# Causalité Modulaire dans un Langage de Flots Synchrone

Pascal Cuoq[1] & Marc Pouzet[2]

*1: INRIA Rocquencourt*
Pascal.Cuoq@inria.fr
*2: Laboratoire d'Informatique de Paris 6*
Marc.Pouzet@lip6.fr
Ce travail a été partiellement financé par le CNET/France Telecom

## Résumé

Nous présentons une analyse de causalité pour un langage de flots synchrones avec ordre supérieur. Cette analyse prend la forme d'un système de types avec rangées (analogues à celles conçues pour l'ajout d'enregistrements extensibles au système de types de ML). Nous proposons aussi une preuve de correction de cette analyse vis-à-vis de la sémantique co-itérative des flots de Lucid Synchrone.

## 1. Introduction

### 1.1. Historique

Cette section présente, par ordre chronologique, différents langages faisant de la manipulation de flots infinis leur paradigme de programmation, et situe Lucid Synchrone[3] parmi ceux-ci.

Lucid[1] a été le premier langage à proposer les flots comme valeurs de première classe. Son but était de rapprocher les langages de programmation de systèmes logiques dans lesquels on peut exprimer et prouver des propriétés. Dans Lucid, des équations récursives sur les flots remplaçaient l'itération impérative.

Ensuite sont apparus d'une part des langages de flots synchrones tels que Lustre[9], Signal[2]. Lustre et Signal sont utilisés dans les domaines du contrôle d'automates et du traitement du signal. Lustre et Signal ne disposent pas de fonctions d'ordre supérieur.

D'autre part, ont été développés des langages fonctionnels paresseux (par exemple Haskell[10]). Les langages paresseux peuvent manipuler des structures de données infinies en général, et des flots en particulier. La représentation des flots comme des listes infinies dans un langage paresseux n'a pas la propriété de réactivité (réaction en un temps borné aux entrées), et manque d'efficacité pour le genre d'applications dans lesquelles Lustre et Signal sont utilisés.

Ces deux familles évoluèrent de manière relativement indépendante, chacune manquant d'un trait considéré comme fondamental dans le domaine d'application de l'autre.

Pour les langages fonctionnels, Wadler a proposé un ensemble de techniques, sous le nom de "déforestation"[19], pour éviter d'allouer les listes qui sont dé-construites immédiatement. Dans certains cas, les programmes optimisés résultant sont très similaires à ceux qui peuvent être générés par Lustre ou Signal.

Lucid Synchrone est un rapprochement entre la famille des langages fonctionnels et celle des langages de flots synchrones, proposant autant de traits des premiers que possible (modularité, ordre supérieur, polymorphisme) tout en utilisant une compilation efficace des flots [4]. Par comparaison à la déforestation qui est une optimisation s'appliquant ou non, Lucid Synchrone, de par sa syntaxe et ses analyses statiques, ne permet d'écrire que des programmes synchrones.

## 1.2. Un exemple tutoriel

Le domaine d'application des langages synchrones est la conception de systèmes réactifs. Le concept mathématique représentant le mieux un système réactif est un automate de Mealy, c'est-à-dire un automate fini recevant une valeur d'entrée et calculant une valeur de sortie déterminée par l'entrée et par l'état dans lequel les entrées précédentes avaient laissé le système. Par exemple une machine à café pourra, pour l'entrée "Appui bouton", réagir en bipant ou en produisant un café, selon que l'entrée "pièce insérée" a été vue récemment ou non.

Comme la sortie courante dépend, outre de l'entrée courante, de l'état du système, qui lui même est fixé par les entrées précédentes, il est plus pratique de voir un système réactif comme un calcul transformant le flot des entrées en flot de sorties, étant entendu que la $n$-ième valeur de sortie ne peut dépendre que des $n$ premières valeurs d'entrée.

Ce point de vue permet de programmer facilement de nombreux comportements utiles. De plus, comme la notion d'état disparaît entièrement des programmes, il devient aussi plus facile de prouver des propriétés sur ces programmes.

