Janvier 1996 – Journées Francophones des Langages Applicatifs – JFLA96

Réseaux de Kahn synchrones

Paul Caspi et Marc Pouzet

Laboratoire Verimag,
Projet INRIA-Spectre
{Paul.Caspi,Marc.Pouzet}@imag.fr

Résumé : Les langages à flots de données synchrones ont été introduits dans le cadre de la programmation des systèmes réactifs, et des méthodes de compilation efficaces ont pu leur être associées. Ces méthodes reposent sur une sémantique opérationnelle synchrone, et sur des vérifications statiques appelées ≪ calculs d'horloge ≫ permettant d'écarter les programmes non synchrones. Nous montrons ici que cette sémantique, et ces vérifications peuvent s'étendre à la pleine fonctionnalité. Cela indique qu'il est possible de concevoir un langage à flots de données synchrone d'usage général, dont on peut attendre qu'il partage avec ses restrictions réactives les mêmes propriétés de compilation efficace.

Mots–clés : Flots de données synchrones, évaluation sans listes, récursivité, récursivité terminale.

1. Introduction

1.1. Flots de données

Dans les années 70, le langage LUCID a été proposé [2, 1], pour pallier l'inefficacité des langages fonctionnels, en introduisant un style d'itération fondé sur les suites de valeurs. A peu près à la même époque, Gilles Kahn montrait [13] que la sémantique de réseaux de processus asynchrones et déterministes, communiquant par des files d'attente, pouvait se décrire comme des systèmes d'équations de suites, très voisins des programmes LUCID. Parallèlement, le concept d'évaluation paresseuse émergeait comme un outil pratique [10, 16], permettant d'introduire dans les langages de programmation des types de données infinis, comme les suites, et maintenant, des langages fonctionnels modernes, comme Lazy ML ou HASKELL permettent d'écrire très facilement des programmes à la LUCID ou à la Kahn, en apportant de nombreuse facilités d'expression, tels la curryfication et l'ordre supérieur.

1.2. Flots de données réactifs

Cependant, l'évaluation paresseuse est elle-même une nouvelle source d'inefficacité, et c'est pour y pallier que Wadler introduisit le concept d'évaluation *listless* [18] : l'idée était d'éviter de construire des listes intermédiaires, dans des programmes de listes paresseuses, lorsque les éléments des ces listes pouvaient être « consommés » aussitôt que créés. Ensuite, il généralisa cette idée à d'autres types récursifs, et appela cette technique *déforestation* [19].

Pendant ce temps, les automaticiens, les électroniciens, et les traiteurs de signaux, étaient confrontés au problème de passer des techniques analogiques ou discrètes, aux microprocesseurs. Ils avaient toujours utilisé les systèmes d'équations de suites comme un moyen naturel de raisonner sur leurs systèmes, et ils trouvaient ennuyeuse, et propice à erreurs, l'obligation qui leur était faite de traduire ces systèmes d'équations en programmes séquentiels. Certains d'entre eux trouvèrent qu'il était possible d'effectuer cette traduction automatiquement : c'est ainsi que de nombreux programmes du pilotage des avions Airbus A320 ont été produits. Cela conduisit des informaticiens à s'intéresser à ces techniques, et à proposer des langages tels LUSTRE ou SIGNAL, des boîtes à outils et des compilateurs pour ce type d'application [11, 3, 14] qu'ils baptisèrent *flots de données synchrones*.

Ces compilateurs se comportent de façon très similaire aux *déforesteurs* de Wadler, en n'associant qu'un élément d'un type donné à une suite d'éléments de ce type : cela procède de deux étapes :

- Il faut détecter statiquement les systèmes d'équations qui ne peuvent être évalués de façon synchrone, c'est ce que l'on nomme : *calcul d'horloge* [11, 3] ou *vérification de consistance* [14].
- Il faut ensuite compiler, sous forme de programmes séquentiels sans listes les systèmes d'équations non rejetés.

1.3. Flots de données synchrones

Mais ces langages réactifs à flots de données sont très restreints de par leur domaine d'application principal : les systèmes réactifs, qui exigent de produire des programmes à mémoire et temps de réaction bornés. En particulier, ils n'admettent ni ordre supérieur, ni fonctions récursives.

