Réactivité des systèmes coopératifs : le cas de ReactiveML

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

1: LRI, CNRS UMR 8623, Université Paris-Sud 11
2: DI, CNRS UMR 8548, École normale supérieure
3: INRIA Paris-Rocquencourt
prenom.nom@ens.fr

Résumé

La concurrence coopérative est un modèle de programmation très répandu. On peut par exemple l'utiliser en OCaml à travers des bibliothèques comme `Lwt`, `Async` ou `Equeue`. Il a de nombreux avantages tels que l'absence de courses critiques et des implantations légères et efficaces. Néanmoins, un des inconvénients majeurs de ce modèle est qu'il dépend de la discipline du programmeur pour garantir que le système est réactif : un processus peut empêcher les autres de s'exécuter.

ReactiveML est un langage qui étend OCaml avec des constructions de concurrence coopérative. Il propose une analyse statique, l'analyse de réactivité, qui permet de détecter les expressions qui risquent de produire des comportements non coopératifs. Dans cet article, nous présentons cette analyse statique qui se définit à l'aide d'un système de types et effets. Ainsi, comme le typage de données aide les programmeurs à détecter des erreurs d'exécution au plus tôt, l'analyse de réactivité aide à détecter des erreurs de concurrence.

1. Introduction

Voici un exemple de courriel que nous avons reçu d'un utilisateur de ReactiveML [12] :

```
Hello,
Je me suis mis un peu au rml et j'essaie d'écrire mon premier programme :
une clock qui émet un signal toutes les secondes et un printer qui affiche
le message "top" quand le signal émis par clock est présent. [...]

Mais quand je lance le process main, il ne se passe rien [...]
Saurais-tu quel est le problème?

Cordialement,
-- Julien Blond
```

Nous allons étudier le programme qui était associé à ce message et voir qu'il s'agit d'un problème de *réactivité*, un problème classique des systèmes concurrents coopératifs. En effet, l'ordonnancement coopératif repose sur le principe que chaque processus va régulièrement suspendre son exécution pour laisser un autre processus s'exécuter. Contrairement à l'ordonnancement préemptif, ce n'est pas le système qui interrompt les processus, mais ce sont les processus eux-mêmes qui se suspendent. Cela rend possible une implémentation séquentielle efficace et sans les problèmes liés au parallélisme (atomicité des opérations, accès concurrent aux ressources partagées). La contrepartie de ce pouvoir donné au programmeur est qu'il est responsable de laisser les autres processus s'exécuter.

Dans les bibliothèques de programmation coopérative, cela se traduit par des bonnes pratiques de programmation. Par exemple, dans la documentation de `Lwt` [21], on peut lire : [1]

> Rules : `Lwt` will always try to execute as much as possible before yielding and switching to another cooperative thread. In order to make it work well, you must follow the following rules :
> – do not write function that may takes time to complete without using `Lwt`,
> – do not do IOs that may block, otherwise the whole program will hang. You must instead use asynchronous IOs operations.

De même, dans la documentation de `Async`, on trouve : [2]

> It's worth noting that it is possible to call a blocking function inside of an Async program, but you should avoid it. What will happen if you do that is that that single blocking operation will stop everything else in the Async world from happening. There are some things that are done in Async to make it harder to call an ordinary blocking function accidentally, but on some level, one just has to be careful.

La contribution de l'article est la définition d'une analyse statique qui garantit qu'un système coopératif est réactif. Nous ne nous intéressons pas ici aux cas des fonctions bloquantes par nature (comme les entrées/sorties) mais aux problèmes de réactivité liés au programme lui-même, comme le fait d'oublier de rendre la main à l'ordonnanceur. Pour pouvoir réaliser cette analyse de réactivité, la concurrence doit être exposée au niveau du langage. C'est le cas de ReactiveML, qui fait partie des langages de la famille ML auquel il ajoute des constructions de concurrence fondées sur le modèle *réactif synchrone* [6]. On obtient ainsi un langage riche (récursion, ordre supérieur, etc) muni d'un modèle de concurrence coopératif et déterministe. L'approche langage permet de définir simplement l'analyse, en évitant par exemple de l'encoder dans le système de types du langage hôte. En outre, l'avantage du modèle réactif synchrone est qu'il définit une notion de temps logique global vu comme une succession d'instants. Cela permet de simplifier la définition de la réactivité : un programme est réactif si son exécution fait progresser les instants logiques. Il n'est pas nécessaire de faire d'hypothèses sur l'équité de l'ordonnanceur ou sur les priorités des différents processus.

Nous allons commencer dans la partie 2 par présenter le langage ReactiveML et illustrer le problème de la réactivité sur l'exemple envoyé par l'utilisateur. Nous donnerons l'intuition et les limitations de l'analyse sur des exemples dans la partie 3. Puis, nous définirons formellement l'analyse à l'aide d'un système de types et effets dans les parties 4 et 5. Enfin, nous discuterons de l'implémentation de l'analyse et des travaux similaires dans la partie 6.

Le compilateur ReactiveML disponible à l'adresse `http://reactiveml.org` implémente l'analyse présentée dans l'article. Une version du toplevel de ReactiveML utilisable en ligne est disponible à l'adresse `http://reactiveml.org/jfla13`.

2. Premier programme

Le programme envoyé par notre utilisateur commençait par la définition suivante :

```
1 let process clock timer s =
2   let time = ref (Unix.gettimeofday ()) in
3   loop
4     let time' = Unix.gettimeofday () in
5     if time' -. !time >= timer
6     then (emit s (); time := time')
7   end
```

1. `http://ocsigen.org/lwt/manual/`
2. `https://bitbucket.org/yminsky/ocaml-core/wiki/DummiesGuideToAsync`

Le mot clé `process` introduit un processus, c'est-à-dire une fonction qui prend du temps : elle peut s'exécuter sur plusieurs instants logiques. Ici, l'utilisateur définit un processus nommé `clock` qui est paramétré par un flottant `timer` et un signal `s` (les signaux sont les canaux de communication entre processus). Il commence par déclarer une référence locale `time` initialisée par l'heure courante (ligne 2). Cette heure est obtenue par l'appel à la fonction `gettimeofday` du module `Unix` d'OCaml. Une fois la référence initialisée, le processus entre dans une boucle infinie (`loop`/`end`, lignes 3 à 7). À chaque itération de la boucle, une nouvelle mesure de l'heure courante est sauvegardée dans une variable `time'` (ligne 4). Puis, si le temps écoulé entre les deux mesures est plus grand que `timer` (ligne 5), alors le signal `s` est émis et la valeur de `time` est mise à jour avec la valeur de `time'` (ligne 6). Ce processus émet donc le signal `s` périodiquement dès que `timer` secondes se sont écoulées.

Si nous compilons ce processus, l'analyse de réactivité va afficher le message suivant :

```
Line 3, characters 2-120:
Warning: This expression may be an instantaneous loop.
```

En effet, le corps de cette boucle est instantané et donc les instants logiques ne vont pas progresser. Pour rendre ce processus coopératif, l'utilisateur doit explicitement indiquer à l'aide de l'expression `pause` (ligne 7) qu'il veut suspendre son exécution et attendre l'instant suivant :

```
1  let process clock timer s =
2    let time = ref (Unix.gettimeofday ()) in
3    loop
4      let time' = Unix.gettimeofday () in
5      if time' -. !time >= timer
6      then (emit s (); time := time');
7      pause
8    end
```

Maintenant, une nouvelle mesure du temps physique est prise à chaque instant logique.

Le programme de l'utilisateur continuait avec la définition suivante dont l'intention est d'afficher le message `top` à chaque émission du signal `s` :

```
10 let process print_clock s =
11   loop
12     do
13       print_string "top"; print_newline ()
14     when s done
15   end
```

La construction `do`/`when` exécute son corps uniquement aux instants où le signal `s` est présent (un signal est présent aux instants où il est émis) et termine en retournant la valeur calculée par son corps à l'instant où le corps termine. La communication se fait par diffusion instantanée : tous les processus qui testent la présence d'un signal au cours du même instant en ont une vison cohérente (il est soit présent, soit absent, mais il ne peux pas changer de statut pendant l'instant).