Voici comment le mécanisme de notre machine à café simpliste pourrait être exprimé en Lucid Synchrone :

```
let rec nombre_pieces = prec_nombre_pieces + pieces_inserees -
(if servir_cafe then 1 else 0)
and prec_nombre_pieces = pre 0 nombre_pieces
and servir_cafe = appui_bouton and (prec_nombre_pieces >= 1)
```

`appui_bouton` et `pieces_inserees` sont les flots d'entrée. `appui_bouton` est un flot booléen, et `pieces_inserees` est un flot d'entiers qui indique le nombre de pièces insérées à l'instant t.

Le flot de sortie est `servir_cafe` ; les autres flots sont seulement définis comme intermédiaires de calcul. Les flots de sorties peuvent être utilisés dans le calcul des autres flots, comme c'est le cas ici pour `nombre_pieces`.

`(if servir_cafe then 1 else 0)` est un flot entier qui prend la valeur `1` à l'instant t si un café est servi à l'instant t, et `0` sinon. On suppose que servir un café est une tâche instantané.

Un autre aspect des systèmes réactifs sur lequel nous n'avons pas encore suffisamment insisté est qu'ils doivent en général être capables de proposer une sortie en un temps borné après qu'on leur ait fournit une entrée. Cet aspect est un peu mis de coté dans les langages synchrones, dans lesquels il est admis (mais peut être vérifié *a posteriori* si nécessaire) que les calculs sont toujours suffisamment rapides. Toutefois, une conséquence importante est que la récursion dans un instant donné est absente du langage. L'opérateur de récursion du langage ne sert qu'à définir des flots récursivement, en utilisant la valeur de l'instant t-1 pour calculer la valeur de l'instant t (`nombre_pieces` est un exemple typique). Dans le genre d'application pour lequel les langages synchrones sont utilisés, ce manque d'expressivité apparent est rarement un problème, alors que réagir en temps borné est une nécessité première.

## 1.3. Plan

Dans la section 2, nous décrivons le noyau de langage qui servira de base à cet article. La section 3 décrit l'analyse de causalité qui est la partie originale de cet article. La section 4 donne une brève présentation de la sémantique co-itérative qui peut être utilisée pour donner une sémantique formelle à notre noyau de langage. Dans la section 5, nous prouvons que l'analyse décrite en 3 est correcte vis-à-vis de la sémantique décrite en 4.

## 2. Un noyau de langage de flots synchrones

Voici le noyau de LUCID SYNCHRONE qui servira de base à cet article :

| programmes : | | |
|---|---|---|
| $t ::=$ | $(t_1, t_2)$ | paire |
| | $\mid$ `fst` $t$ $\mid$ `snd` $t$ | projections |
| | $\mid x$ | variable |
| | $\mid \lambda x.t$ | $\lambda$-abstraction |
| | $\mid t_1\ t_2$ | application |
| | $\mid$ `let` $x = t_1$ `in` $t_2$ | let non récursif |
| | $\mid$ `rec` $x_1 = t_1$ `and` $x_2 = t_2$ | définition récursive |
| | $\mid c$ | constantes |
| | $\mid p$ | primitives |
| | $\mid$ `pre` $c\ t$ | un retard, initialisé avec la constante $c$ |

Les valeurs constantes $c$ sont importées depuis un langage scalaire externe (OCAML dans le cas de LUCID SYNCHRONE), et étendues en un flot constant. De même, les fonctions primitives importées du monde scalaire sont étendues point-à-point sur les flots : la fonction `f` du monde scalaire devient la fonction associant au flot $(\mathtt{u}_1, \mathtt{u}_2, \ldots)$ le flot $(\mathtt{f}\ \mathtt{u}_1, \mathtt{f}\ \mathtt{u}_2, \ldots)$.

L'abstraction, l'application, la construction de paire, les projections, et le combinateur de récursion se comportent exactement comme dans le $\lambda$-calcul.

Enfin, `pre` $c\ u$ concatène la valeur scalaire $c$ devant le flot $u$. Dans l'interprétation temporelle des flots comme valeurs successives prises par un noeud au cours du temps, `pre` introduit un retard d'un instant.

| $u$ | $u_1$ | $u_2$ | $u_3$ | $u_4$ | $\ldots$ |
|---|---|---|---|---|---|
| `pre` $c\ u$ | $c$ | $u_1$ | $u_2$ | $u_3$ | $\ldots$ |

Beaucoup de langages de flots synchrones (Lustre, Signal, Lucid Synchrone,...) ont en commun un trait qui n'est pas présent dans ce noyau : une horloge est associée à chaque expression. Un système d'horloges permet d'extraire un sous-flot d'un flot par *échantillonage*, et la cohérence du système d'horloges garantit que le programme reste synchrone bien qu'il comporte des flots avançant à des vitesses différentes. Cette analyse est essentiellement orthogonale à l'analyse de causalité, c'est pourquoi nous n'en parlons pas plus ici.