Notre objectif, dans cet article[1] est de montrer que ces restriction peuvent être facilement levées, c'est-à-dire que leurs concepts de base : l'évaluation synchrone, et le calcul d'horloge, peuvent être étendus

[1] librement adapté de [9].

raisonnablement pour prendre en compte des constructions telles que l'abstraction, laquelle permet de lever les restrictions précitées.

Pour cela, nous décrirons d'abord un noyau suffisant d'un langage synchrone à la LUSTRE étendu à l'abstraction, qui nous permettra d'illustrer les concepts de synchronisme. Ensuite, nous décrirons le calcul d'horloge de ce langage étendu. Nous conclurons sur les usages possibles d'un tel langage, et les développements envisagés.

2. Un langage à flots de données synchrone

Un tel langage peut être plongé dans un langage fonctionnel paresseux à la HASKELL[12], en définissant ainsi les opérateurs du langage :

2.1. Opérateurs primitifs

- Les suites infinies d'éléments de type a se définissent simplement par :

```
data Stream a = a : Stream a.
```

- `const` permet de transformer des valeurs scalaire en suites infinies :

```
const i = i : const i
```

  Par exemple `const True` est la suite infinie :

$$[True; True; \ldots]$$

- `extend` permet d'appliquer des suites de fonctions à des suites de valeurs :

```
extend (f:fs) (x:xs) = (f x) : (extend fs xs)
```

  Par exemple,

```
notl x    = extend (const not) x
x andl y  = extend (extend (const and) x) y
```

  définissent respectivement un inverseur et une porte « et » ; ainsi, `notl (const True)` est la suite :

$$[False; False; \ldots]$$

- `fby` est l'opérateur retard :

  ```
  (x : xs) fby  y = x : y
  ```

  En effet, pour tout n, le $n$ième élément de la suite y est le $n+1$ième élément de la suite résultat. Cet opérateur permet donc des définitions récursives de suites, (réseaux rebouclés) en évitant les interblocages; ainsi :

  ```
  half = (const True) fby (notl half)
  ```

  a le comportement infini périodique :

  $$[True; False; True; False; \ldots]$$

  Lorsque le premier argument de `fby` est une suite constante, on utilisera la fonction plus simple `pre`, dont le premier argument est scalaire :

  ```
  pre s x = s : x
  ```

  Ainsi, nous aurions pu écrire : `half = pre True (notl half)`

- `when` permet d'extraire des sous–suites de suites :

  ```
  (x : xs) when (True : cs)  = x : (xs when cs)
  (x : xs) when (False : cs) = xs when cs
  ```

  Par exemple :

  $$\begin{array}{lcl} \texttt{x} & = & [x_0; x_1; \ldots x_n; \ldots] \\ \texttt{x when half} & = & [x_0; x_2; \ldots x_{2n}; \ldots] \end{array}$$

- `merge` permet réciproquement de reconstruire des suites à partir de sous–suites :

  ```
  merge (True:cs) (x:xs) y  = x:(merge cs xs y)
  merge (False:cs) x (y:ys) = y:(merge cs x ys)
  ```

  Par exemple :

  $$\begin{array}{lcl} \texttt{x} & = & [x_0; x_1; \ldots x_n; \ldots] \\ \texttt{y} & = & [y_0; y_1; \ldots y_n; \ldots] \\ \texttt{merge half x y} & = & [x_0; y_0; x_1; y_1; \ldots x_n; y_n; \ldots] \end{array}$$

Ces primitives sont proches, à des détails syntaxiques près de celles du langage LUSTRE[11], langage qui a été utilisé avec succès dans plusieurs projets industriels logiciels.

Exemple Un programme utilisant ces primitives est donné par un ensemble d'équations, éventuellement récursives, définissant des suites de valeurs. Ainsi, le programme :

```
nat = pre 2 (extend (extend (const (+)) nat) (const 1)
```

définit la suites des entiers à partir de 2.

Ce programme peut être exécuté comme un programme gérant des listes, ce qui conduit à une implantation assez pauvre. On verra qu'il peut aussi être exécuté en évitant de gérer des listes. Pour cela, nous allons définir une *sémantique opérationnelle synchrone* pour laquelle aucune gestion de liste n'est nécessaire. Nous introduisons tout d'abord le problème par un exemple.

