttt{process}\, e : \tau\, \texttt{process}}$$

$$\frac{H \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad H \vdash e_2 : \tau_1}{H \vdash e_1\, e_2 : \tau_2} \qquad \frac{H \vdash e_1 : (\tau_1, \tau_2)\, \texttt{event} \quad H \vdash e_2 : \tau_1}{H \vdash \texttt{emit}\, e_1\, e_2 : \texttt{unit}}$$

$$\frac{H \vdash e_1 : \tau_1 \quad H \vdash e_2 : \tau_2 \quad H[x_1 : Gen(H, (e_1, e_2), \tau_1); x_2 : Gen(H, (e_1, e_2), \tau_2)] \vdash e : \tau}{H \vdash \texttt{let}\, x_1 = e_1\, \texttt{and}\, x_2 = e_2\, \texttt{in}\, e : \tau}$$

$$\frac{H \vdash e_1 : \tau_2 \quad H \vdash e_2 : \tau_1 \rightarrow \tau_2 \rightarrow \tau_2 \quad H[s : (\tau_1, \tau_2)\, \texttt{event}] \vdash e : \tau}{H \vdash \texttt{signal}\, s\, \texttt{default}\, e_1\, \texttt{gather}\, e_2\, \texttt{in}\, e : \tau}$$

$$\frac{H \vdash e : (\tau_1, \tau_2)\, \texttt{event} \quad H \vdash e_1 : \tau \quad H \vdash e_2 : \tau}{H \vdash \texttt{present}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : \tau} \qquad \frac{H \vdash e : \tau\, \texttt{process}}{H \vdash \texttt{run}\, e : \tau}$$

$$\frac{H \vdash e : (\tau_1, \tau_2)\, \texttt{event} \quad H \vdash e_1 : \tau \quad H[x : \tau_2] \vdash e_2 : \tau}{H \vdash \texttt{do}\, e_1\, \texttt{until}\, e(x)\, \texttt{->}\, e_2 : \tau}$$

$$\frac{H \vdash e : (\tau_1, \tau_2\, \texttt{multiset})\, \texttt{event} \quad H \vdash e_1 : \tau \quad H[x : \tau_2] \vdash e_2 : \tau}{H \vdash \texttt{do}\, e_1\, \texttt{until}\, e(\{x\})\, \texttt{->}\, e_2 : \tau}$$

$$\frac{H \vdash e_1 : (\tau_1, \tau_2)\, \texttt{event} \quad H \vdash e : \tau}{H \vdash \texttt{do}\, e\, \texttt{when}\, e_1 : \tau} \qquad \frac{H \vdash e : (\tau_1, \tau_2)\, \texttt{event}}{H \vdash \texttt{pre}\, e : \texttt{bool}} \qquad \frac{H \vdash e : (\tau_1, \tau_2)\, \texttt{event}}{H \vdash \texttt{pre}\, \texttt{?}e : \tau_2}$$

Figure 5. *Le système de type de* REACTIVEML

et l'instanciation et la généralisation sont définies comme suit :

$$\tau'[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n] \leq \forall \alpha_1, \ldots, \alpha_k.\tau$$

$$Gen(H, e, \tau) = \begin{cases} \forall \alpha_1, ..., \alpha_n.\tau \text{ où } \{\alpha_1, ..., \alpha_k\} = fv(\tau) \backslash fv(H) \text{ si } e \text{ non-expansive} \\ \tau \text{ sinon} \end{cases}$$

5. Sémantique comportementale

La sémantique comportementale voit l'exécution d'un programme comme une suite de réactions à des événements d'entrée. Elle s'inspire de la *sémantique logique comportementale* d'ESTEREL (Berry, 1998).

La nature «grand pas» de cette sémantique permet de définir la réaction d'un instant sans prendre en compte l'ordonnancement fin à l'intérieur de l'instant. Ainsi, c'est un bon formalisme pour prouver le déterminisme et l'unicité de la réaction.

La sémantique intègre la description du comportement des parties réactives et instantanées. Ceci permet d'obtenir une formalisation complète du langage en rendant explicite les interactions entre les deux mondes à travers un formalisme commun. Ce n'était pas le cas des sémantiques précédentes du modèle réactif (Boussinot *et al.*, 1996; Hazard *et al.*, 1999), ni de la sémantique comportementale d'ESTEREL (Berry, 1998).

5.1. *Définition de la sémantique*

La réaction d'un programme se définit par rapport à un ensemble de signaux. Dans cette section, ces ensembles et les opérations permettant de les manipuler sont définis formellement.

Les noms de signaux sont notés n et appartiennent à un ensemble dénombrable $\mathcal{N}$. Si $N_1 \subseteq \mathcal{N}$ et $N_2 \subseteq \mathcal{N}$, nous notons $N_1 \cdot N_2$ l'union de ces deux ensembles qui est définie uniquement si $N_1 \cap N_2 = \emptyset$.

Un *environnement de signaux* S est une fonction :

$$S \quad ::= \quad [(d_1, g_1, p_1, m_1)/n_1, ..., (d_k, g_k, p_k, m_k)/n_k]$$

qui, à un nom de signal n_i, associe un quadruplet (d_i, g_i, p_i, m_i) où d_i est la valeur par défaut de n_i, g_i est une fonction de combinaison, p_i est une paire représentant le statut du signal à l'instant précédent et sa dernière valeur et m_i le multi-ensemble des valeurs émises pendant la réaction.

Si le signal n_i est de type $(\tau_1, \tau_2)\,\texttt{event}$ alors les valeurs associées au signal ont les types suivants :

$$d_i : \tau_2 \qquad p_i : \texttt{bool} \times \tau_2 \qquad g_i : \tau_1 \rightarrow \tau_2 \rightarrow \tau_2 \qquad m_i : \tau_2\,\texttt{multiset}$$

on note $S^d(n_i) = d_i$, $S^g(n_i) = g_i$, $S^p(n_i) = p_i$ et $S^m(n_i) = m_i$. On définit la valeur associée à un signal par $S^v(n_i) = \mathit{fold}\ g_i\ m_i\ d_i$ avec *fold* tel que :

$$\begin{array}{lcl} \mathit{fold}\ f\ (\{v_1\} \uplus m)\ v_2 & = & \mathit{fold}\ f\ m\ (f\ v_1\ v_2) \\ \mathit{fold}\ f\ \emptyset\ v & = & v \end{array}$$

On utilise la notation $n \in S$ lorsque n est présent dans S ($S^m(n) \neq \emptyset$) et $n \notin S$ lorsqu'il est absent ($S^m(n) = \emptyset$).

Un *événement* E est une fonction des noms de signaux dans les multi-ensembles de valeurs :

$$E \quad ::= \quad [m_1/n_1, ..., m_k/n_k]$$

Les événements servent à représenter des ensembles de valeurs qui sont émises sur des signaux. S^m est l'événement associé à l'environnement S.

$$E = E_1 \sqcup E_2 \text{ ssi } \forall n \in Dom(E_1) \cup Dom(E_2) : \; E(n) = E_1(n) \uplus E_2(n)$$

$$E = E_1 \sqcap E_2 \text{ ssi } \forall n \in Dom(E_1) \cup Dom(E_2) : \; E(n) = E_1(n) \uplus E_2(n)$$

$$E_1 \sqsubseteq E_2 \text{ ssi } \forall n \in Dom(E_1) : E_1(n) \Subset E_2(n)$$

$$S_1 \sqsubseteq S_2 \text{ ssi } S_1^m \sqsubseteq S_2^m$$

$$(S + [v/n])(n') = \begin{cases} S(n') & \text{si } n' \neq n \\ (S^d(n), S^g(n), S^p(n), S^m(n) \uplus \{v\}) & \text{si } n' = n \end{cases}$$

Figure 6. *Opérations sur les événements et les environnements de signaux*

Les opérations sur les environnements de signaux et les événements sont définies figure 6 où $\uplus$, $\uplus$ et $\Subset$ sont les opérations sur les multi-ensembles. L'opération $S+[v/n]$ représente l'ajout de la valeur v au multi-ensemble associé à n dans S.