## 3. Analyse de causalité des programmes Lucid Synchrone

Dans le monde réel, les lois de la causalité interdisent l'utilisation d'informations du futur dans le présent. De telles interférences ne peuvent se produire dans le contexte d'un langage de flots synchrones sans horloges, car il n'existe pas d'opérateur pour exprimer le futur. Cependant, les calculs étant supposés instantanés, la causalité peut être enfreinte en utilisant dans l'instant la valeur même qui est en train d'être définie.

Les définitions récursives telles que `rec x=x` (qui est valide en tant qu'équation, mais ne définit pas un flot) ou `rec x=x+1` (qui n'a pas de solution) doivent donc être détectées et rejetées statiquement. C'est le but de l'analyse de causalité.

Par contre, grâce au retard introduit par l'opérateur `pre`, l'expression $1 + (\texttt{pre}\ 0\ x)$ ne dépend plus instantanément de $x$ ; la définition récursive $\texttt{rec}\ x = 1 + (\texttt{pre}\ 0\ x)$, qui dénote le flot des entiers strictement positifs, est valide, et doit être acceptée.

Dans le cas de définitions mutuellement récursives, il serait trop restrictif d'imposer à chaque expression à la droite d'un signe = de contenir un `pre`. Par exemple, $\texttt{rec}\ x = y + 1\ \texttt{and}\ y = \texttt{pre}\ 0\ x$ est une définition valide de deux flots $x$ et $y$.

L'analyse de causalité qui va être présentée a été conçue pour accepter ce genre de programmes et être compatible avec les fonctions d'ordre supérieur.

Cette analyse prend en fait la forme d'un système de types avec rangées. Elle comporte le let-polymorphisme des systèmes de type de ML et Haskell. La principale différence est que toutes les expressions qui auraient eu un type de base (int, float, bool, ...) dans un système classique ont ici un type de la forme $\{\varphi\}$. La rangée $\varphi$ encode les dépendances de l'expression en question.

On souhaite bien sûr pouvoir, pour une même expression, indiquer ses dépendances vis-à-vis de plusieurs variables de récursion (voire toutes). C'est pour cette raison que $\varphi$ est une rangée ([20],[14]) dont les étiquettes sont les noms des variables de récursion. Les rangées ont été conçues à l'origine pour introduire des enregistrements extensibles en ML. D'une manière générale, elles permettent dans certains cas (dont le nôtre) d'utiliser le polymorphisme pour remplacer le sous-typage.

Les rangées suivent la syntaxe suivante :

rangées :

| | | |
|---|---|---|
| $\varphi ::=$ | $\rho$ | variable de rangée |
| | $\mid x : \pi, \varphi$ | un champ concernant la variable $x$ |

Sous quelle forme la dépendance $\pi$ de l'expression par rapport à une variable particulière $x$ doit-elle être représentée ? Dans le contexte qui nous préoccupe, il est suffisant, pour une expression donnée, de pouvoir exprimer si l'on est dans l'un des trois cas suivant :

1. l'expression est effectivement considérée comme dépendant de $x$.
2. il est interdit que l'expression dépende de $x$
3. ou bien, on ne s'est encore engagé ni pour l'une ni pour l'autre des deux premières possibilités.

Ces trois cas sont représentés par la grammaire suivante :

information de présence :

| | | |
|---|---|---|
| $\pi ::=$ | $a$ | absent |
| | $\mid p$ | présent |
| | $\mid \delta$ | variable de présence |

Une variable $\delta$ peut être instanciée en un atome $a$ ou $p$.