2.2. Pourquoi le synchronisme?

Considérons l'expression : `x andl (x when half)`. Les valeurs successives des variables peuvent être représentée par :

`x`	$=$	x_0	x_1	x_2	x_3	x_4	x_5
`half`	$=$	f	t	f	t	f	t
`x when half`	$=$		x_1		x_3		x_5
`x andl (x when half)`	$=$		x_0 and x_1		x_1 and x_3		x_2 and x_5

En terme de théorie du contrôle et de matériel, **`x when half`** est échantillonnée sur une fréquence 1/2. L'expression définie n'est pas *synchrone* : intuitivement, l'opérateur **`and`** attend ses deux arguments et les utilise à la « même vitesse » mais dans l'expression finale, le premier argument a deux fois plus d'éléments que le second. La k^{eme} valeur de la liste finale est la conjonction de la k^{eme} valeur de x et de sa $2k+1^{eme}$ valeur. D'où, pour calculer la k^{eme} valeur de la liste finale, il est nécessaire d'avoir stocké k valeurs successives de `x`. Par conséquent, il n'est pas possible de remplacer la liste `x` par une valeur scalaire.

3. La sémantique synchrone

La sémantique opérationnelle synchrone est définie par la relation de transition $\sigma \vdash e \xrightarrow{v} e'$ signifiant que dans l'environnement σ, un pas d'exécution de l'expression e produit la valeur v et se transforme en l'expression e'. Les environnements et les valeurs ont la forme :

$$\sigma ::= [x_1 \xrightarrow{v_1} y_1, \ldots, x_n \xrightarrow{v_n} y_n] \qquad v ::= [] \mid [e] \text{ avec } scalar(\text{e})$$

où x_i et y_i dénotent des noms de variables, et une valeur v est soit une expression scalaire e soit vide (dénotée par $[]$). Une exécution de e dans l'environnement σ_0 peut s'écrire :

$$e = e_0 \xrightarrow{v_0} e_1 \xrightarrow{v_1} e_2 \ldots \xrightarrow{v_n} e_{n+1} \xrightarrow{v_{n+1}} \ldots$$

ssi $\sigma_0 \vdash e_i \xrightarrow{v_i} e_{i+1}$. La valeur produite par l'expression e est la suite infinie $v_0.v_1 \ldots v_n \ldots$. La relation $\vdash$ est décrite par les axiomes et règles de la table 1.

- `const` e produit e et se réécrit en elle-même. Si aucune opération n'attend cette valeur, la valeur n'a pas à être produite. On parle alors de règle vide.
- la primitive `extend` applique une suite de fonctions scalaires à une suite de valeurs. La sémantique synchrone dit que les deux arguments doivent être, soit simultanément présents, soit simultanément absents.

  Remarque : Nous voyons ici le caractère synchrone de cette sémantique, qui tient au fait qu'il manque des règles : en effet, en toute rigueur, la définition du `extend` devrait comporter 4 règles, obtenues en croisant les cas de production et de non production des deux arguments. En n'en donnant que deux, qui correspondent aux seuls cas de consommation immédiate, nous interdisons l'évaluation d'expressions nécessitant le stockage de résultats intermédiaires sous forme de listes.
- La sémantique de `merge` est la suivante : si la condition produit une valeur vraie, le `merge` produit la valeur produite par le deuxième argument, sinon, la valeur produite par le troisième argument. Dans le cas où aucune valeur n'est produite par la condition, les autres arguments doivent se comporter comme des règles vides.
- Une expression `when` produit la valeur de son second argument seulement si la valeur du premier est vraie. Sinon, la valeur du premier argument est oubliée.
- L'opérateur `fby` ne peut produire une valeur que si son premier argument peut en produire une. Dans ce cas, il se réécrit en `pre` .
- L'opération `pre` consiste à placer un élément en tête de liste, comme le constructeur `cons` classique dans les langages fonctionnels. Nous trouvons ici un cas particulier où l'expression réécrite est fonction de x_0, c'est-à-dire la valeur produite par l'argument x.
- La règle pour une expression récursive est très classique. Cependant, il faut remarquer que, dans le fragment réactif, ne

$$\frac{}{\sigma, x \xrightarrow{v_0} x_1 \vdash x \xrightarrow{v_0} x_1}$$

$$\frac{}{\sigma \vdash \mathtt{const}\ \ e \xrightarrow{[e]} \mathtt{const}\ \ e} \qquad \frac{}{\sigma \vdash \mathtt{const}\ \ v \xrightarrow{[]} \mathtt{const}\ \ v}$$