Nous pouvons maintenant définir la réaction en un instant d'une expression e en une expression e' par une relation de transition de la forme :

$$N \vdash e \xrightarrow[S]{E,\, b} e'$$

où N est l'ensemble des noms de signaux créés par la réaction. S est l'environnement de signaux dans lequel e doit réagir ; il contient les signaux d'entrée, de sortie et les signaux locaux. L'événement E représente les signaux émis pendant la réaction. b est le statut de terminaison, c'est une valeur booléenne indiquant si l'expression e' doit être activée à l'instant suivant ou si elle a terminé sa réaction.

L'exécution d'un programme est une succession potentiellement infinie de réactions. L'exécution est terminée lorsque que le statut b est vrai. À chaque instant, un programme lit des entrées (I_i) et émet des sorties (O_i). L'exécution d'un instant est définie par le plus petit environnement de signaux S_i (pour l'ordre $\sqsubseteq$) tel que :

$$N_i \vdash e_i \xrightarrow[S_i]{E_i,\, b} e'_i$$

où

(1)	$(I_i \sqcup E_i) \sqsubseteq S_i^m$	(4)	$S_i^d \subseteq S_{i+1}^d$ et $S_i^g \subseteq S_{i+1}^g$
(2)	$O_i = next(S_i)$	(5)	$O_i \subseteq S_{i+1}^p$
(3)	$\forall n \in N_{i+1}.\, n \notin Dom(S_i)$		

(1) L'environnement S_i^m doit contenir les signaux d'entrée et ceux émis pendant la réaction, cela garantit la propriété de diffusion instantanée des événements. (2) La sortie O_i associe à chaque signal son statut (présent/absent) et une valeur qui est la

combinaison des valeurs émises. Donc la fonction $next$ qui calcule la sortie est définie par :

$$\forall n \in Dom(S).\ next(S)(n) = \begin{cases} (\textit{false}, v) & \text{si } n \notin S \text{ et } S^p(n) = (b, v) \\ (\textit{true}, v) & \text{si } n \in S \text{ et } S^v(n) = v \end{cases}$$

La sortie associée à un signal n est donc *false* et la valeur précédente de n, s'il est absent. Sinon, c'est *true* et la combinaison des valeurs émises pendant l'instant.

(3) La condition $\forall n \in N_{i+1}.n \notin Dom(S_i)$ garantit que les noms introduits lors de la réaction sont des noms frais. (4) Les conditions $S_i^d \subseteq S_{i+1}^d$ et $S_i^g \subseteq S_{i+1}^g$ indiquent que les valeurs par défaut et les fonctions de combinaison des signaux sont gardées d'un instant à l'autre. (5) Enfin, $O_i \subseteq S_{i+1}^p$ est la transmission des valeurs des signaux pour le calcul du `pre`.

Les contraintes $S_i^d \subseteq S_{i+1}^d$ et $S_i^g \subseteq S_{i+1}^g$ ont pour conséquence de garder dans l'environnement tous les signaux créés par la réaction du programme. Mais on peut remarquer que les signaux de S_i qui ne sont pas des variables libres dans e_i' peuvent être supprimés de l'environnement de signaux S_{i+1}. Dans l'implantation, cette opération est réalisée automatiquement par le glaneur de cellules (GC).

La relation de transition pour les expressions instantanées (celles pour lesquelles $0 \vdash e$) est définie figure 7. Ces expressions étant instantanées, le statut de terminaison b est toujours vrai.

Détaillons la règle de la paire. Pour évaluer (e_1, e_2), on évalue les deux branches dans le même instant. Ces deux branches réagissent dans le même environnement de signaux S pour avoir une vision globale et cohérente des signaux présents pendant un instant. E_1 et E_2 sont les signaux émis par la réaction de e_1 et e_2, donc la réaction de (e_1, e_2) émet l'union de E_1 et E_2. Enfin, les signaux créés par cette réaction sont ceux créés par la réaction de e_1 et e_2. Les noms de ces signaux sont pris respectivement dans N_1 et N_2.

Nous définissons les règles de sémantique du noyau réactif dans la figure 8

– Le comportement du `let/and/in` consiste à exécuter e_1 et e_2 en parallèle. Quand ces expressions sont réduites en des valeurs v_1 et v_2, alors x_1 et x_2 sont substitués respectivement par v_1 et v_2 dans e.

– `signal` x `default` e_1 `gather` e_2 `in` e déclare un nouveau signal x. La valeur par défaut (e_1) et la fonction de combinaison (e_2) sont évaluées au moment de la déclaration. Le nom x est substitué par un nom frais n dans e. Dans l'environnement des signaux, le `pre` du signal est initialisé avec le statut absent et avec la valeur par défaut. Pour le multi-ensemble m, les valeurs émises pendant l'instant doivent être *devinées*. Cela veut dire que la dérivation complète du programme doit vérifier que m contient bien les valeurs émises pendant l'instant.

– Dans le test de présence d'un signal, si le signal est présent, la branche `then` est exécutée instantanément. Sinon, comme le statut de terminaison est $b = \textit{false}$, la branche `else` est exécutée à l'instant suivant.

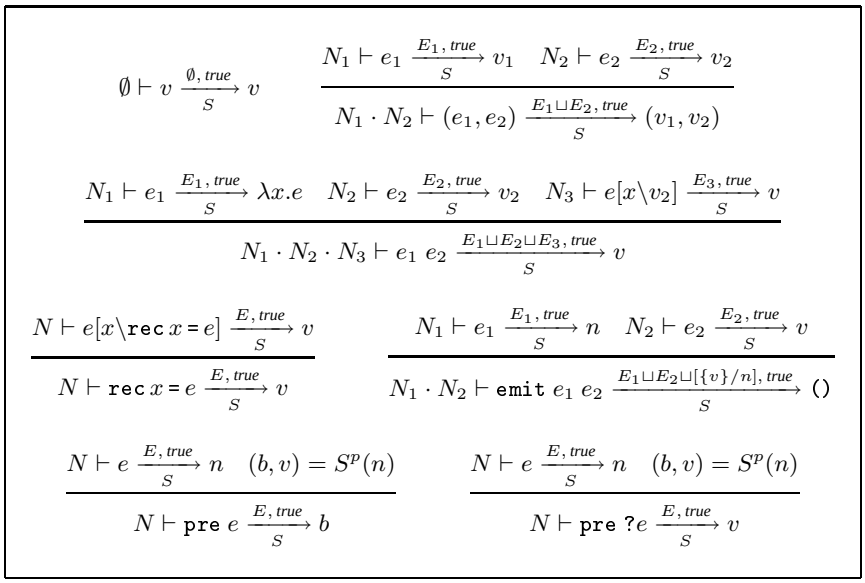


Figure 7. *Sémantique comportementale (1) : expressions instantanées*

– `run` e évalue e en une définition de processus et l'exécute.

La sémantique des instructions de suspension et de préemption est donnée dans la figure 9.

– Le `do/when` exécute son corps uniquement quand le signal qui le contrôle est présent. Quand le corps est actif et termine son exécution, le `do/when` termine aussi instantanément. Dans chaque règle on évalue le signal mais à la première activation l'expression e est évaluée en une valeur n. Donc pour les activations aux instants suivant le `do/when` sera toujours contrôlé par le même signal n.

– Enfin, le `do/until` active toujours son corps. Si le corps termine, le `do/until` se réécrit en la valeur de son corps. Sinon, il y a préemption si le signal n est émis ($n \in S$) et sa valeur associée peut être filtrée par le motif p ($p \preceq S^v(n)$). Si la variable introduite par le motif p est x, alors $e[p \backslash v]$ est égal à e dans laquelle x est substituée par v ($e[x \backslash v]$).