La compilation de ce processus `print_clock` produit le message suivant :

```
Line 11, characters 2-78:
Warning: This expression may be an instantaneous loop.
```

Une fois encore, cette boucle peut être instantanée et empêcher les instants logiques de progresser. Tant que le signal `s` est absent, le processus coopère, mais lorsque le signal est présent, l'exécution de `print_string "top"; print_newline()` est instantanée et termine. Donc le `do`/`when` termine ainsi que le corps de la boucle. Dans le même instant, une nouvelle instance de la boucle est exécutée

et le do/when teste à nouveau si le signal s est présent. Puisque l'on est encore dans le même instant, on affiche une seconde fois "top", puis on recommence la boucle et ainsi de suite. Pour corriger ce processus, on peut, comme pour le processus clock, simplement ajouter un appel à pause.

Une fois ces deux processus corrigés, le processus principal main définit bien :

```
[...] une clock qui émet un signal toutes les secondes et un printer qui affiche
le message "top" quand le signal émis par clock est présent.[...]
```

```
18 let process main =
19   signal s default () gather (fun () () -> ()) in
20   run (print_clock s) || run (clock 1. s)
```

La déclaration d'un signal local (ligne 19) prend en argument une valeur par défaut et une fonction de combinaison qui permet de déterminer la valeur du signal en cas d'émissions multiples sur ce signal au cours du même instant. Ici, la valeur par défaut du signal est () et il garde la valeur () en cas d'émissions multiples. L'opérateur || (ligne 20) indique la composition parallèle synchrone. Il exécute ses deux branches en parallèle en garantissant qu'à chaque instant les deux branches vont être exécutées. Ainsi, des instances des processus print_clock et clock s'exécutent en parallèle et communiquent à travers le signal local s (run marque l'instanciation de processus).

3. Intuition et limitations

La partie précédente nous a montré que les boucles infinies peuvent introduire des problèmes de réactivité. De façon plus générale, en ReactiveML, la récursion peut aussi être source de problèmes. Par exemple, le processus ci-dessous ne coopère jamais :

```
1 let rec process intantaneous s =
2   emit s ();
3   run (intantaneous s)
```

Le compilateur affiche le message suivant :

```
Line 3, characters 2-22:
Warning: This expression may produce an instantaneous recursion.
```

Une condition suffisante pour qu'un processus récursif soit réactif est qu'*il y ait toujours au moins un instant entre l'instanciation du processus et un appel récursif*. L'idée de l'analyse proposée est de vérifier statiquement cette condition.

On peut tout de suite remarquer que c'est une condition très forte : elle va rejeter des programmes corrects intéressants. Par exemple, elle va refuser l'exécution d'un map en parallèle (la construction let/and exécute ses deux branches en parallèle) :

```
let rec process par_map p l =
  match l with
  | [] -> []
  | x :: l -> let x' = run (p x)
              and l' = run (par_map p l) in
              x' :: l'
```

Ce processus fait des appels récursifs instantanés, mais il est coopératif car cette récursion termine (si la liste l est finie). Comme nous ne voulons pas avoir à prouver la terminaison des processus, nous avons donc choisi que notre analyse affiche des avertissements plutôt qu'elle rejette des programmes. Pour les mêmes raisons, nous faisons l'hypothèse que toutes les fonctions terminent de façon instantanée

et nous nous intéressons uniquement à la réactivité des processus. En particulier, nous ne gérons pas les fonctions qui font des entrées/sorties bloquantes et peuvent rendre un programme non réactif. Une solution à ce problème serait de proposer des implantations coopératives des fonctions d'entrées/sorties, comme le font les bibliothèques `Async` et `Lwt`.

L'analyse ne prend pas non plus en compte la valeur des signaux et sur-approxime les comportements possibles. Considérons l'exemple suivant :

```
let rec process imprecise =
  signal s default () gather (fun () () -> ()) in
  present s then () else (* pause implicite *) ();
  run imprecise
```

Le comportement de la construction `present/then/else` est d'exécuter instantanément la branche `then` si le signal est présent ou d'exécuter la branche `else` avec un instant de délai si le signal est absent. [3] Ici, comme le signal `s` est local et n'est pas émis, il est absent et la branche `else` sera toujours exécutée. Ce processus est donc coopératif mais l'analyse va quand même afficher un avertissement car si le signal `s` pouvait être présent, il y aurait une récursion instantanée.

Enfin, ce n'est pas parce que nous garantissons que le système est réactif qu'il est temps réel, c'est-à-dire qu'il réagit en temps et mémoire bornés, comme le montre le programme suivant :

```
let rec process server add =
  await add(p, ack) in
  run (server add) || let v = run p in emit ack v
```

Ce processus reçoit sur un signal `add` un processus `p` à exécuter et un signal `ack` sur lequel la valeur calculée doit être renvoyée. Comme `server` crée un nouveau processus à chaque fois que le signal `add` est émis, ce programme peut exécuter un nombre arbitraire de processus. Il n'est donc pas temps réel mais est bien réactif, puisque attendre la valeur d'un signal prend un instant (on doit attendre l'instant suivant pour combiner toutes les valeurs émises pendant l'instant).

4. Comportements réactifs

Dans cette partie, nous allons définir un langage très simple pour exprimer le comportement réactif des processus. On appellera *comportements* les termes de ce langage, que l'on notera κ dans la suite. Dans la partie suivante, nous définirons un système de types et effets qui permettra d'abstraire les processus en comportements. La réactivité des processus, c'est-à-dire la non-instantanéité des boucles et le fait qu'un instant au moins passe avant chaque appel récursif de processus, sera exprimée comme une condition de réactivité des comportements dans la partie 4.2.

4.1. Langage des comportements

Les termes du langage de *comportements* sont définis par :

$$\kappa ::= \bullet \mid 0 \mid \phi \mid \kappa \,||\, \kappa \mid \kappa + \kappa \mid \kappa ; \kappa \mid \mu\phi.\, \kappa \mid \texttt{run}\ \kappa$$

On doit tout d'abord distinguer les actions (sûrement) non instantanées, notées $\bullet$, dont l'exécution prend au moins un instant, comme `pause`, des actions potentiellement instantanées, notées 0, comme l'appel d'une fonction ou l'émission d'un signal. Puisque le langage est d'ordre supérieur, on aura aussi besoin de variables ϕ pour représenter les comportements inconnus des processus pris en argument.

3. Cela permet d'éviter des erreurs de causalité, c'est-à-dire d'incohérence sur la présence des signaux.

Les comportements doivent aussi traduire la structure du programme, en particulier la composition parallèle et les alternatives :

```
let process par_comb q1 q2 =
  loop
    run q1 || run q2
  end
```

```
let process if_comb c q1 q2 =
  loop
    if c then run q1 else run q2
  end
```

Dans le premier cas, la boucle est non instantanée si `q1` ou `q2` sont non instantanés, alors que dans le second cas il faut que les deux soit non instantanés. On va donc ajouter aux comportements la composition parallèle que l'on notera $||$ et un opérateur de choix non déterministe noté $+$. On choisit de complètement abstraire les valeurs et de simplement représenter les différents choix possibles.

Si l'on considère maintenant le cas des fonctions récursives, on doit aussi ajouter une notion de séquence dans les comportements :

```
let rec process good_rec =
  pause; run good_rec
```

```
let rec process bad_rec =
  run bad_rec; pause
```

L'ordre entre l'appel récursif et le `pause` est important puisque le premier processus récursif est correct alors que le second aboutit à une récursion instantanée. On notera donc ; l'opérateur de séquence. Si on note κ le comportement de `good_rec`, le comportement associé au corps du processus est $\bullet; \texttt{run}\ \kappa$. On verra dans la partie 5 que l'opérateur `run` permet de résoudre un problème technique dans le système de types. Comme le processus est défini récursivement, on doit avoir que κ est équivalent à $\bullet; \texttt{run}\ \kappa$ (on définira plus précisément la relation d'équivalence dans la partie 4.3). Pour gérer ce cas, on ajoute un opérateur de récursion explicite au langage des comportements et on obtient donc $\kappa = \mu\phi.\, \bullet; \texttt{run}\ \phi$.