$$\frac{\sigma \vdash f \xrightarrow{[]} f_1 \quad \sigma \vdash x \xrightarrow{[]} x_1}{\sigma \vdash \mathtt{extend}\ \ f\ x \xrightarrow{[]} \mathtt{extend}\ \ f_1\ x_1}$$

$$\frac{\sigma \vdash f \xrightarrow{[f_0]} f_1 \quad \sigma \vdash x \xrightarrow{[x_0]} x_1}{\sigma \vdash \mathtt{extend}\ \ f\ x \xrightarrow{[f_0 x_0]} \mathtt{extend}\ \ f_1\ x_1}$$

$$\frac{\sigma \vdash c \xrightarrow{[]} c_1 \quad \sigma \vdash x \xrightarrow{[]} x_1 \quad \sigma \vdash y \xrightarrow{[]} y_1}{\sigma \vdash \mathtt{merge}\ \ c\ x\ y \xrightarrow{[]} \mathtt{merge}\ \ c_1\ x_1\ y_1}$$

$$\frac{\sigma \vdash c \xrightarrow{[t]} c_1 \quad \sigma \vdash x \xrightarrow{[x_0]} x_1 \quad \sigma \vdash y \xrightarrow{[]} y_1}{\sigma \vdash \mathtt{merge}\ \ c\ x\ y \xrightarrow{[x_0]} \mathtt{merge}\ \ c_1\ x_1\ y_1}$$

$$\frac{\sigma \vdash c \xrightarrow{[f]} c_1 \quad \sigma \vdash x \xrightarrow{[]} x_1 \quad \sigma \vdash y \xrightarrow{[y_0]} y_1}{\sigma \vdash \mathtt{merge}\ \ c\ x\ y \xrightarrow{[y_0]} \mathtt{merge}\ \ c_1\ x_1\ y_1}$$

$$\frac{\sigma \vdash c \xrightarrow{[]} c_1 \quad \sigma \vdash x \xrightarrow{[]} x_1}{\sigma \vdash x\ \mathtt{when}\ \ c \xrightarrow{[]} x_1\ \mathtt{when}\ \ c_1}$$

$$\frac{\sigma \vdash c \xrightarrow{[t]} c_1 \quad \sigma \vdash x \xrightarrow{[x_0]} x_1}{\sigma \vdash x\ \mathtt{when}\ \ c \xrightarrow{[x_0]} x_1\ \mathtt{when}\ \ c_1}$$

$$\frac{\sigma \vdash c \xrightarrow{[f]} c_1 \quad \sigma \vdash x \xrightarrow{[x_0]} x_1}{\sigma \vdash x\ \mathtt{when}\ \ c \xrightarrow{[]} x_1\ \mathtt{when}\ \ c_1}$$

$$\frac{\sigma \vdash x \xrightarrow{[]} x_1}{\sigma \vdash \mathtt{pre}\ \ v\ x \xrightarrow{[]} \mathtt{pre}\ \ v\ x_1} \qquad \frac{\sigma \vdash x \xrightarrow{[x_0]} x_1}{\sigma \vdash \mathtt{pre}\ \ v\ x \xrightarrow{[v]} \mathtt{pre}\ \ x_0\ x_1}$$

$$\frac{\sigma \vdash x \xrightarrow{[]} x_1 \quad \sigma \vdash y \xrightarrow{[]} y_1}{\sigma \vdash x\ \mathtt{fby}\ \ y \xrightarrow{[]} x_1\ \mathtt{fby}\ \ y_1} \qquad \frac{\sigma \vdash x \xrightarrow{[x_0]} x_1 \quad \sigma \vdash y \xrightarrow{[]} y_1}{\sigma \vdash x\ \mathtt{fby}\ \ y \xrightarrow{[]} \mathtt{pre}\ \ x_0\ y_1}$$

$$\frac{\sigma \vdash x \xrightarrow{[x_0]} x_1 \quad \sigma \vdash y \xrightarrow{[y_0]} y_1}{\sigma \vdash x\ \mathtt{fby}\ \ y \xrightarrow{[x_0]} \mathtt{pre}\ \ y_0\ y_1}$$

$$\frac{\sigma, x \xrightarrow{x_0} x_1 \vdash e \xrightarrow{e_0} e_1}{\sigma \vdash rec\ \ x.e \xrightarrow{rec\ x_0.e_0} rec\ \ x_1.e_1[rec\ \ x_0.e_0/x_0]}$$