On voudrait aussi que les rangées $\{x : p, y : p, \varphi_1\}$ et $\{y : p, x : p, \varphi_2\}$ soient unifiables si les rangées $\varphi_1$ et $\varphi_2$ le sont. Pour cette raison, les rangées que nous manipulerons seront en fait les classes d'équivalence des rangées syntaxiques pour la relation de commutation (par exemple $x : \pi, y : \pi', \varphi = y : \pi', x : \pi, \varphi$). Comme d'habitude, on notera une telle classe sous la forme de l'un de ses représentants avec les abus de notation usuels.

Pour que les théorèmes généraux pour les systèmes avec rangées démontrés dans [13, 14] s'appliquent, il faut encore définir une notion de sorte sur celles-ci. Comme pour notre théorie équationnelle sur les rangées, notre notion de sorte est en fait la même que celle des enregistrements extensibles : la sorte d'une rangée $\varphi$ est un ensemble d'étiquettes (dans notre cas, de variables de récursion) qui intuitivement sont les champs qui ne peuvent pas apparaître dans $\varphi$, car il sont déjà présents dans la partie gauche d'une plus grande rangée dont $\varphi$ est la queue. Il est facile de voir que toutes les règles listées dans cette section sont bien sortées. Cela nous permet d'affirmer que le système proposé ici vérifie la propriété du type principal, et peut être implémenté par un algorithme d'unification.

Dans la rangée $\{x:\pi, \rho\}$, la variable $\rho$ représente "l'état de toutes les variables de récursion sauf $x$". Ceci était très pratique dans le cadre des enregistrements extensibles, et cela l'est ici aussi : Dans l'exemple

```
rec x . (rec y . let f z = x + z in ... (f y) ...)
```

la fonction `f` aura le type $\{x:p, \rho\} \to \{x:p, \rho\}$, indiquant que l'argument de `f` est autorisé à dépendre de `x`, que dans tous les cas son résultat dépend de `x`, et que pour le reste, les dépendances du résultat de `f` sont les mêmes que celles de son argument. Le système peut ainsi inférer que `(f y)` dépend de `x` et de `y`.

La grammaire des types est comme suit :

types :

| | | |
|---|---|---|
| $\tau ::=$ | $\alpha$ | variable de type |
| | $\mid \{\varphi\}$ | information de dépendance |
| | $\mid \tau_1 * \tau_2$ | paire |
| | $\mid \tau_1 \to \tau_2$ | type flèche |

schémas de type :

$\sigma ::= \quad \tau \mid \forall\alpha, \sigma \mid \forall\rho, \sigma \mid \forall\delta, \sigma$

Le lieur $\forall$ joue le rôle habituel dans les schémas de types. Il y a aussi la relation classique "être une instance de" entre types et schémas de types, notée $\tau < \sigma$, et définie inductivement par :

$\tau < \tau$

Si $\tau < \sigma$, alors pour tout $\tau'$ et $\alpha$, $\tau[^{\tau'}/_{\alpha}] < \forall\alpha, \sigma$

Si $\tau < \sigma$, alors pour n'importe quelle rangée $\varphi$ et variable de rangée $\rho$ de la même sorte, $\tau[^{\varphi}/_{\rho}] < \forall\rho, \sigma$

Si $\tau < \sigma$, alors pour tout $\pi$ et $\delta$, $\tau[^{\pi}/_{\delta}] < \forall\delta, \sigma$

Nous supposerons que toutes les variables liées par un `rec` du programme analysé sont distinctes. Dans le cadre d'une sémantique opérationnelle (définie par une notion de réduction), cette hypothèse serait ennuyeuse car elle ne serait pas conservée par réduction. Mais la sémantique co-itérative pour laquelle nous allons prouver la correction de l'analyse de causalité est donnée par traduction dans un langage fonctionnel, qui conserve cette propriété.

Les jugements de typage sont de la forme $\Gamma \vdash t : \tau$. L'environnement de typage $\Gamma$ associe des schémas de types aux variables de programme.

L'originalité de ce système réside dans les types donnés aux opérateurs `rec` et `pre`. Cela ne devrait pas être trop surprenant puisque `rec` est la construction qui rend cette analyse nécessaire, alors que `pre` est, par contre, la construction qui rend les récursions bien fondées.