Pour décrire le comportement d'une boucle, on introduit un opérateur noté κ^∞ qui correspond intuitivement à la répétition infinie du comportement κ. Il est défini comme une récursion par $\kappa^\infty \triangleq \mu\phi.\, \kappa; \texttt{run}\ \phi$.

4.2. Réactivité des comportements

Maintenant que l'on peut décrire le comportement des processus, on va pouvoir caractériser les comportements que l'on souhaite rejeter, c'est-à-dire les boucles et les récursions instantanées. Comme on l'a dit précédemment, un processus récursif est réactif s'il se passe au moins un instant avant chaque appel récursif (la réciproque est fausse comme le montre l'exemple de `par_map`). Dans le langage de comportements, cela se traduit en disant qu'un comportement récursif $\mu\phi.\, \kappa$ est réactif si la variable de récursion ϕ n'apparaît pas dans le premier instant de son corps κ. Puisque le comportement d'une boucle est $\kappa^\infty \triangleq \mu\phi.\, \kappa; \texttt{run}\ \phi$, on remarque que cette condition de réactivité des comportements récursifs implique aussi que le corps d'une boucle est non instantané.

On dit qu'un comportement est réactif si tous ses comportements récursifs sont réactifs. Plus formellement, on définit une relation $R \vdash \kappa$ qui signifie que κ est bien réactif dans l'ensemble de variables R, c'est-à-dire que ces variables n'apparaissent pas dans le premier instant de κ et que toutes les récursions de κ sont réactives :

$$\frac{}{R \vdash 0} \qquad \frac{}{R \vdash \bullet} \qquad \frac{\phi \notin R}{R \vdash \phi} \qquad \frac{R \vdash \kappa_1 \qquad \kappa_1 \downarrow^\bullet \qquad \emptyset \vdash \kappa_2}{R \vdash \kappa_1; \kappa_2} \qquad \frac{R \vdash \kappa_1 \qquad not(\kappa_1 \downarrow^\bullet) \qquad R \vdash \kappa_2}{R \vdash \kappa_1; \kappa_2}$$

$$\frac{R \vdash \kappa_1 \qquad R \vdash \kappa_2}{R \vdash \kappa_1 \,||\, \kappa_2} \qquad \frac{R \vdash \kappa_1 \qquad R \vdash \kappa_2}{R \vdash \kappa_1 + \kappa_2} \qquad \frac{R \cup \{\phi\} \vdash \kappa}{R \vdash \mu\phi.\, \kappa} \qquad \frac{R \vdash \kappa}{R \vdash \texttt{run}\ \kappa}$$

On voit que dans le cas de la séquence $\kappa_1;\kappa_2$, si le comportement κ_1 est non instantané, ce que l'on note $\kappa_1 \downarrow^\bullet$, alors on n'a plus besoin de vérifier la présence des variables de R dans κ_2. Il faut tout de même vérifier que les récursions de κ_2 sont réactives, c'est-à-dire que $\emptyset \vdash \kappa_2$. Le prédicat de non-instantanéité est défini, uniquement pour les comportements réactifs, par :

$$\frac{}{\bullet \downarrow^\bullet} \qquad \frac{}{\phi \downarrow^\bullet} \qquad \frac{\kappa_1 \downarrow^\bullet}{\kappa_1;\kappa_2 \downarrow^\bullet} \qquad \frac{\kappa_2 \downarrow^\bullet}{\kappa_1;\kappa_2 \downarrow^\bullet} \qquad \frac{\kappa_1 \downarrow^\bullet}{\kappa_1 \,||\, \kappa_2 \downarrow^\bullet} \qquad \frac{\kappa_2 \downarrow^\bullet}{\kappa_1 \,||\, \kappa_2 \downarrow^\bullet}$$

$$\frac{\kappa_1 \downarrow^\bullet \qquad \kappa_2 \downarrow^\bullet}{\kappa_1 + \kappa_2 \downarrow^\bullet} \qquad \frac{\kappa \downarrow^\bullet}{\mu\phi.\,\kappa \downarrow^\bullet} \qquad \frac{\kappa \downarrow^\bullet}{\texttt{run}\ \kappa \downarrow^\bullet}$$

Un comportement récursif est non instantané si son corps n'est pas instantané. Le fait que les variables soient toujours non instantanées renvoie la vérification au moment de l'appel d'un processus plutôt qu'à sa définition. Au moment de la définition d'un processus, on suppose que ces arguments sont non instantanés. Si jamais ce n'est pas le cas, alors on pourra observer la non-réactivité du comportement une fois la variable instanciée.

4.3. Équivalence de comportements

On peut définir une relation d'équivalence sur les comportements qui permet intuitivement de les simplifier. Elle est définie de telle sorte que si $\kappa_1 \equiv \kappa_2$ et $R \vdash \kappa_1$ alors $R \vdash \kappa_2$.

La relation $\equiv$ est une relation d'équivalence, c'est-à-dire une relation réflexive, symétrique et transitive. Les opérateurs ; et $||$ et $+$ sont compatibles avec cette relation d'équivalence et idempotents. ; et $||$ et $+$ sont associatifs. $||$ et $+$ sont commutatifs (mais pas ;). Le comportement $\texttt{0}$ (resp. $\bullet$) est l'élément neutre de ; et $||$ (resp. $+$). On ajoute aussi les règles suivantes, qui expriment les propriétés de l'opérateur de récursion et traduisent l'intuition de récursion :

$$\frac{\kappa_1 \equiv \kappa_2}{\mu\phi.\,\kappa_1 \equiv \mu\phi.\,\kappa_2} \qquad \mu\phi.\,\kappa \equiv \kappa[\phi \leftarrow \mu\phi.\,\kappa] \qquad \bullet^\infty \equiv \bullet$$

On peut par exemple montrer que le comportement $\mu\phi.\,((\bullet \,||\, \texttt{0});(\texttt{run}\ \phi + \texttt{run}\ \phi))$ est équivalent à $\mu\phi.\,\bullet;\texttt{run}\ \phi$.

5. Analyse de réactivité

L'analyse de réactivité se fait en deux temps : un système de *types et effets* calcule les comportements associés aux processus du programme, puis on vérifie leur réactivité en suivant les règles de la partie 4.2. Nous allons maintenant formaliser ce système de types sur un noyau de ReactiveML.

5.1. Syntaxe

La syntaxe abstraite du noyau est la suivante :

$$\begin{aligned} e ::= {} & x \mid c \mid (e,e) \mid \lambda x.e \mid e\ e \mid \texttt{rec}\ x = e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{process}\ e \mid \texttt{run}\ e \mid \texttt{pause} \\ & \mid \texttt{let}\ x = e\ \texttt{and}\ x = e\ \texttt{in}\ e \mid \texttt{signal}\ x\ \texttt{default}\ e\ \texttt{gather}\ e\ \texttt{in}\ e \mid \texttt{emit}\ e\ e \\ & \mid \texttt{present}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{loop}\ e \mid \texttt{do}\ e\ \texttt{until}\ e(x) \rightarrow e \mid \texttt{do}\ e\ \texttt{when}\ e \end{aligned}$$