Table 1: Sémantique opérationnelle synchrone du fragment réactif

$$ABST \quad \frac{\sigma, x \xrightarrow{x_0} x_1 \vdash e \xrightarrow{e_0} e_1}{\sigma \vdash \lambda x.e \xrightarrow{\lambda x_0.e_0} \lambda x_0.\lambda x_1.e_1}$$

$$abst \quad \frac{\sigma \vdash e \xrightarrow{e_0} e_1}{\sigma \vdash \lambda x.e \xrightarrow{\lambda x.e_0} \lambda x.e_1}$$

$$APP \quad \frac{\sigma \vdash f \xrightarrow{f_0} f_1 \quad \sigma \vdash e \xrightarrow{e_0} e_1}{\sigma \vdash f\ e \xrightarrow{f_0\ e_0} f_1\ e_0\ e_1}$$

$$app \quad \frac{\sigma \vdash f \xrightarrow{f_0} f_1}{\sigma \vdash f\ v \xrightarrow{f_0\ v} f_1\ v}$$

Table 2: Sémantique opérationnelle synchrone de l'extension

disposant pas de l'abstraction, son comportement est particulier. En effet, dans le langage des scalaires, une valeur telle que $rec\, x.x + 1$ est interdite car elle conduit à une récursion infinie. Par conséquent, la valeur $rec\, x_0.e_0$ apparaissant dans la règle n'est correcte que si x_0 n'est pas libre dans e_0. C'est toujours le cas en LUSTRE, car de tels blocage sont facilement détectable statiquement : il suffit de vérifier que toute variable non scalaire apparaissant dans la portée d'un *rec* est sous la portée d'un `pre` .

Ces règles sont celles du fragment réactif correspondant à LUSTRE. Celles qui suivent concernent l'extension fonctionnelle (table 2) :

- La règle de l'abstraction *ABST* ressemble à la règle classique. Cependant, il faut noter qu'une fonction de suite est une fonction dynamique, qui peut avoir besoin de mettre à jour une mémoire interne. Ainsi, il se peut que l'expression e_1 ait la variable instantanée x_0 comme variable libre. C'est notamment le cas lorsque l'expression e contient des opérateurs `pre`. C'est pourquoi on doit aussi abstraire sur la variable instantanée.
- Cela nous montre aussi qu'il se peut que l'on ait à abstraire sur une variable instantanée.
- A ces deux abstractions correspondent deux règles d'application, *APP* et *app*, selon que l'expression argument est une expression de suite, ou une expression scalaire.

Il est facile de voir, d'une part, que cette sémantique permet d'exécuter les programmes qui ne nécessitent pas de listes intermédiaire, et que, d'autre part, les programmes nécessitant de telles listes ne peuvent être exécutés, puisqu'il n'y a pas de règle permettant de les manipuler.

4. Calcul d'horloge

L'idée est de vérifier statiquement qu'un programme est évaluable de façon synchrone. Nous présentons ici une version légèrement différente que celles données dans [8, 11]. Il s'agit d'une généralisation dans le sens où elle utilise l'unification au lieu de points fixes. Cette version rend plus facile la généralisation aux extensions fonctionnelles et la preuve du théorème 1.

Le but du calcul d'horloge est de montrer le jugement :

$$H \vdash e : cl$$

qui s'interprète comme : « l'expression e a pour horloge cl dans l'environnement H ». L'environnement H est une liste d'hypothèses sur les horloges des variables libres de e. H est tel que :

$$H ::= [x_0 : cl_0, ..., x_n : cl_n]$$

cl est soit une variable horloge α, soit une sous–horloge d'une autre horloge, cl `on` e contrôlée par une expression booléenne e, soit une horloge fonctionnelle. De plus, comme dans le typage classique à la Hindley-Milner [15], il nous faut introduire une notion de polymorphisme d'horloges, ce qui est obtenu en créant une classe de schémas d'horloges :

$$\begin{array}{lcl} cl & ::= & \alpha \mid cl \;\texttt{on}\; e \mid cl \xrightarrow{x} cl \\ \sigma & ::= & cl \mid \forall \alpha_1 \ldots \alpha_n . cl \end{array}$$

Cependant, il faut remarquer qu'il ne s'agît pas d'un système de Hindley-Milner conventionnel : en effet, il se peut que les expressions d'horloges contiennent des expressions ordinaires (cf. le cas du constructeur `on`). Cela explique pourquoi, les expressions d'horloges fonctionnelles doivent se souvenir de l'identité de la variable abstraite, car celle-ci peut apparaître dans l'expression d'horloge du résultat. Les axiomes et règles d'inférence sont donnés dans la table 3 :

- Une constante s'unifie avec n'importe quelle horloge. Elle a l'horloge polymorphique α.
- Les horloges des deux arguments d'une expression `extend` doivent être les mêmes.
- L'horloge d'une expression `when` dépend de la valeur du deuxième argument. Elle est représentée en utilisant la construction `on` .