`pre` $c$ reçoit le schéma de type $\forall\rho, \rho', \{\rho\} \to \{\rho'\}$ :

$$\frac{\Gamma \vdash t : \{\varphi\}}{\Gamma \vdash \texttt{pre}\ c\ t : \{\varphi'\}}$$

Dans cette règle, les variables dont $t$ dépend (les variables marquées $p$ dans $\varphi$) n'imposent pas de contraintes sur la rangée $\{\varphi'\}$ de `pre` $c$ $t$.

Cela peut sembler contre-intuitif : on s'attend à une règle s'énonçant "pre $c$ $t$ ne dépend d'aucune variable", alors que sa signification effective ressemble plus à "Les dépendances de pre $c$ $t$ peuvent être choisies arbitrairement".

Mais remarquons que si l'on choisit $\{\varphi'\}$ comme étant par exemple $\{x{:}a, y{:}p, \rho\}$, on ne fait que perdre (correctement) l'information que pre $c$ $t$ ne dépend pas de $y$.

Il est ainsi possible d'accepter un terme tel que plus $x$ (pre $0$ $x$) où l'addition point-à-point sur les flots plus a le type $\forall\rho, \{\rho\} \to \{\rho\} \to \{\rho\}$.

Dans ce cas la règle pour pre est instanciée avec des rangées identiques pour $\{\varphi\}$ et $\{\varphi'\}$. Le système infère alors que plus $x$ (pre $0$ $x$) a les même dépendances que $x$.

La règle pour la récursion unaire :

$$\frac{\Gamma, x : \{x{:}p, \varphi\} \vdash t : \{x{:}a, \varphi\}}{\Gamma \vdash \texttt{rec}\ \ x = t : \{x{:}\pi, \varphi\}}$$

La variable de récursion $x$ est placée dans l'environnement associée à la rangée $\{x : p, \varphi\}$. De cette manière, n'importe quelle expression $t$ dépendant de $x$ (comme $x$ lui-même, ou plus $x$ (pre $0$ $x$)) aura dans sa rangée un champ $x$ associé à $p$. Cette rangée ne sera pas unifiable avec celle attendue de la forme $\{x{:}a, \varphi\}$, et rec $x = t$ sera rejeté.

Une autre conclusion, peut-être plus intuitive, pour cette règle serait $\Gamma \vdash \texttt{rec}\ x = t : \{\varphi\}$. Malheureusement, cela enfreindrait les règles de sortage sur les rangées : $\varphi$ ne doit être utilisé qu'à la droite d'un champ $x : \pi$.

La récursion mutuelle utilise une rangée différente pour chacune des variables de récursion :

$$\frac{\begin{array}{c}\Gamma, x : \{x{:}p, y{:}\pi_1, \varphi_1\}, y : \{x{:}\pi_2, y{:}p, \varphi_2\} \vdash t_1 : \{x{:}a, y{:}\pi_1, \varphi_1\} \\ \Gamma, x : \{x{:}p, y{:}\pi_1, \varphi_1\}, y : \{x{:}\pi_2, y{:}p, \varphi_2\} \vdash t_2 : \{x{:}\pi_2, y{:}a, \varphi_2\}\end{array}}{\Gamma \vdash \texttt{rec}\ \ x = t_1\ \ \texttt{and}\ \ y = t_2 : \{x{:}\pi_3, y{:}\pi_4, \varphi_1\} * \{x{:}\pi_5, y{:}\pi_6, \varphi_2\}}$$

Cette règle est une généralisation de celle de la récursion unaire. $x$ (resp. $y$) est associé dans l'environnement à une rangée qui transporte l'information "dépend de $x$(resp. $y$)".

Bien sûr $t_1$ peut dépendre de $y$, comme dans l'exemple rec $x = y$ and $y =$ pre $0$ $y$. Il suffit d'instancier $\pi_1$ par $p$.

On a vu que, si $t_1$ dépend de $y$, on veut interdire que $t_2$ dépende de $x$. Voici comment ceci est réalisé : si $t_1$ dépend de $y$, $t_2$ est analysé dans un environnement dans lequel $x$ est associé à $\{x : p, y : p, \rho\}$. Dans ces circonstances, si $t_2$ dépend de $x$, il se retrouve aussi avec $y : p$ dans ses dépendances, et la seconde prémisse ne peut pas être satisfaite.