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`) ainsi que plusieurs structures de contrôles : le test de présence d'un signal (`present`), la boucle inconditionnelle (`loop`), la préemption

faible (`until`) et la suspension (`when`). L'expression `do` e_1 `until` $s(x) \to e_2$ exécute le corps e_1 et, en cas de présence du signal s, interrompt l'exécution de e_1 puis exécute à l'instant suivant la continuation e_2 en liant x à la valeur de s. On notera _ les variables qui n'apparaissent pas libres dans le corps du `let` et () l'unique valeur de type `unit`. A partir de ce noyau, on peut encoder la plupart des autres constructions du langage, par exemple :

$$e_1 \,||\, e_2 \triangleq \texttt{let } \_ = e_1 \texttt{ and } \_ = e_2 \texttt{ in } ()$$
$$\texttt{let } x = e_1 \texttt{ in } e_2 \triangleq \texttt{let } x = e_1 \texttt{ and } \_ = () \texttt{ in } e_2$$
$$\texttt{let } f\, x_1 \ldots x_p = e_1 \texttt{ in } e_2 \triangleq \texttt{let } f = \lambda x_1 \ldots . \lambda x_p . e_1 \texttt{ in } e_2$$
$$\texttt{let rec } f\, x_1 \ldots x_p = e_1 \texttt{ in } e_2 \triangleq \texttt{let } f = (\texttt{rec } f = \lambda x_1 \ldots . \lambda x_p . e_1) \texttt{ in } e_2$$
$$\texttt{let process } f\, x_1 \ldots x_p = e_1 \texttt{ in } e_2 \triangleq \texttt{let } f = \lambda x_1 \ldots . \lambda x_p . \texttt{process } e_1 \texttt{ in } e_2$$
$$\texttt{let rec process } f\, x_1 \ldots x_p = e_1 \texttt{ in } e_2 \triangleq \texttt{let } f = (\texttt{rec } f = \lambda x_1 \ldots . \lambda x_p . \texttt{process } e_1) \texttt{ in } e_2$$
$$e_1 ; e_2 \triangleq \texttt{let } \_ = e_1 \texttt{ in } e_2$$
$$\texttt{await } e_1(x) \texttt{ in } e_2 \triangleq \texttt{do } (\texttt{loop pause}) \texttt{ until } e_1(x) \to e_2$$

Les types sont définis de façon classique par :

$$\tau ::= \alpha \mid T \mid \tau \times \tau \mid \tau \to \tau \mid \tau\, \texttt{process}[\kappa] \mid (\tau, \tau)\, \texttt{event} \qquad \text{(types)}$$
$$\sigma ::= \tau \mid \forall \phi.\, \sigma \mid \forall \alpha.\, \sigma \qquad \text{(schémas de types)}$$
$$\Gamma ::= \emptyset \mid \Gamma, x : \sigma \qquad \text{(environnements)}$$

Un type peut être une variable de type α, un type de base T (comme `unit` ou `bool`), un produit, une fonction, un processus ou un signal. Le type d'un processus est paramétré par son type de retour et son comportement. Le type d'un signal est paramétré par le type des valeurs émises et le type des valeurs lues.

Les schémas de type sont des types où l'on peut quantifier universellement des variables de type α et des variables de comportement ϕ. On note $ftv(\tau)$ (resp. $fbv(\tau)$) l'ensemble des variables de type (resp. de comportement) libres dans τ et $fv(\tau) = ftv(\tau), fbv(\tau)$. Les fonctions de généralisation et d'instanciation sont définies par :

$$\sigma[\alpha \leftarrow \tau] \leq \forall \alpha.\, \sigma \qquad\qquad \sigma[\phi \leftarrow \kappa] \leq \forall \phi.\, \sigma$$
$$gen(\tau, e, \Gamma) = \tau \qquad \text{si } e \text{ est expansive}$$
$$gen(\tau, e, \Gamma) = \forall \bar{\alpha}. \forall \bar{\phi}.\, \tau \text{ où } \bar{\alpha}, \bar{\phi} = fv(\tau) \setminus fv(\Gamma) \qquad \text{sinon}$$

Comme pour les références en ML, il faut veiller à ne pas généraliser les expressions qui allouent des signaux. On utilise pour cela le critère syntaxique d'expressions expansives et non-expansives [20].

5.2. Règles de typage

Les jugements de typage sont de la forme « $\Gamma \vdash e : \tau \mid \kappa$ » qui signifie que dans l'environnement de types Γ, l'expression e a le type τ et le comportement κ. L'environnement de typage initial Γ_0 contient les définitions des différentes primitives :

$$\Gamma_0 \triangleq [\texttt{emit} : \forall \alpha_1, \alpha_2.\, (\alpha_1, \alpha_2)\, \texttt{event} \to \alpha_1 \to \texttt{unit}; \texttt{true} : \texttt{bool}; \texttt{fst} : \forall \alpha_1, \alpha_2.\, \alpha_1 \times \alpha_2 \to \alpha_1; \ldots]$$

Les règles de typage sont données dans la figure 1. Si l'on efface tout ce qui concerne les comportements, les règles sont identiques à celles du système de types de ReactiveML présenté dans [12], qui est lui-même une extension du système de types de ML.

En ce qui concerne les comportements, on peut faire plusieurs remarques et commentaires :

– La règle Equiv montre l'utilité de la relation d'équivalence sur les comportements : elle permet de simplifier le comportement obtenu. On peut à tout moment dans les règles de typage remplacer un comportement par un comportement équivalent.

EQUIV
$$\frac{\Gamma \vdash e : \tau \mid \kappa_1 \qquad \kappa_1 \equiv \kappa_2}{\Gamma \vdash e : \tau \mid \kappa_2} \qquad \frac{\tau \leq \Gamma(x)}{\Gamma \vdash x : \tau \mid 0} \qquad \frac{\tau \leq \Gamma_0(c)}{\Gamma \vdash c : \tau \mid 0} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \mid 0 \qquad \Gamma \vdash e_1 : \tau_2 \mid 0}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 \mid 0}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \mid 0}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2 \mid 0} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \mid 0 \qquad \Gamma \vdash e_2 : \tau_2 \mid 0}{\Gamma \vdash e_1\ e_2 : \tau_1 \mid 0} \qquad \frac{\Gamma, x : \tau \vdash e : \tau \mid 0}{\Gamma \vdash \texttt{rec}\ x = e : \tau \mid 0}$$

$$\frac{\Gamma \vdash e : \tau \mid \kappa}{\Gamma \vdash \texttt{process}\ e : \tau\, \texttt{process}[\kappa] \mid 0} \qquad \frac{\Gamma \vdash e : \tau\, \texttt{process}[\kappa] \mid 0}{\Gamma \vdash \texttt{run}\ e : \tau \mid \texttt{run}\ \kappa} \qquad \Gamma \vdash \texttt{pause} : \texttt{unit} \mid \bullet$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \mid 0 \qquad \Gamma \vdash e_2 : \tau \mid \kappa_2 \qquad \Gamma \vdash e_3 : \tau \mid \kappa_3}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau \mid \kappa_2 + \kappa_3}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid \kappa_1 \qquad \Gamma \vdash e_2 : \tau_2 \mid \kappa_2 \qquad \Gamma, x_1 : gen(\tau_1, e_1, \Gamma), x_2 : gen(\tau_2, e_2, \Gamma) \vdash e : \tau \mid \kappa}{\Gamma \vdash \texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e : \tau \mid (\kappa_1 \parallel \kappa_2); \kappa}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \mid 0 \qquad \Gamma \vdash e_2 : \tau_1 \to \tau_2 \to \tau_2 \mid 0 \qquad \Gamma, x : (\tau_1, \tau_2)\, \texttt{event} \vdash e : \tau \mid \kappa}{\Gamma \vdash \texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e : \tau \mid \kappa} \qquad \frac{\Gamma \vdash e_1 : (\tau_1, \tau_2)\, \texttt{event} \mid 0 \qquad \Gamma \vdash e_2 : \tau \mid \kappa_2 \qquad \Gamma \vdash e_3 : \tau \mid \kappa_3}{\Gamma \vdash \texttt{present}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau \mid \kappa_2 + (\bullet; \kappa_3)}$$

$$\frac{\Gamma \vdash e : \tau \mid \kappa}{\Gamma \vdash \texttt{loop}\ e : \texttt{unit} \mid \kappa^\infty} \qquad \frac{\Gamma \vdash e_1 : \tau \mid \kappa_1 \qquad \Gamma \vdash e_2 : (\tau_1, \tau_2)\, \texttt{event} \mid 0 \qquad \Gamma, x : \tau_2 \vdash e_3 : \tau \mid \kappa_3}{\Gamma \vdash \texttt{do}\ e_1\ \texttt{until}\ e_2(x) \to e_3 : \tau \mid \kappa_1 + \bullet; \kappa_3} \qquad \frac{\Gamma \vdash e_1 : \tau \mid \kappa \qquad \Gamma \vdash e_2 : (\tau_1, \tau_2)\, \texttt{event} \mid 0}{\Gamma \vdash \texttt{do}\ e_1\ \texttt{when}\ e_2 : \tau \mid \kappa}$$