- L'expression `merge` e_1 e_2 e_3 utilise un élément de e_2 quand la valeur de e_1 est vraie, sinon, elle utilise un élément de e_3. Par conséquent, une telle expression a une horloge correcte quand l'horloge de e_2 est celle de e_1 restreinte au cas où e_1 est vrai et quand l'horloge de e_3 est celle de e_1 restreinte au cas où e_1 est faux. Il faut aussi considérer la règle symétrique.

- La règle du *rec* est la règle classique de calcul du type.

Ces règles sont celles du fragment réactif. Celles qui suivent concernent l'extension fonctionnelle (table 4).

- Les règles GEN et $INST$ sont les règles usuelles du polymorphisme.

- La règle APP montre la nécessité de se souvenir de l'identité de la variable abstraite par la règle $ABST$; en effet, si la variable x est libre dans l'expression d'horloge résultat cl', il faudra lui substituer la valeur de l'argument effectif.

- Enfin, les règles *abst* et *app* traitent des cas où l'on abstrait et applique des scalaires.

Cela nous permet d'établir le principal résultat de ce texte, à savoir que toute expression ayant une horloge correcte peut être évaluée de façon synchrone.

Théoreme 1 (Consistance) *Pour tout e, H et cl, si $H \vdash e : cl$ alors il existe σ, v, e', H' et cl' tels que*

i) $\sigma \vdash e \xrightarrow{v} e'$, *et*

ii) $H' \vdash e' : cl'$.

L'idée de la preuve est la suivante : à chaque arbre de preuve d'horloge on peut associer au moins un arbre d'exécution en construisant un morphisme ϕ des noeuds de la preuve d'horloge vers les noeuds de la preuve d'exécution. Et ce morphisme préserve une propriété de consistance : toutes les expressions partageant la même horloge peuvent soit produire une valeur, soit s'évaluer à ≪ vide ≫ , et toute expression ayant une sous–horloge produit une valeur si et seulement si son horloge s'évalue à vrai. Alors, le fait que la réécriture d'une expression ait elle aussi une horloge correcte garantit que l'exécution va se poursuivre de même. □

$$CONST \quad \frac{}{H \vdash \texttt{const}\ \ v : \alpha}$$

$$TAUT \quad \frac{}{H, x : cl \vdash x : cl}$$

$$EXT \quad \frac{H \vdash e_1 : cl \quad H \vdash e_2 : cl}{H \vdash \texttt{extend}\ \ e_1\ e_2 : cl}$$

$$WHEN \quad \frac{H \vdash e_1 : cl \quad H \vdash e_2 : cl}{H \vdash e_1\ \texttt{when}\ \ e_2 : cl\ \texttt{on}\ e_2}$$

$$MERGE \quad \frac{H \vdash e_1 : cl \quad H \vdash e_2 : cl\ \texttt{on}\ e_1 \quad H \vdash e_3 : cl\ \texttt{on}\ (\texttt{notl}\ e_1)}{H \vdash \texttt{merge}\ \ e_1\ e_2\ e_3 : cl}$$

$$PRE \quad \frac{H \vdash e : cl}{H \vdash \texttt{pre}\ \ v\ e : cl}$$

$$MERGE' \quad \frac{H \vdash e_1 : cl \quad H \vdash e_2 : cl\ \texttt{on}\ (\texttt{notl}\ e_1) \quad H \vdash e_3 : cl\ \texttt{on}\ e_1}{H \vdash \texttt{merge}\ \ (\texttt{notl}\ e_1)\ e_2\ e_3 : cl}$$

$$FBY \quad \frac{H \vdash e_1 : cl \quad H \vdash e_2 : cl}{H \vdash e_1\ \texttt{fby}\ \ e_2 : cl}$$