En effet, on peut vérifier qu'il n'y a pas de dérivation pour le terme rec $x = y$ and $y = x$.

La clé est dans le partage de $\pi_1$ and $\pi_2$ entre les deux prémisses.

On donne aux constantes le schéma de type $\forall\rho, \{\rho\}$ :

$$\frac{}{\Gamma \vdash c : \{\varphi\}}$$

Toutes les règles restantes sont identiques à celles du système de Hindley-Milner.

$$\frac{\tau < \sigma}{\Gamma, x : \sigma \vdash x : \tau} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : \tau_1 * \tau_2}$$

$$\frac{\Gamma \vdash t : \tau_1 * \tau_2}{\Gamma \vdash \texttt{fst}\ t : \tau_1} \qquad \frac{\Gamma \vdash t : \tau_1 * \tau_2}{\Gamma \vdash \texttt{snd}\ t : \tau_2}$$

$$\frac{\Gamma, x : \tau \vdash t : \tau'}{\Gamma \vdash \lambda x.t : \tau \to \tau'} \qquad \frac{\Gamma \vdash t_1 : \tau \to \tau' \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1\ t_2 : \tau'}$$

$$\frac{\Gamma \vdash t : \tau \quad \Gamma, x : \forall(\mathcal{FV}(\tau) - \mathcal{FV}(\Gamma)), \tau \vdash t' : \tau'}{\Gamma \vdash \texttt{let}\ x = t\ \texttt{in}\ t' : \tau'}$$

Les règles pour `pre` et `rec` utilisent des rangées $\{\varphi\}$ au lieu de types généraux $\tau$ : le second argument de l'opérateur `pre` ne peut donc être qu'un flot, et les seules définitions récursives possibles sont des définitions de flots. Étant donné qu'il est impossible de définir une paire récursivement dans le langage, l'opérateur `rec` doit gérer explicitement la récursion mutuelle, au lieu de laisser le programmeur l'encoder dans une récursion simple sur une paire, comme on le fait dans un langage fonctionnel traditionnel.

Ces restrictions ne sont pas aussi arbitraires qu'il peut sembler à première vue. Tout d'abord, nous n'avons pas de sémantique claire pour (`pre` $c$ $e$) où $e$ est une fonction. Ensuite, supposant que l'on ait trouvée une telle sémantique, et supposant que (`pre` $c$) ait le type $\forall\alpha\forall\alpha', \alpha \to \alpha'$ au lieu de $\forall\rho\forall\rho', \{\rho\} \to \{\rho'\}$, le système de type permettrait d'inférer que (`pre` ($\lambda x$.`pre` $0$ $x$) ($\lambda x.x$)) $y$ ne dépend pas de $y$, alors que (encore une fois, a condition que l'on sache donner du sens a ce programme) ce serait le cas après le premier instant.

Les fonctions définies récursivement posent un problème différent. Pour la récursion d'ordre supérieur, la bonne fondation de la récursion dépend de conditions sémantiques et ne peut pas être vérifiée par cette analyse de causalité. Bien que les fonctions récursives ne soient pas autorisées par ce système, il est possible, dans une implémentation effective, d'avoir un opérateur différent pour la récursion fonctionnelle, et une option du compilateur pour autoriser ou non son utilisation.

Exemples :