5.2. *Déterminisme et unicité*

Nous présentons maintenant les propriétés principales de la sémantique comportementale. La première est le *déterminisme* : dans un environnement de signaux donné, un programme ne peut réagir que d'une seule façon. La seconde propriété que nous nommons *unicité* dit que si un programme est réactif, alors il existe un unique plus pe-

$$\frac{N_1 \vdash e_1 \xrightarrow[S]{E_1, b_1} e_1' \quad N_2 \vdash e_2 \xrightarrow[S]{E_2, b_2} e_2' \quad b_1 \wedge b_2 = \mathit{false}}{N_1 \cdot N_2 \vdash \texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e \xrightarrow[S]{E_1 \sqcup E_2, \mathit{false}} \texttt{let}\ x_1 = e_1'\ \texttt{and}\ x_2 = e_2'\ \texttt{in}\ e}$$

$$\frac{N_1 \vdash e_1 \xrightarrow[S]{E_1, \mathit{true}} v_1 \quad N_2 \vdash e_2 \xrightarrow[S]{E_2, \mathit{true}} v_2 \quad N_3 \vdash e[x_1 \backslash v_1, x_2 \backslash v_2] \xrightarrow[S]{E, b} e'}{N_1 \cdot N_2 \cdot N_3 \vdash \texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e \xrightarrow[S]{E_1 \sqcup E_2 \sqcup E, b} e'}$$

$$\frac{\begin{array}{c} N_1 \vdash e_1 \xrightarrow[S]{E_1, \mathit{true}} v_1 \quad N_2 \vdash e_2 \xrightarrow[S]{E_2, \mathit{true}} v_2 \quad S(n) = (v_1, v_2, (\mathit{false}, v_1), m) \\ N_3 \vdash e[x \backslash n] \xrightarrow[S]{E, b} e' \end{array}}{N_1 \cdot N_2 \cdot N_3 \cdot \{n\} \vdash \texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e \xrightarrow[S]{E_1 \sqcup E_2 \sqcup E, b} e'}$$

$$\frac{N_1 \vdash e \xrightarrow[S]{E, \mathit{true}} n \quad n \in S \quad N_2 \vdash e_1 \xrightarrow[S]{E_1, b} e_1'}{N_1 \cdot N_2 \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[S]{E \sqcup E_1, b} e_1'}$$

$$\frac{N \vdash e \xrightarrow[S]{E, \mathit{true}} n \quad n \notin S}{N \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[S]{E, \mathit{false}} e_2}$$

$$\frac{N_1 \vdash e \xrightarrow[S]{E, \mathit{true}} \texttt{process}\ e_1 \quad N_2 \vdash e_1 \xrightarrow[S]{E_1, b} e_1'}{N_1 \cdot N_2 \vdash \texttt{run}\ e \xrightarrow[S]{E \sqcup E_1, b} e_1'}$$

Figure 8. *Sémantique comportementale (2) : expressions réactives*

tit environnement de signaux dans lequel il peut réagir. Nous reprenons la définition de réactif donnée dans (Berry, 1998) : il existe au moins un environnement de signaux S tel que $N \vdash e \xrightarrow[S]{E, b} e'$.

La combinaison de ces deux propriétés garantit que tous les programmes réactifs sont corrects. Cette propriété n'est pas vraie en ESTEREL. Elle montre qu'il n'y a pas besoin d'analyse de causalité en REACTIVEML, i.e, tous les programmes sont causaux.

Rappelons que ces propriétés sont données sur le noyau du langage qui ne contient pas les effets de bord et présente une version limitée du filtrage dans le `do/until`. Nous reviendrons sur le détermine de l'ensemble du langage dans la section 9.3.

$$\frac{N \vdash e \xrightarrow[S]{E,\,true} n \quad n \notin S}{N \vdash \texttt{do}\ e_1\ \texttt{when}\ e \xrightarrow[S]{E,\,false} \texttt{do}\ e_1\ \texttt{when}\ n}$$

$$\frac{N \vdash e \xrightarrow[S]{E,\,true} n \quad n \in S \quad N_1 \vdash e_1 \xrightarrow[S]{E_1,\,false} e_1'}{N \cdot N_1 \vdash \texttt{do}\ e_1\ \texttt{when}\ e \xrightarrow[S]{E \sqcup E_1,\,false} \texttt{do}\ e_1'\ \texttt{when}\ n}$$

$$\frac{N \vdash e \xrightarrow[S]{E,\,true} n \quad n \in S \quad N_1 \vdash e_1 \xrightarrow[S]{E_1,\,true} v}{N \cdot N_1 \vdash \texttt{do}\ e_1\ \texttt{when}\ e \xrightarrow[S]{E \sqcup E_1,\,true} v}$$

$$\frac{N \vdash e \xrightarrow[S]{E,\,true} n \quad N_1 \vdash e_1 \xrightarrow[S]{E_1,\,true} v}{N \cdot N_1 \vdash \texttt{do}\ e_1\ \texttt{until}\ e(p)\ \texttt{->}\ e_2 \xrightarrow[S]{E \sqcup E_1,\,true} v}$$

$$\frac{N \vdash e \xrightarrow[S]{E,\,true} n \quad N_1 \vdash e_1 \xrightarrow[S]{E_1,\,false} e_1' \quad n \in S \quad p \preceq S^v(n)}{N \cdot N_1 \vdash \texttt{do}\ e_1\ \texttt{until}\ e(p)\ \texttt{->}\ e_2 \xrightarrow[S]{E \sqcup E_1,\,false} e_2[p \backslash S^v(n)]}$$

$$\frac{N \vdash e \xrightarrow[S]{E,\,true} n \quad N_1 \vdash e_1 \xrightarrow[S]{E_1,\,false} e_1' \quad n \notin S \vee p \not\preceq S^v(n)}{N \cdot N_1 \vdash \texttt{do}\ e_1\ \texttt{until}\ e(p)\ \texttt{->}\ e_2 \xrightarrow[S]{E \sqcup E_1,\,false} \texttt{do}\ e_1'\ \texttt{until}\ n(p)\ \texttt{->}\ e_2}$$

Figure 9. *Sémantique comportementale (3) : expressions de contrôle*

Propriété 1 (Déterminisme). *Pour toute expression e, la sémantique comportementale est déterministe. C'est à dire :*
$\forall e, \forall S, \forall N.$ *si* $\forall n \in Dom(S).\ S^g(n) = f$ *et* $f(x, f(y, z)) = f(y, f(x, z))$
et $N \vdash e \xrightarrow[S]{E_1,\,b_1} e_1'$ *et* $N \vdash e \xrightarrow[S]{E_2,\,b_2} e_2'$
alors $E_1 = E_2$ *et* $b_1 = b_2$ *et* $e_1' = e_2'$.

Démonstration. La preuve se fait par induction sur les dérivations. Nous ne présentons que le cas le plus intéressant de la preuve.
Cas `do` e_1 `until` $e(p)$ `->` e_2 **avec préemption :** Supposons que l'on ait les deux dérivations suivantes :

$$\frac{N \vdash e \xrightarrow[S]{E_1,\,true} n_1 \quad N_1 \vdash e_1 \xrightarrow[S]{E_{1_1},\,true} e_{1_1}' \quad n_1 \in S \quad p \preceq S^v(n_1)}{N \cdot N_1 \vdash \texttt{do}\ e_1\ \texttt{until}\ e(p)\ \texttt{->}\ e_2 \xrightarrow[S]{E_1 \sqcup E_{1_1},\,false} e_2[p \backslash S^v(n_1)]}$$

$$\frac{N \vdash e \xrightarrow[S]{E_2,\, true} n_2 \quad N_1 \vdash e_1 \xrightarrow[S]{E_{1_2},\, true} e'_{1_2} \quad n_2 \in S \quad p \preceq S^v(n_2)}{N \cdot N_1 \vdash \texttt{do}\ e_1\ \texttt{until}\ e(p)\ \texttt{->}\ e_2 \xrightarrow[S]{E_2 \sqcup E_{1_2},\, false} e_2[p \backslash S^v(n_2)]}$$