$$FBY' \quad \frac{H \vdash e_1 : cl \quad H \vdash e_2 : cl\ \texttt{on}\ e}{H \vdash e_1\ \texttt{fby}\ \ e_2 : cl\ \texttt{on}\ e}$$

$$REC \quad \frac{H, x : cl \vdash e : cl}{H \vdash rec\ \ x.e : cl}$$

Table 3: Calcul d'horloge du fragment réactif

$$INST \quad \frac{H \vdash e : \forall \alpha_1 ... \alpha_n . cl \quad FV(cl) \cap FV(cl_i) = \emptyset}{H \vdash e : cl[cl_1/\alpha_1, ..., cl_n/\alpha_n]}$$

$$GEN \quad \frac{H \vdash e : cl \quad \alpha_1, ..., \alpha_n \notin FV(H)}{H \vdash e : \forall \alpha_1 ... \alpha_n . cl}$$

$$ABST \quad \frac{H, x : cl \vdash e : cl'}{H \vdash \lambda x.e : (cl \xrightarrow{x} cl')}$$

$$abst \quad \frac{H \vdash e : cl}{H \vdash \lambda x.e : cl}$$

$$APP \quad \frac{H \vdash e : cl \xrightarrow{x} cl' \quad H \vdash e' : cl}{H \vdash e\ e' : cl'[e'/x]}$$

$$app \quad \frac{H \vdash e : cl \quad scalar(e')}{H \vdash e\ e' : cl}$$

Table 4: Calcul d'horloge de l'extension

5. Exemples

5.1. Un exemple vraiment récursif: le crible d'Erathostene

Le programme adapté de Kahn [13] est le suivant :

```
let first x =
       let v = x fby v
       in  v

    sieve x =
        let filter = (first x) not_divides_l x
        in  merge filter
                 (sieve (x when filter))
                 (pre True (const  False))

in nat when (sieve(nat))
```

où `not_divides_l` est l'extension aux suites de l'opération correspondante sur les scalaires.

Il nous semble qu'il est assez facile de se convaincre que ce programme est synchrone: il n'a besoin que d'une pile de processus s'échangeant des valeurs ordinaires, et non des listes de valeurs; on montre aussi, assez facilement que le calcul d'horloge confirme cette impression. Enfin, cet exemple illustre aussi l'intérêt des horloges : Chaque processus `sieve` appelle son fils avec une horloge ≪ plus rare ≫ [2] que celle avec laquelle on l'appelle. Si cela n'avait pas été le cas, on aurait risqué d'avoir une récursion infinie.

5.2. Des exemples réactifs à récursion terminale

L'idée de récursion terminale, telle que proposée dans [7], est fondée sur le principe d'horloges qui sont initialement vraies, et qui, dès qu'elles deviennent fausses, le restent pour toujours. De telles horloges, utilisées en conjonction avec des opérateurs `merge`, permettent à un processus de se reconvertir, pour toujours, en un processus fils.

Par exemple, la fonction `while` transforme toute suite booléenne en une suite ayant cette propriété :

```
init = pre True (const(False))

while c =
  let w = merge init
                (const True)
                ((c andl (pre True w)) whennot init)
 in w
```

où `whilenot` et `whennot` sont les fonctions correspondantes, précédées d'un inverseur `notl`.

Réinitialisation d'une fonction dynamique : Une telle fonction, associée à une construction récursive, permet de réinitialiser une fonction dynamique, sans la reprogrammer :

```
reset_f r x =
  let w = whilenot r
  in  merge w
            (f (x when w))
            (reset_f ( r whennot w)
                     ( x whennot w))
```

On voit que, tant que `r` n'est pas vrai, la sortie de la fonction est celle de

[2] Ce qui correspond au fait que la suite booléenne `filter` ne vaut pas toujours ≪ vrai ≫ .

`f`. Dès que `r` devient vrai, `w` devient faux pour toujours, et une nouvelle fonction `reset_f` est activée, qui reçoit ses premiers échantillons, et se trouve donc dans son état initial.

Réinitialisation interne : Ici, `r` est une suite externe. On peut imaginer la même construction avec une suite interne :

```
loop_body x =
    let (yy, done) = body(x)
        w          = whilenot done
    in  merge w
              (yy when w)
              (loop_body (x whennot w))
```

Il s'agît, là aussi, d'une récursion terminale, qui montre comment ce type de récursion peut aider à la programmation d'applications réactives, en l'enrichissant de structures de contrôle.