FIGURE 1 – Le système de types

- Le langage ReactiveML impose une séparation nette entre les expressions nécessairement instantanées (c-à-d. les expressions ML pures) et les processus. Ainsi, on ne peut par exemple pas appeler `pause` ou `await` dans le corps d'une fonction qui doit être instantané. Une analyse statique séparée et préalable au typage vérifie cette propriété de bonne formation des expressions, notée $k \vdash e$ dans [12]. On remarque que toutes les expressions (nécessairement) instantanées (c'est-à-dire telles que $0 \vdash e$) vérifient $\Gamma \vdash e : \tau \mid 0$. C'est pourquoi de nombreuses règles imposent un comportement égal à 0, sans que cela n'apporte de contraintes supplémentaires par rapport à l'analyse de bonne formation des expressions. La réciproque n'est pas vraie puisque le comportement 0 est associé aux processus et opérateurs *potentiellement* instantanés alors que $0 \vdash e$ signifie que e est *nécessairement* instantanée.
- On ne cherche pas à prouver la terminaison des fonctions OCaml pures (sans comportement réactif) : la règle de l'application montre que l'on suppose que les appels de fonctions terminent toujours. C'est pourquoi il n'y a pas de comportement associé à une fonction, puisqu'il est toujours instantané (il n'y a pas de comportement sur les flèches contrairement à la tradition dans les systèmes de types et effets).
- Dans le modèle synchrone réactif, on ne peut pas réagir instantanément à l'absence d'un signal. C'est pourquoi dans le cas du `present` la branche `then` est exécutée immédiatement si le signal est présent et la branche `else` à l'instant suivant dans le cas contraire. Cela se traduit de façon immédiate dans le comportement de `present`. De même, dans un `until`, en cas de préemption, la continuation est exécutée à l'instant suivant la réception du signal. Cela est reflété dans la règle de typage.

– On peut vérifier que l'encodage des primitives que l'on a donné permet bien de retrouver les comportements attendus. Par exemple, on peut considérer le cas de $e_1; e_2$:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash e_1 : \tau_1 \mid \kappa_1 \qquad \Gamma \vdash () : \texttt{unit} \mid 0 \qquad \Gamma \vdash e_2 : \tau_2 \mid \kappa_2}{\Gamma \vdash \texttt{let \_} = e_1 \texttt{ and \_} = () \texttt{ in } e_2 : \tau_2 \mid (\kappa_1 \mid\mid 0); \kappa_2} \qquad \dfrac{\kappa_1 \mid\mid 0 \equiv \kappa_1 \qquad \kappa_2 \equiv \kappa_2}{(\kappa_1 \mid\mid 0); \kappa_2 \equiv \kappa_1; \kappa_2}}{\Gamma \vdash e_1; e_2 : \tau \mid (\kappa_1 \mid\mid 0); \kappa_2}}{\Gamma \vdash e_1; e_2 : \tau_2 \mid \kappa_1; \kappa_2}$$

On peut de la même façon vérifier que $e_1 \,||\, e_2$ a bien un comportement équivalent à $\kappa_1 \mid\mid \kappa_2$ ou que $\texttt{await } e_1(x) \texttt{ in } e_2$ a le comportement $\bullet^\infty + (\bullet; \kappa_2) \equiv \bullet; \kappa_2$ après simplification.

– Il n'y a pas de règles particulières pour les processus récursifs. C'est l'ajout de l'opérateur `run` qui force le comportement des processus récursifs à être de la forme $\mu\phi.\,\kappa$ avec $\phi \in fbv(\kappa)$. Considérons par exemple le processus défini par `let rec process p = run p`, que l'on écrit $\texttt{rec } p = \texttt{process } (\texttt{run } p)$ dans notre noyau :

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, p : \beta\,\texttt{process}[\kappa] \vdash p : \beta\,\texttt{process}[\kappa] \mid 0}{\Gamma, p : \beta\,\texttt{process}[\kappa] \vdash \texttt{run } p : \beta \mid \texttt{run } \kappa} \qquad \texttt{run } \kappa \equiv \kappa}{\Gamma, p : \beta\,\texttt{process}[\kappa] \vdash \texttt{run } p : \beta \mid \kappa}\ \textsc{Equiv}}{\Gamma, p : \beta\,\texttt{process}[\kappa] \vdash \texttt{process } (\texttt{run } p) : \beta\,\texttt{process}[\kappa] \mid 0}}{\Gamma \vdash \texttt{rec } p = \texttt{process } (\texttt{run } p) : \beta\,\texttt{process}[\kappa] \mid 0}$$

On voit sur la dérivation de typage que l'on doit vérifier $\texttt{run } \kappa \equiv \kappa$ pour pouvoir typer cette expression. Seul le comportement $\mu\phi.\,\texttt{run } \phi$ vérifie cette propriété. Cela explique aussi pourquoi aucune règle d'équivalence ne concerne l'opérateur `run`, qui ne doit pas être simplifiable. En effet, sans le `run`, la condition $\kappa \equiv \kappa$ serait toujours vraie. On aurait donc pu prendre par exemple $\kappa = 0$ et rater la récursion instantanée.

– La construction `loop` ne fait pas habituellement partie du noyau du langage puisque l'on peut l'obtenir par :

$$\texttt{loop } e \triangleq \texttt{ run } ((\texttt{rec } \mathit{loop} = \lambda x.\texttt{process } (\texttt{run } x; \texttt{run } (\mathit{loop}\ x))) \ (\texttt{process } e))$$

Si l'on applique les règles du système de types et en supposant que $\Gamma \vdash e : \tau \mid \kappa$, on obtient les types suivants :

$$\mathit{loop} : \forall\phi.\,\alpha\,\texttt{process}[\phi] \to \alpha'\,\texttt{process}[\mu\phi'.\,\texttt{run } \phi; \texttt{run } \phi']$$
$$\texttt{process } e : \tau\,\texttt{process}[\kappa]$$

On obtient donc le comportement $\texttt{run } (\mu\phi'.\,\texttt{run } \kappa; \texttt{run } \phi')$ pour `loop` e. Ce comportement n'est pas équivalent à κ^∞, mais il est réactif si et seulement si κ^∞ est réactif, puisque les `run` n'influent pas sur la réactivité des comportements. On aurait donc pu enlever `loop` du noyau présenté ici sans aucune incidence sur le résultat de l'analyse de réactivité.

– Ce n'est par contre pas le cas pour `pause`, que l'on peut encoder par :

$$\texttt{pause} \triangleq \texttt{signal } s \texttt{ default } () \texttt{ gather } (\lambda x.\lambda y.()) \texttt{ in present } s \texttt{ then } () \texttt{ else } ()$$

Nous avons fait le choix de complètement abstraire les valeurs et les présences des signaux. Comme dans l'exemple `imprecise` de la partie 3, on ne prend pas en compte le fait que le signal s n'est jamais présent et que seule la seconde branche du `present` est exécutée. On obtient donc le comportement $0 + \bullet; 0 \equiv 0$ qui n'est pas du tout ce que l'on souhaite.