Par induction on a $E_1 = E_2$ et $n_1 = n_2$ et également $E_{1_1} = E_{1_2}$ et $e'_{1_1} = e'_{1_2}$ donc $S(n_1) = S(n_2) = (d, g, pre, m)$. Avec la propriété d'associativité et de commutativité de la fonction de combinaison g, nous sommes sûrs que *fold* est déterministe, donc $S^v(n_1) = S^v(n_2)$. Par conséquence, $E_1 \sqcup E_{1_1} = E_2 \sqcup E_{1_2}$ et $e_2[p \backslash S^v(n_1)] = e_2[p \backslash S^v(n_2)]$. □

L'associativité et la commutativité des fonctions de combinaison expriment le fait qu'elles ne doivent pas dépendre de l'ordre des émissions pendant l'instant. C'est une contrainte assez forte mais même si elle n'est pas satisfaite le programme peut être déterministe. Par exemple, s'il n'y a pas de multi-émission, la fonction de combinaison n'a pas à être associative et commutative. Ou si la fonction de combinaison construit la liste des valeurs émises et que toutes les opérations faites sur cette liste ne dépendent pas de l'ordre des éléments, alors le programme reste déterministe.

Nous donnons maintenant la propriété d'unicité.

Propriété 2 (Unicité). *Pour toute expression e, soit $\mathcal{S}$ l'ensemble des environnements de signaux tel que $\mathcal{S} = \left\{ S \mid \exists N, E, b.\ N \vdash e \xrightarrow[S]{E,\, b} e' \right\}$ alors il existe un unique plus petit environnement ($\sqcap\mathcal{S}$) tel que $\exists N, E, b.\ N \vdash e \xrightarrow[\sqcap\mathcal{S}]{E,\, b} e'$*

Démonstration. La preuve de cette propriété est basée sur le lemme suivant qui dit que si un programme peut réagir dans deux environnements de signaux différents, alors il peut réagir dans l'intersection de ces deux environnements.

De plus, $(S, \sqsubseteq, \sqcup, \sqcap)$ définit un treillis avec un minimum. Donc l'intersection de tous les environnements dans lesquels l'expression peut réagir est unique. □

Ce lemme est basé sur l'absence de réaction instantanée à l'absence d'événement. Cette caractéristique de la sémantique garantit que l'absence d'un signal ne peut pas générer des émissions.

Lemme 1. *Pour toute expression e, soit S_1 et S_2 deux environnements dans lesquels e peut réagir : $N_1 \vdash e \xrightarrow[S_1]{E_1,\, b_1} e_1$ et $N_2 \vdash e \xrightarrow[S_2]{E_2,\, b_2} e_2$.*

Soit S_3 tel que $S_3^m = S_1^m \sqcap S_2^m$. Alors il existe E_3, b_3 et e_3 tels que

$$N_3 \vdash e \xrightarrow[S_3]{E_3,\, b_3} e_3 \quad \textit{et} \quad b_3 \Rightarrow (b_1 \wedge b_2) \quad \textit{et} \quad E_3 \sqsubseteq (E_1 \sqcap E_2) \quad \textit{et} \quad N_3 \subseteq (N_1 \cap N_2)$$

La preuve est donnée dans (Mandel, 2006). Elle se fait par induction sur les dérivations.

$$\lambda x.e\ v/S \rightarrow_\varepsilon e[x\backslash v]/S \qquad \texttt{rec}\ x = e/S \rightarrow_\varepsilon e[x\backslash \texttt{rec}\ x = e]/S$$

$$\texttt{run}\ (\texttt{process}\ e)/S \rightarrow_\varepsilon e/S \qquad \texttt{let}\ x_1 = v_1\ \texttt{and}\ x_2 = v_2\ \texttt{in}\ e/S \rightarrow_\varepsilon e[x_1\backslash v_1, x_2\backslash v_2]/S$$

$$\texttt{emit}\ n\ v/S \rightarrow_\varepsilon ()/S + [v/n] \qquad \texttt{present}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2/S \rightarrow_\varepsilon e_1/S \text{ si } n \in S :$$

si $n \notin Dom(S)$

$$\texttt{signal}\ x\ \texttt{default}\ v_1\ \texttt{gather}\ v_2\ \texttt{in}\ e/S \rightarrow_\varepsilon e[x\backslash n]/S[(v_1, v_2, (\textit{false}, v_1), \emptyset)/n]$$

$$\texttt{do}\ v\ \texttt{until}\ n(p)\ \texttt{->}\ e/S \rightarrow_\varepsilon v/S \qquad \texttt{do}\ v\ \texttt{when}\ n/S \rightarrow_\varepsilon v/S \text{ si } n \in S$$

$$\texttt{pre}\ n/S \rightarrow_\varepsilon b/S \text{ si } S^p(n) = (b, v) \qquad \texttt{pre}\ \texttt{?}n/S \rightarrow_\varepsilon v/S \text{ si } S^p(n) = (b, v)$$

Figure 10. *Réduction en tête*

6. Sémantique opérationnelle

La sémantique comportementale que nous venons de définir ne peut pas être implantée simplement. Elle suppose la connaissance *a priori* de tous les signaux qui vont être émis par la réaction. Nous présentons ici une sémantique à petits pas où la réaction construit l'environnement de signaux.

La sémantique opérationnelle est décomposée en deux étapes. La première décrit la réaction pendant l'instant comme une succession de micro-réactions. La seconde étape, appelée *réaction de fin d'instant* prépare la réaction pour l'instant suivant.

6.1. *Sémantique à réduction*

La première étape de la sémantique est une extension de la sémantique à réduction de ML. La réaction d'un instant est représentée par une succession de réactions de la forme $e/S \rightarrow e'/S'$. Ces réductions définissent la réaction du programme tout en construisant l'ensemble des valeurs émises.

Pour définir la réaction $\rightarrow$, on commence par se donner des axiomes pour la relation de réduction en tête de terme ($\rightarrow_\varepsilon$). Ces axiomes sont donnés dans la figure 10.

Parmi ces règles, on peut remarquer que le `present` ne peut être réduit que si le signal est présent. La construction `signal/in` alloue un nouveau signal initialisé comme étant absent.

Nous définissons maintenant la réduction en profondeur ($\rightarrow$) :

$$\frac{e/S \rightarrow_{\varepsilon} e'/S'}{\Gamma(e)/S \rightarrow \Gamma(e')/S'} \qquad \frac{n \in S \quad e/S \rightarrow e'/S'}{\Gamma(\texttt{do}\ e\ \texttt{when}\ n)/S \rightarrow \Gamma(\texttt{do}\ e'\ \texttt{when}\ n)/S'}$$

où Γ est un contexte d'évaluation. Dans la première règle, l'expression e se réduit en tête, donc elle peut se réduire dans n'importe quel contexte. La seconde règle définit la suspension. Elle montre que le corps d'un `do/when` ne peut être évalué que si le signal est présent.