- Les flots constants
  Les flots constants n'avaient pas réellement besoin d'être intégrés dans le langage, car le flot constant $c$ peut être défini comme :
  `rec z = pre c z`
  Le schéma de type attribué à cette expression, $\forall\rho, \delta,\ \{z\!:\!\delta, \rho\}$, est équivalent au schéma de type des flots constants $\forall\rho, \{\rho\}$ sous notre hypothèse que toutes les variables liées d'un terme sont distinctes.
- Le combinateur de rebouclage :
  $\lambda$`f.rec x = f(x)` est causal, de schéma de type $\forall\rho, \delta,\ (\{x\!:\!p, \rho\} \to \{x\!:\!a, \rho\}) \to \{x\!:\!\delta, \rho\}$.
- Récursion mutuelle : l'exemple
  `rec x = y and y = pre 0 x`
  est accepté. Dans la règle de la récursion mutuelle, $\pi_1$ (information de présence de $y$ dans la rangée associée à $x$ dans l'environnement) est instanciée par $p$, et $\pi_2$, information de présence de $x$ dans la rangée de $y$, est instanciée par $a$.
- Récursion mutuelle d'ordre : la fonction
  $F \equiv \lambda$`f.`$\lambda$`g.rec x = g(y) and y = f(x)`
  est bien typée, et a le schéma de type :
  $\forall\rho_1, \rho_2, \delta_1, \delta_2, \delta_3, \delta_4, \delta_5, \delta_6, (\{x\!:\!\delta_2, y\!:\!p, \rho_2\} \to \{x\!:\!a, y\!:\!\delta_1, \rho_1\}) \to$
  $(\{x\!:\!p, y\!:\!\delta_1, \rho_1\} \to \{x\!:\!\delta_2, y\!:\!p, \rho_2\}) \to (\{x\!:\!\delta_3, y\!:\!\delta_4, \rho_1\} * \{x\!:\!\delta_5, y\!:\!\delta_6, \rho_2\})$

  De plus, l'application partielle de $F$ à une fonction stricte telle que l'identité donne une fonction qui

peut être appliquée à $\lambda x.\mathtt{pre}\ c\ x$ mais pas à une fonction stricte, alors que l'application partielle de $F$ à $\lambda x.\mathtt{pre}\ c\ x$ donne une fonction qui peut être appliquée à une fonction stricte.
– Défaut de polymorphisme : l'exemple
`rec x = fst` $\big((\lambda\mathtt{f.(f\ 0,f\ x))}(\lambda\mathtt{y.y})\big)$
est rejeté, bien qu'il soit causal (il définit le flot constant 0). La raison est que le type de `f` n'est pas généralisé.

# 4. Co-itération

Dans [4], les flots synchrones se voient donner une représentation concrète constituée d'un état initial et d'une fonction de transition retournant une valeur et un nouvel état quand appliquée à un état.

Dans cette section, nous présentons très brièvement cette représentation. Le langage cible est un langage fonctionnel évalué paresseusement.

Les flots sont représentés dans le langage cible par le triplet $(\llbracket t\rrbracket_i, \llbracket t\rrbracket_v, \llbracket t\rrbracket_s)$ où :
- $\llbracket t\rrbracket_i$ est l'état initial.
- $\llbracket t\rrbracket_v$ est une fonction retournant une valeur quand appliquée à l'état courant.
- $\llbracket t\rrbracket_s$ est une fonction retournant le nouvel état quand appliquée à l'état courant.

Par rapport à [4], nous avons ici séparé la fonction de transition en deux, l'une calculant la valeur courante et l'autre le nouvel état. Bien sûr les deux représentations sont équivalentes, mais celle-ci permet d'exprimer plus simplement la causalité des programmes.

Un flot constant ne nécessite pas de mémoire, donc son état est `Nil`. La représentation d'un flot constant est donc :

$$\begin{array}{rcl}\llbracket c\rrbracket_i & = & \mathtt{Nil}\\ \llbracket c\rrbracket_v & = & \lambda\mathtt{Nil}.c\\ \llbracket c\rrbracket_s & = & \lambda\mathtt{Nil}.\mathtt{Nil}\end{array}$$

`pre` est la construction qui nécessite une mémoire, mémorisant la valeur prise par le flot à l'instant précédent.

`pre` c t :

$$\begin{array}{rcl}\llbracket \mathtt{pre}\ c\ t\rrbracket_i & = & (c, \llbracket t\rrbracket_i)\\ \llbracket \mathtt{pre}\ c\ t\rrbracket_v & = & \lambda(v,s).v\\ \llbracket \mathtt{pre}\ c\ t\rrbracket_s & = & \lambda(v,s).(\llbracket t\rrbracket_v\ s, \llbracket t\rrbracket_s\ s)\end{array}$$

La traduction de la plupart des constructions du langage est évidente, par exemple, pour la paire et les projections :