5.3. Exemples

Dans le cas de l'exemple `timer` présenté dans la partie 2, le comportement de la boucle (lignes 3 à 7) est $((0 \parallel 0); (0 + (0; 0)))^\infty$ (équivalent à 0^∞), qui n'est pas réactif. Une fois le processus corrigé, on obtient le comportement $((0 \parallel 0); (0 + (0; 0)); \bullet)^\infty$ (équivalent à $\bullet^\infty$) qui est lui réactif. De même, le comportement associé au processus `instantaneous` défini au début de la partie 3 est $\mu\phi.\, 0; \texttt{run}\ \phi$, dont on vérifie simplement qu'il n'est pas réactif.

Le fait de fonder l'analyse sur les types plutôt que sur les noms des variables permet de gérer simplement les cas d'alias comme dans les exemples suivants :

```
let rec process p =
  let q = p in
  run q
```

```
let rec process p =
  let q = (fun x -> x) p in
  run q
```

```
val p : 'a process[rec 'r. run 'r]
Warning: This expression may produce an instantaneous recursion.
```

Dans les deux cas, `q` a le même type que `p` et donc on obtient bien le comportement $\mu\phi.\, \texttt{run}\ \phi$ pour `p` qui n'est pas réactif.

Dans le cas d'un processus prenant en argument un autre processus, aucun avertissement ne sera affiché au moment de la définition puisque l'on ne peut pas encore décider de la réactivité :

```
let process par_comb q1 q2 =
  loop
    run q1 || run q2
  end
val par_comb : 'a process['r1] -> 'b process ['r2] ->
                 unit process[rec 'r3. run 'r1  ||  run 'r2 ;  'r3]
```

Cela se traduit formellement par le fait que les variables de comportement sont réactives et non instantanées. C'est au moment de l'instanciation que l'on pourra réellement tester la réactivité :

```
let process p1 = run (par_comb (process ()) (process (pause)))
val p1 : unit process[rec 'r. run 0  ||  run * ;  'r]

let process p2 = run (par_comb (process ()) (process ()))
val p2 : unit process[rec 'r. run 0  ||  run 0 ;  'r]
Warning: This expression may produce an instantaneous recursion.
```

Notre analyse peut également rejeter des programmes utilisant réellement l'ordre supérieur, avec des fonctions manipulant des processus :

```
let higher_order f =
 let rec process p =
   let q = f p in
   run q
 in p
val higher_order : ('a process[run 'r1] -> 'a process['r1]) -> 'a process[run 'r1]

let process good = run (higher_order (fun x -> process (pause; run x)))
val good : 'a process[run (run (rec 'r1. * ; run (run 'r1)))]

let process pb = run (higher_order (fun x -> process (run x)))
val pb : 'a process[run (run (rec 'r2. run (run 'r2)))]
Warning: This expression may produce an instantaneous recursion.
```

Un autre cas intéressant rejeté par notre analyse est celui où l'on programme un opérateur `fix` de récursion. Il prend en entrée une fonction attendant une continuation, puis l'applique en donnant la fonction elle-même comme continuation. On peut utiliser cet opérateur pour créer un processus récursif, dont notre analyse vérifiera bien la réactivité :

```
let rec fix f x = f (fix f) x
val fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b

let process main =
  let process p k v =
    print_int v; print_newline ();
    run (k (v+1))
  in
  run (fix p 0)
val main : 'a process[run rec 'r0. run 'r0]
Warning: This expression may produce an instantaneous recursion.
```

5.4. Analyse de réactivité avec rangées

Le système de types que l'on vient de présenter peut être incapable de donner un type à un programme correct du point de vue du systèmes de types (classique) de ReactiveML. C'est par exemple le cas du programme suivant :

```
let process p = pause
let process q = ()
let l = [p; q]
```

Le processus `p` a le type $\texttt{unit process}[\bullet]$ alors que `q` a le type $\texttt{unit process}[0]$. On ne sait pas unifier ces deux types avec les règles présentées précédemment. On peut résoudre ce soucis avec une extension simple du systèmes de types, inspirée des types *rangées* [16], en ne changeant que la règle de typage des processus :

$$\frac{\Gamma \vdash e : \tau \mid \kappa}{\Gamma \vdash \texttt{process}\ e : \tau\ \texttt{process}[\kappa + \kappa'] \mid 0}$$

Intuitivement, on ne dit plus qu'un processus a (exactement) le comportement de son corps mais qu'il a au moins le comportement de son corps. Au moment de l'unification de deux processus, on obtiendra un processus dont le comportement est plus grand (au sens de $+$) que ceux des deux processus. Dans l'exemple ci-dessus, on aura désormais :

$p : \forall \phi_p.\, \texttt{unit process}[\bullet + \phi_p] \qquad q : \forall \phi_q.\, \texttt{unit process}[0 + \phi_q] \qquad l : \forall \phi.\, \texttt{unit process}[0 + \bullet + \phi]\ \texttt{list}$

Avec cette extension, le comportement inféré d'un processus est toujours soit une variable (par ex. pour les processus en argument) soit une rangée, c'est-à-dire de la forme $\kappa + \phi$. On peut donc toujours unifier le comportement de deux processus. Cela permet de donner un type et un comportement à tout programme ReactiveML correct (du point de vue du système de types classique de ReactiveML).

Un des inconvénients de cette modification du système de types est que les comportements inférés comportent un grand nombre de variables, souvent inutiles. Par exemple, on obtient des comportements complexes dans l'exemple suivant :

```
let process p = pause
val p : unit process[* + 'r0]
let process q = run p
val q : unit process[run (* + 'r0) + 'r1]
let process r = run q
val r : unit process[run (run (* + 'r0) + 'r1) + 'r2]
```

Une solution partielle à ce problème est d'utiliser ce qu'on appelle le *masquage d'effets* [14]. On introduit pour cela la règle de typage suivante :

$$\frac{\Gamma \vdash e : \tau \mid \kappa \qquad \phi \notin ftv(\Gamma, \tau)}{\Gamma \vdash e : \tau \mid \kappa[\phi \leftarrow \bullet]}$$

Elle signifie intuitivement que si la variable de comportement ϕ n'apparait pas libre dans l'environnement, alors elle n'est pas contrainte. On peut en particulier choisir le comportement $\bullet$ qui est l'élément neutre pour $+$. Cela permet de simplifier les types et on obtient alors des comportements plus raisonnables dans notre exemple :

```
let process p = pause
val p : unit process[* + 'r0]
let process q = run p
val q : unit process[run * + 'r0]
let process r = run q
val r : unit process[run (run *) + 'r0]
```

6. Discussion

6.1. Implémentation

Le système de types présenté dans la partie 5 est implémenté dans le compilateur de ReactiveML avec l'extension présentée dans la partie 5.4. Le typeur de ReactiveML a été étendu pour calculer les comportements, sans que sa structure ni sa complexité ne changent. Il utilise des techniques classiques pour gérer les comportements récursifs [15] et les types rangées [16]. Il faut ensuite vérifier la réactivité des comportements en suivant les règles énoncées dans la partie 4.2, que l'on peut très simplement traduire en un algorithme de complexité linéaire en la taille des comportements. L'implémentation ne prend en compte qu'une partie des règles d'équivalence, ce qui n'a aucune incidence sur la correction, mais simplement sur la taille des comportements calculés. En pratique, l'analyse a un faible impact sur le temps de compilation.