Les contextes d'évaluation sont définis de la façon suivante :

$$\begin{array}{rcl}
\Gamma & ::= & [\,] \mid \Gamma\ e \mid e\ \Gamma \mid (\Gamma, e) \mid (e, \Gamma) \mid \texttt{run}\ \Gamma \mid \texttt{pre}\ \Gamma \mid \texttt{pre ?}\Gamma \\
& & \mid \texttt{let}\ x\ \texttt{=}\ \Gamma\ \texttt{and}\ x\ \texttt{=}\ e\ \texttt{in}\ e \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{and}\ x\ \texttt{=}\ \Gamma\ \texttt{in}\ e \\
& & \mid \texttt{emit}\ \Gamma\ e \mid \texttt{emit}\ e\ \Gamma \mid \texttt{present}\ \Gamma\ \texttt{then}\ e\ \texttt{else}\ e \\
& & \mid \texttt{signal}\ x\ \texttt{default}\ \Gamma\ \texttt{gather}\ e\ \texttt{in}\ e \\
& & \mid \texttt{signal}\ x\ \texttt{default}\ e\ \texttt{gather}\ \Gamma\ \texttt{in}\ e \mid \texttt{do}\ e\ \texttt{when}\ \Gamma \\
& & \mid \texttt{do}\ e\ \texttt{until}\ \Gamma(p)\ \texttt{->}\ e \mid \texttt{do}\ \Gamma\ \texttt{until}\ n(p)\ \texttt{->}\ e
\end{array}$$

Si on étudie les contextes des définitions parallèles (`let/and/in`), on constate que l'ordre d'évaluation des deux définitions n'est pas spécifié. Dans l'implantation de REACTIVEML, le choix de l'ordonnancement est fixé de sorte que l'exécution soit toujours reproductible d'une exécution à l'autre. Nous reviendrons plus tard (section 9) sur l'ordre d'exécution du parallèle et les choix faits dans les autres langages.

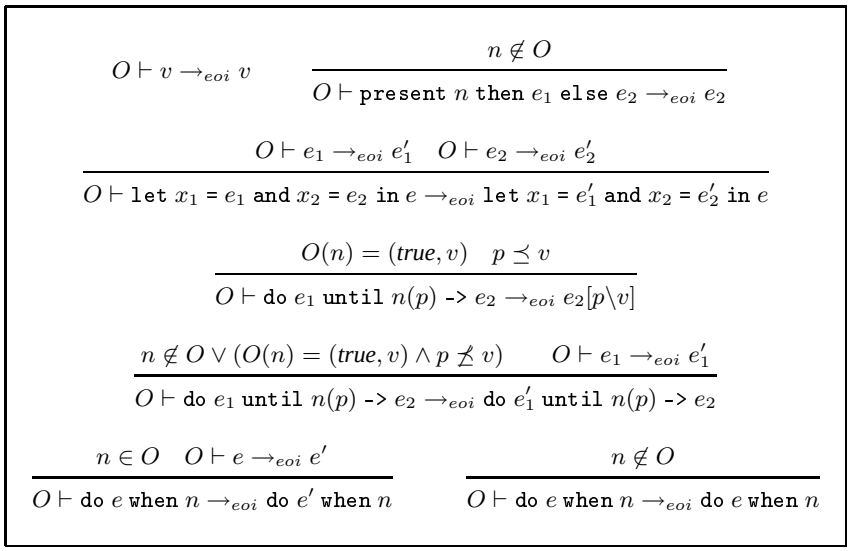
On note également que $\texttt{do}\ \Gamma\ \texttt{when}\ n$ n'est pas un contexte car on ne veut pas pouvoir évaluer dans un `do/when` lorsque le signal n est absent.

6.2. *Réaction de fin d'instant*

Le modèle réactif est basé sur l'absence de réaction instantanée à l'absence d'un signal. Il faut donc attendre la fin d'instant pour traiter l'absence des signaux et préparer le programme pour l'instant suivant.

La réaction de l'instant s'arrête lorsqu'il n'y a plus de réductions $\rightarrow$ possibles. À partir de ce moment, l'environnement des signaux est figé, il ne peut plus y avoir de nouvelles émissions. Les signaux qui n'ont pas été émis sont supposés absents. On peut alors calculer la sortie O du programme avec la fonction $next$ précédemment définie.

Les règles pour le traitement de fin d'instant ont la forme suivante : $O \vdash e \rightarrow_{eoi} e'$. Leur définition est donnée figure 11. Nous pouvons remarquer que les règles ne sont données que pour un sous-ensemble d'expressions car elles seront appliquées seulement quand l'expression e ne peut plus être réduite par $\rightarrow$. Nous appelons ces expressions des *expressions de fin d'instant*.



$$O \vdash v \rightarrow_{eoi} v \qquad \frac{n \notin O}{O \vdash \texttt{present}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightarrow_{eoi} e_2}$$

$$\frac{O \vdash e_1 \rightarrow_{eoi} e_1' \quad O \vdash e_2 \rightarrow_{eoi} e_2'}{O \vdash \texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e \rightarrow_{eoi} \texttt{let}\ x_1 = e_1'\ \texttt{and}\ x_2 = e_2'\ \texttt{in}\ e}$$

$$\frac{O(n) = (\mathit{true}, v) \quad p \preceq v}{O \vdash \texttt{do}\ e_1\ \texttt{until}\ n(p)\ \texttt{->}\ e_2 \rightarrow_{eoi} e_2[p \backslash v]}$$

$$\frac{n \notin O \vee (O(n) = (\mathit{true}, v) \wedge p \not\preceq v) \qquad O \vdash e_1 \rightarrow_{eoi} e_1'}{O \vdash \texttt{do}\ e_1\ \texttt{until}\ n(p)\ \texttt{->}\ e_2 \rightarrow_{eoi} \texttt{do}\ e_1'\ \texttt{until}\ n(p)\ \texttt{->}\ e_2}$$

$$\frac{n \in O \quad O \vdash e \rightarrow_{eoi} e'}{O \vdash \texttt{do}\ e\ \texttt{when}\ n \rightarrow_{eoi} \texttt{do}\ e'\ \texttt{when}\ n} \qquad \frac{n \notin O}{O \vdash \texttt{do}\ e\ \texttt{when}\ n \rightarrow_{eoi} \texttt{do}\ e\ \texttt{when}\ n}$$

Figure 11. *Réaction de fin d'instant*

6.3. *Exécution d'un programme*

L'exécution d'un instant est définie par la relation :

$$e_i/S_i \Rightarrow e_i'/S_i'$$

Si on note I_i et O_i les entrées et les sorties de la réaction de l'instant, l'environnement des signaux doit avoir les propriétés suivantes. Tous les signaux sont par défaut absents sauf les signaux donnés en entrée ($I_i = S_i^m$). Les sorties sont calculées à partir de l'environnement à la fin de la réaction ($O_i = next(S_i')$). Les valeurs par défaut, les fonctions de combinaison des signaux sont conservées d'un instant à l'autre ($S_i'^d = S_{i+1}^d$ et $S_i'^g = S_{i+1}^g$). Les sorties définissent le `pre` de l'instant suivant ($O_i = S_{i+1}^p$).

L'exécution d'un instant se décompose en deux étapes : la réduction de e_i jusqu'à l'obtention d'un point fixe e_i'', puis la préparation à l'instant suivant.

$$\frac{e_i/S_i \hookrightarrow e_i''/S_i' \quad O_i = next(S_i') \quad O_i \vdash e_i'' \rightarrow_{eoi} e_i'}{e_i/S_i \Rightarrow e_i'/S_i'}$$

où $e/S \hookrightarrow e'/S'$ si $e/S \rightarrow^* e'/S'$ et $e'/S' \not\rightarrow$. La relation $\rightarrow^*$ est la fermeture réflexive et transitive de $\rightarrow$.

6.4. *Sûreté du typage*

À partir de cette sémantique opérationnelle, nous pouvons prouver la sûreté du typage en utilisant des techniques classiques (Pierce, 2002).

La preuve se décompose en deux parties : la sûreté du typage de la réduction $\rightarrow$ et la préservation du typage pour la réduction $\rightarrow_{eoi}$. Nous énonçons ici seulement ces deux propriétés, les preuves étant disponibles dans (Mandel, 2006).

Propriété 3 (Sûreté du typage). *Si $TC \vdash e : \tau$ et $k \vdash e$ et $e/S \rightarrow^* e'/S'$ et e'/S' est en forme normale vis-à-vis de $\rightarrow$, alors e' est une expression de fin d'instant.*

Notons que dans notre définition de la sûreté du typage, les formes normales ne sont pas des valeurs mais des expressions de fin d'instant.