Récursion mutuelle : Cette technique semble permettre de coder des automates de façon plaisante. Par exemple, un automate à deux états, marche–arrêt pourra s'écrire :

```
on_off  (start, stop) =
  let w  = whilenot stop
  in  merge w
           (const True)
           (off_on (( start, stop) whennot w)

off_on  (start, stop )=
  let w  = whilenot start
  in  merge w
           (const False)
           (off_on ( (start,stop) whennot w)
```

Comme nous l'avons déjà remarqué dans [7] cet exemple peut être dessiné, et il montre alors un moyen curieux de représenter, sur un même plan, le flot des données et le flot de contrôle d'un programme!

6. Conclusion

Nous avons donc proposé une sémantique opérationnelle synchrone qui généralise à la pleine fonctionnalité celle correspondant aux langages réactifs à flots de données synchrones. Nous avons aussi généralisé les règles de sémantique statique (dites calcul d'horloges) qui fournissent des conditions suffisantes pour que des programmes soient évaluables de façon synchrone.

Dans des publications précédentes, nous avions déjà examiné l'intérêt de telles extensions, pour la programmation en général [6], et même pour la programmation de systèmes réactifs [7], où nous avions montré que le concept de récursion terminale permet de mieux structurer des programmes réactifs en fournissant, dans un cadre strictement fonctionnel, des constructions apparentées à celles de langages impératifs à la ESTEREL [4].

D'autre part, ce travail est clairement relié aux travaux sur la déforestation [18, 19]. Mais le travail le plus proche est sans doute celui de Vree et Hartel [17]. Cependant, leur approche ne s'applique pas à des fonctions ne préservant pas la longueur, telles que `merge` , qui sont pourtant centrales dans notre approche. Enfin, il faut noter le travail de Frédéric Boussinot sur les réseaux réactifs [5], qui décrit de même des réseaux synchrones, mais ne fait pas le lien avec un calcul d'horloges en vue d'une compilation efficace des canaux.

Ces travaux de compilation sont en cours et visent à introduire un type de flots paresseux synchrones, donc compilable efficacement, dans un langage fonctionnel strict à la ML.

References

[1] E. A. Ashcroft and W. W. Wadge. *Lucid, the data-flow programming language.* Academic Press, 1985.

[2] E.A. Ashcroft and W.W. Wadge. Lucid, a formal system for writing and proving programs. *SIAM j. Comp.*, 3:336–354, 1976.

[3] A. Benveniste, P. LeGuernic, and Ch. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[4] G. Berry and G. Gonthier. The Esterel synchronous programming language, design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[5] F. Boussinot. Réseaux de processus réactifs. Rapport de Recherche 1588, INRIA, Sophia-Antipolis, janvier 1992.

[6] P. Caspi. Lucid synchrone. In *Actes du colloque INRIA OPOPAC, Lacanau.* HERMES, novembre 1993.

[7] P. Caspi. Towards recursive block diagrams. In *Proc. 19th IFAC/IFIP Workshop on real-time programming, Isle of Reichenau, Germany*. IFAC, June 1994.

[8] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. Lustre: a declarative language for programming synchronous systems. In *Proc. 14th ACM Symposium on Principles of Programming Languages*. ACM, 1987.

[9] P. Caspi and M. Pouzet. A functional extension to LUSTRE. In M. A. Orgun and E. A. Ashcroft, editors, *8th Int. Symp. on Languages for Intensional Programming*. World Scientific, May 1995. to appear.

[10] D.P. Friedman and D.S. Wise. CONS should not evaluate its arguments. In *Proc. ICALP'76*, pages 257–284, 1976.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[12] P. Hudak, S. Peyton Jones, and P. Wadler. Report on the programming language haskell, a non strict purely functional language (version 1.2). *ACM SIGPLAN Notices*, 27(5), 1990.

[13] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74 Congress*. North Holland, Amsterdam, 1974.

[14] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers.*, 36(2), 1987.

[15] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[16] D.A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, 9:31–49, 1979.

[17] W.G. Vree and P.H. Hartel. Communication lifting: fixed point computation for parallelism. *J. Functional Programming*, 1(1):1–33, 1993.

[18] P. Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile time. In *Proc. 1984 ACM Symposium on LISP and Functional Programming*, pages 45–52, 1984.

[19] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.