La principale difficulté de l'analyse est de générer des messages d'avertissement utiles, en particulier en ce qui concerne la localisation des erreurs. En effet, un comportement non-réactif peut apparaître à plusieurs endroits dans l'arbre de syntaxe, mais il convient de n'afficher qu'un seul message, au plus près de la source du problème de réactivité. Considérons par exemple le programme suivant :

```
let rec process false_warning n =
  if n < 0 then ()
  else run (false_warning (n-1))

let process main = run (false_warning 10) || run (false_warning 5)
```

Notre analyse, qui ne prend pas en compte les valeurs, va afficher un avertissement au moment de la définition de `false_warning`. Mais son comportement non-réactif apparaît aussi à chacune de ses utilisations. Pour ne pas afficher d'erreurs à chaque fois que l'on utilise un processus dont le comportement est non-réactif, on marque le comportement récursif fautif pour se rappeler que l'on a déjà signalé cette erreur. On ne peut par contre pas marquer les comportements récursifs corrects au cours du typage puisqu'ils peuvent devenir incorrects par la suite par unification, comme dans l'exemple suivant :

```
let process unify =
  signal s default (process (pause)) gather (fun x _ -> x) in
  (await s(p) in loop run p end) || emit s (process ())
```

Lorsque l'on a fini de typer la boucle, le type de `s` est $(\alpha\,\texttt{process}[\bullet + \phi], \alpha\,\texttt{process}[\bullet + \phi])\,\texttt{event}$. La boucle semble donc être non instantanée. Pourtant, on émet par la suite un processus de type $\alpha\,\texttt{process}[0 + \phi']$ sur `s`, donc son type devient $(\alpha\,\texttt{process}[0 + \bullet + \phi''], \alpha\,\texttt{process}[0 + \bullet + \phi''])\,\texttt{event}$, ce qui rend la boucle potentiellement instantanée.

6.2. Cas des références

Le noyau présenté au début de la partie 5 omet volontairement les primitives liées aux références. En effet, on peut les utiliser pour encoder la récursion, comme dans l'exemple suivant qui renvoie un processus dont l'exécution va provoquer une boucle instantanée :

```
let landin () =
  let f = ref (process ()) in
  f := process (run !f);
  !f
val landin : unit -> unit process[0 + (rec 'r1. run (0 + 'r1)) + 'r2]
Warning: This expression may produce an instantaneous recursion.
```

La récursion dans les comportements est introduite par l'unification et non par la présence d'un `rec` dans le source. L'analyse découvre donc cette récursion cachée.

6.3. Travaux similaires

L'analyse de réactivité est une analyse classique dans les langages synchrones. En Esterel [3], le langage qui est à la source des travaux sur le modèle réactif synchrone [6], l'analyse de réactivité est très simple car le langage est du premier ordre et sans récursion.

Parmi les langages fondés sur le modèle réactif synchrone, FunLoft [7] et ULM [4] proposent des analyses de réactivité. L'analyse de FunLoft se base sur les travaux de [1]. Elle prouve non seulement que les instants terminent mais elle donne aussi une borne sur la durée des instants par une analyse des valeurs. Cette analyse se limite au cas d'un langage du premier ordre. En ULM, une pause est introduite implicitement à chaque appel récursif. Ainsi, par construction, il ne peut pas y avoir de récursion instantanée. La contrepartie de ce choix est une perte d'expressivité. Par exemple, dans un processus comme `server` (partie 3), un message pourrait être perdu entre la réception du message `add` et la nouvelle mise en attente.

Dans les langages synchrones flot de données comme Lustre [8] et Lucid Synchrone [9], la réactivité du système est liée à la causalité et donc de façon plus générale aux conditions de garde des définitions récursives. Comme pour l'analyse de causalité de Lucid Synchrone [10], nous avons utilisé des types rangées pour traiter le sous-typage en présence d'ordre supérieur.

Le système de types présenté ici est fortement inspiré des *behaviours* de [2]. Les auteurs présentent leur système de types sur le langage ConcurrentML [17], qui est une extension de ML avec des primitives d'envoi de messages. Les comportements des processus sont plus riches puisqu'ils contiennent les réceptions et envois sur chaque canal de communication. Il s'agit en quelque sorte d'une méta-analyse : on peut utiliser le résultat pour prouver de nombreuses propriétés, selon le traitement que l'on fait sur les comportements obtenus. Les auteurs l'utilisent par exemple pour démontrer que les émissions sur un canal précèdent toujours celles sur un autre canal sur un exemple particulier. Leur système de type est plus complexe puisque les types peuvent apparaître dans les comportements, ce qui engendre plusieurs difficultés techniques.

Le système de types présenté ici est un système de types et effets [11, 14], dont le principe est d'associer à chaque expression un type, associé à la valeur finale après réduction de l'expression, mais aussi un effet représentant l'impact de la réduction de cette expression. Ils ont été particulièrement

utilisés pour la gestion de la mémoire par régions [19]. Alors que les effets associés à une expression sont en général des ensembles (de régions par exemple), on peut aussi garder la structure (ou une abstraction de la structure) du programme dans les effets [2, 18]. [5] utilise un système de types et effets pour prouver la terminaison de programmes fonctionnels en présence de références, en stratifiant la mémoire pour éviter les encodages de la récursion par les références.

7. Conclusion

Nous avons présenté une analyse de réactivité sur le langage synchrone fonctionnel ReactiveML. Il s'agit d'un système de types et effets dans lequel les effets, aussi appelés comportements, rendent compte du comportement réactif des processus. Les conditions de réactivité d'un programme sont exprimées sous la forme de conditions de réactivité des comportements. Cette analyse est implémentée dans le compilateur de ReactiveML.

Il reste bien entendu à prouver la correction de notre analyse : si un programme est bien typé et si l'on suppose que tous les appels de fonction terminent, alors l'exécution de chaque instant du programme termine. L'intuition de la preuve est que la condition de réactivité des comportements implique que le premier instant d'un processus est décrit par un comportement de taille finie, sans aucune récursion. Le nombre de processus exécuté au cours de l'instant est donc borné.

Remerciements Nous souhaiterions remercier Julien Blond qui a accepté que nous utilisions son message qui avait été source de motivation pour travailler sur ce sujet. Merci à Florence Plateau et Marc Pouzet avec qui nous avons travaillé sur des versions précédentes de cette analyse. Merci à Gérard Boudol et Frédéric Boussinot pour les discussions sur le sujet.

Ce travail a été effectué avec le support de l'ANR Partout ANR-08-EMER-010.

Références

[1] R.M. Amadio and F. Dabrowski. Feasible reactivity in a synchronous π-calculus. In *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 221–230. ACM, 2007.

[2] T. Amtoft, F. Nielson, and H. Nielson. *Type and Effect Systems : Behaviours for Concurrency.* Imperial College Press, 1999.

[3] G. Berry. The Esterel v5 language primer. *Ecole des Mines and INRIA*, 1997.

[4] G. Boudol. Ulm : A core programming model for global computing. In David Schmidt, editor, *Programming Languages and Systems*, volume 2986 of *Lecture Notes in Computer Science*, pages 234–248. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-24725-8_17.

[5] G. Boudol. Typing termination in a higher-order concurrent imperative language. *Information and Computation*, 208(6) :716–736, 2010.

[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. Safe Reactive Programming : The FunLoft Language, 2010.

[8] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre : a declarative language for real-time programming. In *POPL '87 : Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188, New York, NY, USA, 1987. ACM.

[9] P. Caspi and M. Pouzet. A Functional Extension to Lustre. In M. A. Orgun and E. A. Ashcroft, editors, *International Symposium on Languages for Intentional Programming*, Sydney, Australia, May 1995. World Scientific.

[10] P. Cuoq and M. Pouzet. Modular Causality in a Synchronous Stream Language. In *European Symposium on Programming (ESOP'01)*, Genova, Italy, April 2001.

[11] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM.

[12] L. Mandel and M. Pouzet. ReactiveML, un langage fonctionnel pour la programmation réactive. *Technique et science informatiques*, 27(9-10) :32, 2008.

[13] Louis 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.

[14] F. Nielson and H. Nielson. Type and effect systems. *Correct System Design*, pages 114–136, 1999.

[15] B.C. Pierce. *Types and programming languages*. The MIT Press, 2002.

[16] D. Rémy. *Type inference for records in a natural extension of ML*. Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design. MIT Press, 1993.

[17] J.H. Reppy. *Concurrent programming in ML*. Cambridge University Press, 2007.