Propriété 4 (Préservation du typage par réduction $\rightarrow_{eoi}$). *Si $H \vdash S$ et $O = next(S)$ et $O \vdash e \rightarrow_{eoi} e'$ alors $e/S \sqsubseteq e'/S$.*

7. Équivalence entre les sémantiques

Nous montrons dans cette partie que la sémantique opérationnelle est équivalente à la sémantique comportementale. Cette propriété construit un cadre sémantique large permettant de choisir le formalisme le plus adapté à la propriété à montrer.

Nous commençons par prouver que si une expression e réagit en une expression e' avec la sémantique à petits pas, alors elle peut réagir dans le même environnement de signaux avec la sémantique grands pas.

Théorème 1. *Pour tout environnement S_{init} et expression e tels que $e/S_{init} \Rightarrow e'/S$, alors il existe N, E et b tels que $N \vdash e \xrightarrow[S]{E,b} e'$ avec $E = S^m \backslash S^m_{init}$.*

Démonstration. La preuve se fait par induction sur le nombre de réductions $\rightarrow$ dans $e/S_{init} \Rightarrow e'/S$. L'idée de la preuve est de supprimer à chaque étape d'induction la dernière réduction $\rightarrow$.

– S'il n'y a pas de réductions $\rightarrow$ possibles, il faut montrer que la réduction $\rightarrow_{eoi}$ est équivalente à la sémantique grands pas (lemme 2).

– S'il y a au moins une réduction $\rightarrow$, on doit montrer qu'une réduction $\rightarrow$ suivie d'une réaction grands pas est équivalente à une réaction grands pas (lemme 3).

Le diagramme de la figure 12 représente la structure de la preuve. Les flèches pleines sont quantifiées universellement et les flèches hachurées le sont existentiellement. Les diagrammes A et B correspondent respectivement aux lemmes 2 et 3. □

La preuve de ce lemme est basée sur les propriétés suivantes :

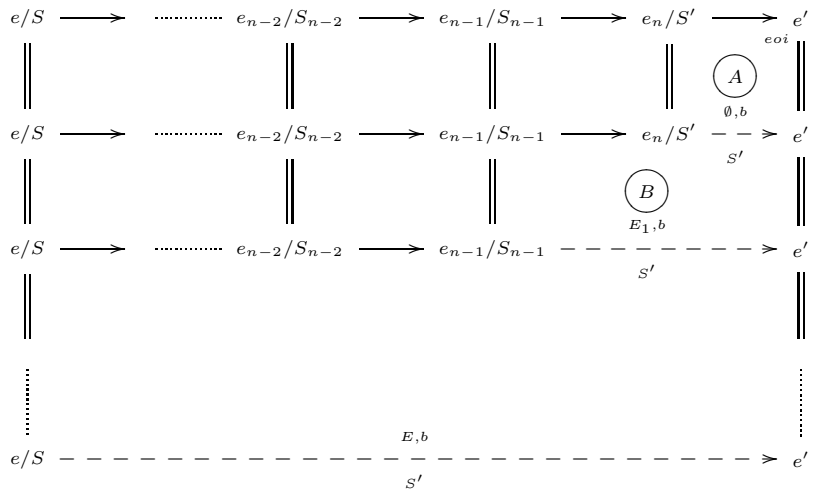


Figure 12. *Structure de la preuve du théorème 1*

Lemme 2. *Si* $e/S \not\rightarrow$ *et* $S \vdash e \rightarrow_{eoi} e'$*, alors il existe* N *et* b *tel que* $N \vdash e \xrightarrow[S]{\emptyset, b} e'$.

Démonstration. La preuve se fait par induction sur la structure de e. □

Lemme 3. *Si* $e/S_0 \rightarrow e_1/S_1$ *et* $N \vdash e_1 \xrightarrow[S]{E', b} e'$ *avec* $S_1 \sqsubseteq S$
alors $N \vdash e \xrightarrow[S]{E, b} e'$ *avec* $E = E' \sqcup (S_1^m \backslash S_0^m)$.

Démonstration. La preuve se fait en deux étapes. On montre d'abord la même propriété pour la réduction $\rightarrow_\varepsilon$. Puis on montre que cette propriété est vraie dans tout contexte Γ. □

On montre maintenant que si un programme peut réagir dans le même environnement avec les deux sémantiques, alors les expressions obtenues à la fin des réactions sont les mêmes. On a donc l'équivalence entre les deux sémantiques.

Théorème 2. *Pour chaque* S_{init} *et* e *tels que :*

- $N \vdash e \xrightarrow[S_1]{E_1, b_1} e_1$ *où* S_1 *est le plus petit environnement tel que* $S_{init} \sqsubseteq S_1$
- $e/S_{init} \Rightarrow e_2/S_2$
- $\forall n \in Dom(S_2) : S_2^q(n) = f$ *et* $f(x, f(y, z)) = f(y, f(x, z))$,

alors $e_1 = e_2$ *et* $S_1 = S_2$

Démonstration. Avec le théorème 1, il existe E_2 et b_2 tels que $N \vdash e \xrightarrow[S_2]{E_2, b_2} e_2$ et on peut remarquer, par construction, que S_2 est le plus petit environnement tel que $S_{init} \sqsubseteq S_2$.

Avec la propriété d'unicité (propriété 2), nous savons qu'il existe un unique plus petit environnement de signaux dans lequel une expression peut réagir avec la sémantique comportementale donc $S_1 = S_2$. Maintenant, avec le déterminisme de la sémantique (propriété 1) on a $E_1 = E_2$, $b_1 = b_2$ et $e_1 = e_2$. □

8. Travaux connexes

Nous avons évoqué la filiation directe de ce travail avec ceux de l'école de la programmation synchrone (Benveniste *et al.*, 2003). D'autres langages ont également considéré l'extension d'un langage fonctionnel avec des traits lui permettant de décrire des systèmes réactifs (Elliott *et al.*, 1997; Wan *et al.*, 2000). Ces langages sont des langages avec appel par nom qui sont basés sur un modèle flot de données. Au contraire, REACTIVEML est un langage avec appel par valeur basé sur un modèle flot de contrôle. Nous discutons ici des travaux directement liés aux modèle réactif synchrone.

REACTIVEC (Boussinot, 1991) est la première implantation du modèle réactif faite par Frédéric Boussinot. Il s'agit d'une extension du langage C avec des procédures réactives. Le langage propose des constructions de bas niveau où les communications se font par mémoire partagée. Une bibliothèque STANDARD ML implante ce modèle (Pucella, 1998). Les SUGARCUBES (Boussinot *et al.*, 1998) et son noyau JUNIOR (Hazard *et al.*, 1999) sont deux bibliothèques pour la programmation réactive en JAVA. Les constructions réactives se présentent comme un ensemble de classes. REJO (Acosta-Bermejo, 2003) est une extension de JAVA se compilant vers JUNIOR. REACTIVEML se distingue de ces travaux par sa construction au dessus d'un langage fonctionnel typé et par son traitement des signaux. En REACTIVEML, les signaux suivent un mécanisme de liaison statique (vs dynamique pour les SUGARCUBES). Ce choix se marie mieux avec OCAML et conduit à une implantation plus efficace, l'allocation/désallocation des signaux pouvant être confiée au glaneur de cellules de OCAML.

REACTIVEML reprend le modèle de la concurrence réactive introduite dans SL (Boussinot *et al.*, 1996). Il la complète en donnant une sémantique complète au modèle réactif prenant en compte les signaux valués et l'interaction avec un langage hôte.

La bibliothèque des FAIR THREADS (Serrano *et al.*, 2004) et le langage associé LOFT (Boussinot, 2003) permettent de mélanger des threads coopératifs et préemptifs. Dans ce modèle, plusieurs ordonnanceurs réactifs peuvent être exécutés de façon asynchrone et les threads peuvent changer d'ordonnanceur en cours d'exécution et passer du mode synchrone au mode asynchrone. ULM (Boudol, 2004) à une ap-

proche similaire aux FAIR THREADS mais dans un cadre distribué. Dans ce modèle, plusieurs sites exécutent des ordonnanceurs réactifs et les threads peuvent migrer d'un site à un autre. REACTIVEML ne considère pas le mélange coopératif/préemptif ni les mécanismes de migration.

Citons enfin des travaux récents sur le modèle réactif à base de threads et visant à définir un noyau minimal (Amadio *et al.*, 2005). L'un des objectifs est d'établir des principes de preuves de programmes réactifs dans l'esprit de ce qui a été conduit dans les calculs de processus issus du π-calcul. Cet aspect n'a pas été abordé dans notre travail.

9. Discussion

9.1. *Le statut de* `emit`

Le critère choisi pour la séparation entre expressions instantanées et réactives est basé sur le nombre d'instants nécessaires pour exécuter les expressions. Une expression qui est toujours exécutée en un instant est instantanée, sinon c'est une expression réactive. L'objectif de cette séparation est à la fois de mélanger arbitrairement les constructions de OCAML et les constructions réactives tout en garantissant que les parties OCAML resteront inchangées lors du processus de compilation. Cette séparation permet ainsi d'utiliser les principales constructions de OCAML (e.g., `if/then/else`, `match/with`) à la fois pour composer des expressions OCAML et des expressions réactives.

Dans cet article, nous avons considéré que l'expression `emit` était une expression instantanée. Nous n'avons pas toujours fait ce choix. Dans (Mandel *et al.*, 2005b), les expressions instantanées sont les expressions purement ML. Cette différence a des répercutions sur la présentation de la sémantique et sur l'expressivité du langage.

D'un point de vue sémantique, interdire la construction `emit` dans les expressions instantanées permet de présenter REACTIVEML comme un langage à deux niveaux. Le langage hôte garde sa sémantique inchangée, et nous proposons par dessus de nouvelles constructions réactives qui peuvent utiliser des constructions du langage hôte. Par exemple dans la sémantique comportementale, la règle pour `emit` s'écrit alors :

$$\frac{e_1 \Downarrow n \qquad e_2 \Downarrow v}{\emptyset \vdash \texttt{emit}\ e_1\ e_2 \xrightarrow[S]{[\{v\}/n],\ \mathit{true}} \texttt{()}}$$

où la réduction $e \Downarrow v$ est la réaction d'une expression ML. Un des avantages de cette approche est de conserver toutes les propriétés du langage hôte. Mais en contrepartie, les constructions qui sont présentes dans les deux niveaux comme `let/in` sont dupliquées.

Dans notre approche, il n'y a plus de séparation entre le langage hôte et la partie réactive. Ceci oblige à refaire les preuves des propriétés des expressions purement ML.

En contrepartie, il n'y a qu'un seul ensemble de règles, ce qui donne une présentation unifiée de la sémantique.

D'un point de vue de la conception du langage, nous avons trouvé à l'usage plus naturel de considérer que la déclaration et l'émission de signal soient des expressions instantanées. Il est commode, par exemple, de pouvoir utiliser les fonctions de la librairie standard de OCAML pour manipuler les listes de signaux. Ainsi, émettre tous les signaux d'une liste s'écrit :

```
let emit_list l = List.iter (fun x -> emit x) l
```

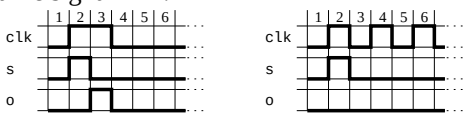
9.2. *Sémantique du* `pre`

L'expression `pre s` s'évalue en `true` si le signal était présent à l'instant précédent et en `false` sinon. Cette expression se compose mal avec la suspension. Illustrons cela avec le processus `danger` :

```
let process danger o =
  signal s in
  emit s; pause;
  if pre s then emit o
```

Même si `s` est local au processus `danger`, la valeur de `pre s` dépend du contexte dans lequel le processus est exécuté. Nous présentons deux exécutions de `danger` lorsqu'il est défini dans un `do/when` contrôlé par le signal `clk`.

```
do run (danger o) when clk
```



Dans le chronogramme de gauche, le signal `clk` est présent pendant deux instants successifs. Dans ce cas, le signal `o` est émis (à l'instant 3, `pre s` est vrai). Dans le chronogramme de droite, `clk` est présent un instant sur deux. Cette fois ci, le signal `o` n'est pas émis car à l'instant 4, lorsque l'expression `pre s` est évaluée, le statut de `s` à l'instant précédent est `false`.

En ESTEREL le comportement du processus `danger` est différent, quelque soit le contexte dans lequel il est exécuté, il émet toujours le signal `o`. Cette différence vient de l'*horloge* des signaux. En ESTEREL, un signal a l'horloge de son contexte de définition. Cela signifie que le statut précédent d'un signal est le dernier statut qu'avait le signal lorsque son contexte de définition était actif. Un signal est défini uniquement si son contexte de définition est actif. En REACTIVEML, tous les signaux sont définis à tous les instants. On dit qu'il sont sur l'horloge de base.

Cette différence entre ESTEREL et REACTIVEML vient du phénomène d'échappement de portée qui existe en REACTIVEML mais pas en ESTEREL. Avec l'échappement de portée un signal peut sortir de son contexte de définition et donc être émis sur une horloge plus rapide que celle sur laquelle le signal est défini. C'est le cas par exemple du programme suivant :

```
let process scope_extrusion clk =
  signal x in
  do
    signal s in
    emit x s;
    await s
  when clk
  ||
  await x ([y]) in loop emit y; pause end
```

`pre` est la seule construction de REACTIVEML qui a un problème de compositionalité. Nous travaillons sur une autre construction qui pourrait la remplacer.

9.3. *Déterminisme*

Dans cette section, nous revenons sur la question du déterminisme. Nous allons voir que la propriété de déterminisme disparait en présence d'effets de bord ou en augmentant l'expressivité des motifs dans le `do/until` au profit d'une propriété plus faible de reproductibilité de l'exécution. Notons que la propriété de déterminisme disparait aussi dans un langage tel que ESTEREL dès lors que le langage importe des fonctions externes qui peuvent potentiellement effectuer des effets de bord.

9.3.1. *Parallèle commutatif et effets de bord*

La modification en parallèle d'une ressource partagée (référence, écran ...) a un comportement non déterministe : la composition parallèle est commutative. Ainsi, l'évaluation de l'expression suivante peut afficher `1` ou `2`.

```
let x = ref 0 in (x := 1 || x := 2); print_int !x
```

Ce choix d'un opérateur commutatif a été guidé par la sémantique à grands pas qui ne spécifie pas l'ordonnancement dans un instant. Un choix différent aurait pu être fait. Par exemple, en SUGARCUBES, l'opérateur `merge` de composition parallèle garantit que la branche gauche est toujours activée avant la branche droite. Cette opérateur est déterministe.

Nous illustrons sur l'exemple suivant que le déterminisme de la composition parallèle n'aide pas à raisonner sur les programmes car l'ordonnancement d'une expression n'est pas conservé par composition parallèle.

```
let process p s1 s2 s3 =
     await immediate s3; print_int 3
  || await immediate s2; print_int 2; emit s3
  || await immediate s1; print_int 1; emit s2
  || emit s1
```

On exécute ce processus avec un ordonnancement de gauche à droite dans trois contextes différents. Au premier instant le processus `p` est exécuté seul, au deuxième instant il est exécuté en parallèle avec l'émission de `s2` à sa droite et au troisième ins-

tant avec l'émission de s2 à sa gauche :

```
let process main =
  signal s1, s2, s3 in
  print_string "Instant␣1␣:␣"; run (p s1 s2 s3); pause;
  print_string ";␣Instant␣2␣:␣"; (run (p s1 s2 s3) || emit s2); pause;
  print_string ";␣Instant␣3␣:␣"; (emit s2 || run (p s1 s2 s3))
```

Avec cet ordonnancement, la sortie affichée par ce processus est la suivante :

```
Instant 1 : 123; Instant 2 : 213; Instant 3 : 231
```

Même avec une sémantique déterministe, l'exécution de p produit des résultats différents. C'est pour cela que nous avons choisi de garder un opérateur de composition parallèle ne spécifiant pas d'ordre d'évaluation.