[18] C. Skalka, S. Smith, and D. Van Horn. Types and trace effects of higher order programs. *Journal of Functional Programming*, 18(02) :179–249, 2008.

[19] J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *Logic in Computer Science, 1992. LICS '92., Proceedings of the Seventh Annual IEEE Symposium on*, pages 162 –173, jun 1992.

[20] M. Tofte. Type inference for polymorphic references. *Information and computation*, 89(1) :1–34, 1990.

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

A. Preuve du systèmes de type

On définit une opération $\texttt{fst}(\kappa)$ qui ne garde que le comportement correspondant au premier instant :

$$\begin{aligned}
\texttt{fst}(0) &= \texttt{fst}(\bullet) = 0 \\
\texttt{fst}(\phi) &= \phi \\
\texttt{fst}(\texttt{run}\ \kappa) &= \texttt{run}\ (\texttt{fst}(\kappa)) \\
\texttt{fst}(\kappa_1 \parallel \kappa_2) &= \texttt{fst}(\kappa_1) \parallel \texttt{fst}(\kappa_2) \\
\texttt{fst}(\kappa_1 + \kappa_2) &= \texttt{fst}(\kappa_1) + \texttt{fst}(\kappa_2) \\
\texttt{fst}(\kappa_1; \kappa_2) &= \begin{cases} \kappa_1; \texttt{fst}(\kappa_2) & \text{si } \texttt{fst}(\kappa_1) = \kappa_1 \\ \texttt{fst}(\kappa_1) & \text{sinon} \end{cases} \\
\texttt{fst}(\mu\phi.\,\kappa) &= \texttt{fst}(\kappa[\phi \leftarrow \mu\phi.\,\kappa])
\end{aligned}$$

Dans le cas d'un comportement récursif, $\texttt{fst}(\kappa)$ n'est bien défini que si le comportement est réactif.

Propriété 1. *Si le comportement κ est réactif, alors $\texttt{fst}(\kappa)$ est un comportement fini (i.e. il est équivalent à un comportement sans comportement récursif).*

Cette propriété est la base de la preuve de correction. Le comportement de tout processus bien typé est fini, donc on sait que l'on va appeler un nombre fini de processus au cours de l'instant. On montre ainsi que l'exécution de l'instant est finie, sous réserve que les fonctions terminent. On raisonne pour cela sur la sémantique à grands pas (ou comportementale) de ReactiveML [12].

Avant d'énoncer le théorème, il faut formaliser l'hypothèse que les fonctions terminent, ce qui se traduit dans le système de types notamment par le fait que les applications ont toujours un comportement instantané 0.

Hypothèse 1. *Il existe une dérivation finie* Π *pour toute expression e (surement) instantanée, c'est-à-dire :*

$$0 \vdash e \Rightarrow \exists v.\ \frac{\Pi}{N \vdash e \xrightarrow[S]{E,b} v}$$

(Il y a la preuve dans le Lemme 8 p. 81 de [13] que les expressions instantanées se réduisent instantanément en une valeur ou se réduisent infiniment. On suppose ici que le deuxième cas ne peut pas arriver.)

La preuve est basée sur le fait que les comportement des sous-expressions sont toujours plus petits. Pour garantir cette propriété, il faut légèrement modifier le système de types. On supprime la règle Equiv qui ne vérifie évidemment pas cette propriété. Il faut également modifier les deux règles suivantes :

$$\frac{\Gamma \vdash e_1 : \tau_2 \mid 0 \qquad \Gamma \vdash e_2 : \tau_1 \to \tau_2 \to \tau_2 \mid 0 \qquad \Gamma, x : (\tau_1, \tau_2)\ \texttt{event} \vdash e : \tau \mid \kappa}{\Gamma \vdash \texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e : \tau \mid 0; \kappa}$$

$$\frac{\Gamma \vdash e_1 : \tau \mid \kappa \qquad \Gamma \vdash e_2 : (\tau_1, \tau_2)\ \texttt{event} \mid 0}{\Gamma \vdash \texttt{do}\ e_1\ \texttt{when}\ e_2 : \tau \mid \kappa + \bullet^\infty}$$

Dans les deux cas, le comportement donné à l'expression est équivalent à celui de la figure 1 mais permet d'avoir un comportement qui décroit. On montre ensuite que si une expression est bien typée dans cette variante du système, elle est aussi bien typée dans le système de la figure 1 avec un comportement équivalent. Comme l'équivalence préserve la réactivité, la correction s'applique donc aussi au système de types de l'article.

On peut maintenant énoncer et prouver la correction de notre analyse :

Théorème A.1 (Sureté)**.** *Si* $\Gamma \vdash e : \tau \mid \kappa$ *et* $[] \vdash \kappa$ *et si on suppose que "les fonctions terminent", alors il existe* e' *tel que* $N \vdash e \xrightarrow[S]{E,b} e'$ *et* $\Gamma \vdash e' : \tau \mid \kappa'$ *et* $[] \vdash \kappa'$*.*

Démonstration. On va montrer par récurrence sur la taille de $\texttt{fst}(\kappa)$ que l'on peut obtenir une dérivation finie dans la sémantique à grands pas pour tout terme bien typé. Pour prouver que l'on obtient une expression bien typée pour l'instant suivant, on étend la preuve de correction du typage, qui est faite sur la sémantique à petits pas (dont l'équivalence avec la sémantique à grands pas a été prouvée dans [12]), pour prendre en compte les comportements.

- Cas de $e_1\ e_2$ ou $\texttt{rec}\ x = e$: Par hypothèse 1.
- Cas de $\texttt{run}\ e$: On a alors que $0 \vdash e$ donc il existe Π tel que

$$\frac{\Pi}{N_1 \vdash e \xrightarrow[S]{E,true} \texttt{process}\ e_1}$$

De plus, comme $\texttt{run}\ e$ est bien typé, on a :

$$\frac{\dfrac{\Gamma \vdash e_1 : \tau \mid \kappa}{\Gamma \vdash \texttt{process}\ e_1 : \tau\, \texttt{process}[\kappa] \mid 0}}{\Gamma \vdash \texttt{run}\ (\texttt{process}\ e_1) : \tau \mid \texttt{run}\ \kappa}$$

On peut appliquer l'hypothèse de récurrence sur e_1 et on obtient donc que :

$$\frac{\Pi_1}{N_2 \vdash e_1 \xrightarrow[S]{E_1,b} e_1'}$$

et on peut donc construire la dérivation complète pour $\texttt{run}\ e$:

$$\frac{\dfrac{\Pi}{N_1 \vdash e \xrightarrow[S]{E,true} \texttt{process}\ e_1} \qquad \dfrac{\Pi_1}{N_2 \vdash e_1 \xrightarrow[S]{E_1,b} e_1'}}{N_1 \cdot N_2 \vdash \texttt{run}\ e \xrightarrow[S]{E \sqcup E_1,b} e_1'}$$

- Cas de `pause` : La dérivation est finie sans aucune hypothèse.
- Cas de $\texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$: De la même façon, on prouve que :

$$\frac{\Pi}{N_1 \vdash e \xrightarrow[S]{E,true} n}$$

La règle de typage nous donne :

$$\frac{\Gamma \vdash e : (\tau_1, \tau_2)\, \texttt{event} \mid 0 \qquad \Gamma \vdash e_1 : \tau \mid \kappa_1 \qquad \Gamma \vdash e_2 : \tau \mid \kappa_2}{\Gamma \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau \mid \kappa_1 + (\bullet; \kappa_2)}$$

On doit traiter les deux cas, selon la présence du signal :

- Cas où $n \in S$: Alors on applique l'hypothèse de récurrence sur e_1 puisque l'on a bien $\texttt{fst}(\kappa_1 + (\bullet; \kappa_2)) = \texttt{fst}(\kappa_1) + 0$ et on conclut simplement.
- Cas où $n \notin S$: Alors la dérivation est finie.

- Autres cas : idem.

□