Enfin, il faut remarquer que dans le modèle réactif, l'exécution des expressions instantanées est atomique. Ainsi nous évitons les problèmes liés aux threads avec ordonnancement préemptif où il faut définir des sections critiques pour pouvoir modifier des ressources partagées.

9.3.2. *Filtrage de signaux*

Dans le noyau du langage introduit dans la section 3 il est possible d'attendre qu'un signal soit émit *exactement* une fois au cours d'un instant. Ainsi, le programme suivant affiche 3 car au premier instant deux valeurs sont émises sur s.

```
signal s in
(emit s 1 || emit s 2); pause; emit s 3
|| await s([x]) in print_int x
```

L'implantation de REACTIVEML autorise un mécanisme un peu plus général permettant de tester la présence d'un motif. Ainsi, `await s(x::_)` permet d'attendre qu'*au moins* un signal ait été emis alors que `await s(x::y::_)` permet d'en attendre au moins deux. On peut ainsi modifier le programme précédent de sorte qu'au moins un signal s soit émis. Le comportement du programme est donc non déterministe et peut afficher 1 ou 2.

```
signal s in
(emit s 1 || emit s 2); pause; emit s 3
|| await s(x::_) in print_int x
```

L'attende d'*au moins* une valeur permet de ne pas nécessairement attendre la fin de l'instant pour pouvoir exécuter la partie droite du `await`, cela n'introduit pas de problème de causalité. Il existe donc en REACTIVEML la construction `await/immediate/one`.

```
signal s in
(emit s 1 || emit s 2); pause; emit s 3
|| await immediate one s(x) in print_int x
```

Ce programme affiche 1 ou 2 de façon non déterministe à l'instant où s est émis.

10. Conclusion

Dans cet article, nous avons présenté le langage REACTIVEML, une extension d'un langage fonctionnel avec des constructions réactives.

REACTIVEML est construit au dessus d'un langage fonctionnel strict (ici OCAML), permettant ainsi de disposer de toute la richesse des constructions de base (e.g., structures de données, structures de contrôle, ramasse miette), essentielle pour construire des applications conséquentes. L'ajout de constructions réactives fondées sur le modèle réactif synchrone permet de programmer des systèmes qui évoluent au cours du temps : les processus peuvent ainsi être créés ou détruits et les canaux de communication entre les processus peuvent évoluer dynamiquement.

De nombreuses applications ont été réalisées. Les deux plus ambitieuses concernent la simulation de protocoles de routage dans les réseaux ad hoc et les réseaux de capteurs (Mandel *et al.*, 2005a; Samper *et al.*, 2006).

Le compilateur REACTIVEML est accessible librement.[3] Il intègre les analyses statiques présentées ici et produit du code OCAML n'utilisant pas de threads. Ce code est ensuite lié à une bibliothèque implantant un ordonnanceur réactif. Notre implantation est aussi efficace que les autres implantations du modèle réactif tels que LOFT et LURC.

Ce langage est encore jeune et de nombreuses extensions peuvent être considérées. La première d'entre elles est l'identification de parties pouvant être ordonnées statiquement (autrement dit «compilées»). Le phénomène d'échappement de portée, qui n'existe pas dans les langages synchrones classiques, rend cependant cette compilation difficile. L'ajout d'une forme de threads de services à la manière des FAIR THREADS afin de traiter les entrées/sorties bloquantes reste à faire également.

11. Bibliographie

Acosta-Bermejo R., Rejo - Langage d'Objets Réactifs et d'Agents, Thèse de doctorat, Ecole des Mines de Paris, 2003.

Amadio R., Boudol G., Boussinot F., Castellani I., « Reactive concurrent programming revisited », *workshop Algebraic Process Calculi : the first twenty five years and beyond*, 2005.

Benveniste A., Caspi P., Edwards S., Halbwachs N., Guernic P. L., de Simone R., « The Synchronous Languages Twelve Years Later », *Proceedings of the IEEE, Special issue on embedded systems*, vol. 91, n° 1, p. 64-83, January, 2003.

Berry G., « Preemption in Concurrent Systems », *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, p. 72-93, 1993.

Berry G., « The Constructive Semantics of Esterel », , `www-sop.inria.fr/esterel.org`, 1998.

Berry G., Couronné P., Gonthier G., « Programmation synchrone des systèmes réactifs , le langage Esterel », *Technique et Science Informatique*, vol. 4, p. 305-316, 1987.

3. `http://moscova.inria.fr/~mandel/rml`

Boudol G., « ULM : A Core Programming Model for Global Computing », *Proceedings of the 13th European Symposium on Programming*, p. 234-248, 2004.

Boussinot F., « Reactive C : An Extension of C to Program Reactive Systems », *Software Practice and Experience*, vol. 21, n° 4, p. 401-428, April, 1991.

Boussinot F., « Concurrent Programming with Fair Threads : The LOFT Language », , `www-sop.inria.fr/meije/rp`, 2003.

Boussinot F., de Simone R., « The SL Synchronous Language », *Software Engineering*, vol. 22, n° 4, p. 256-266, 1996.

Boussinot F., Susini J.-F., « The SugarCubes Tool Box : A Reactive Java Framework », *Software Practice and Experience*, vol. 28, n° 4, p. 1531-1550, 1998.

Elliott C., Hudak P., « Functional Reactive Animation », *Proceedings of the international conference on Functional programming*, New York, NY, USA, p. 263-273, 1997.

Ferg S., « Event-Driven Programming : Introduction, Tutorial, History », , `http://eventdrivenpgm.sourceforge.net`, 2006.

Hazard L., Susini J.-F., Boussinot F., The Junior Reactive Kernel, RR n° 3732, INRIA, 1999.

Kahn G., « The Semantics of Simple Language for Parallel Programming », *Proceedings of IFIP 74 Conference*, p. 471-475, 1974.

Kleiman S., Shah D., Smaalders B., *Programming with threads*, SunSoft Press, 1996.

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

Mandel L., Benbadis F., « Simulation of Mobile Ad hoc Network Protocols in ReactiveML », *Proceedings of Synchronous Languages, Applications, and Programming*, Scotland, 2005a.

Mandel L., Pouzet M., « ReactiveML, a Reactive Extension to ML », *Proceedings of 7th International conference on Principles and Practice of Declarative Programming*, July, 2005b.

Milner R., « A Theory of Type Polymorphism in Programming. », *Journal of Computer and System Sciences*, vol. 17, n° 3, p. 348-375, 1978.

Milner R., *Communicating and Mobile Systems*, Cambridge University Press, 1999.

Ousterhout J. K., Why Threads Are A Bad Idea (for most purposes), Invited talk, USENIX Technical Conference, January, 1996. `http://home.pacbell.net/ouster/`.

Pierce B. C., *Types and Programming Languages*, MIT Press, 2002.

Pucella R., « Reactive Programming in Standard ML », *Proceedings of the IEEE International Conference on Computer Languages*, p. 48-57, 1998.

Samper L., Maraninchi F., Mounier L., Mandel L., « GLONEMO : Global and Accurate Formal Models for the Analysis of Ad-Hoc Sensor Networks », *Proceedings of the InterSense Conference*, Nice, France, May, 2006.

Serrano M., Boussinot F., Serpette B., « Scheme Fair Threads », *Proceedings of 6th International conference on Principles and Practice of Declarative Programming*, p. 203-214, 2004.

von Behren R., Condit J., Brewer E., « Why events are a bad idea (for high-concurrency servers) », *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, may, 2003.

Wan Z., Hudak P., « Functional Reactive Programming from first principles », *Proceedings of the conference on Programming language design and implementation*, p. 242-252, 2000.

Wright A. K., « Simple imperative polymorphism », *Lisp and Symbolic Computation*, vol. 8, n° 4, p. 343-355, 1995.