$(t_1, t_2)$ :

$$\begin{array}{rcl}\llbracket (t_1,t_2)\rrbracket_i & = & (\llbracket t_1\rrbracket_i, \llbracket t_2\rrbracket_i)\\ \llbracket (t_1,t_2)\rrbracket_v & = & \lambda(s_1,s_2).(\llbracket t_1\rrbracket_v\ s_1, \llbracket t_2\rrbracket_v\ s_2)\\ \llbracket (t_1,t_2)\rrbracket_s & = & \lambda(s_1,s_2).(\llbracket t_1\rrbracket_s\ s_1, \llbracket t_2\rrbracket_s\ s_2)\end{array}$$

`fst` $t$ :

$$\begin{array}{rcl}\llbracket \mathtt{fst}\ t\rrbracket_i & = & \llbracket t\rrbracket_i\\ \llbracket \mathtt{fst}\ t\rrbracket_v & = & \lambda s.\mathtt{fst}\ (\llbracket t\rrbracket_v\ s)\\ \llbracket \mathtt{fst}\ t\rrbracket_s & = & \llbracket t\rrbracket_s\end{array}$$

Nous ne donnerons pas ici les traductions de l'abstraction et de l'application, qui sont assez techniques. On pourra se reporter à [4] et à la version longue de cet article.

Récursion :

$$
\begin{array}{lcl}
[\![\texttt{rec}\ x = t_1\ \texttt{and}\ y = t_2]\!]_i & = & ([\![t_1]\!]_i, [\![t_2]\!]_i)[^{\texttt{Nil}}/_{[\![x]\!]_i}][^{\texttt{Nil}}/_{[\![y]\!]_i}] \\
[\![\texttt{rec}\ x = t_1\ \texttt{and}\ y = t_2]\!]_v & = & \lambda(s_1, s_2). \\
 & & \texttt{let rec}\ (x_v, y_v) = \\
 & & \qquad \big(([\![t_1]\!]_v\ s_1, [\![t_2]\!]_v\ s_2)[^{\lambda\texttt{Nil}.x_v, \lambda\texttt{Nil}.y_v}/_{[\![x]\!]_v, [\![y]\!]_v}]\big) \\
 & & \text{in}\ (x_v, y_v) \\
[\![\texttt{rec}\ x = t_1\ \texttt{and}\ y = t_2]\!]_s & = & \lambda(s_1, s_2). \\
 & & \texttt{let rec}\ (x_v, y_v) = \\
 & & \qquad \big(([\![t_1]\!]_v\ s_1, [\![t_2]\!]_v\ s_2)[^{\lambda\texttt{Nil}.x_v, \lambda\texttt{Nil}.y_v}/_{[\![x]\!]_v, [\![y]\!]_v}]\big)\ \text{in} \\
 & & \big(([\![t_1]\!]_s\ s_1, [\![t_2]\!]_s\ s_2)[^{\lambda\texttt{Nil}.x_v, \lambda\texttt{Nil}.y_v, \lambda\texttt{Nil}.\texttt{Nil}, \lambda\texttt{Nil}.\texttt{Nil}}/_{[\![x]\!]_v, [\![y]\!]_v, [\![x]\!]_s, [\![y]\!]_s}]\big)
\end{array}
$$

Les définitions de $[\![t]\!]_i$, $[\![t]\!]_v$ et $[\![t]\!]_s$ utilisent inductivement les traductions des sous-termes de $t$. Dans le cas de la traduction de $\texttt{rec}\ x = t_1\ \texttt{and}\ y = t_2$, $x$ et $y$ sont des variables libres de $t_1$ et $t_2$, et il nous faut quelque chose à substituer à $[\![x]\!]_i$, $[\![x]\!]_v$,...dans $[\![t_1]\!]$ et $[\![t_2]\!]$.

En particulier, nous remplaçons $[\![x]\!]_v$ par la fonction renvoyant la valeur en train d'être calculée récursivement.

# 5. Preuve de Correction

Dans cette section, nous présentons un aperçu de la preuve de correction pour le système donné en section 3.

Ici, nous considérons la langage cible comme un $\lambda$-calcul équipé de la $\beta$-réduction forte. Nous utiliserons en particulier beaucoup la propriété de Church-Rosser.

Nous utiliserons aussi la propriété que les termes du langage cible qui n'utilisent pas l'opérateur de récursion ont une $\beta$-forme normale. Ceci provient du fait que tous les termes que nous manipulons peuvent être typés.

La propriété que nous souhaitons prouver est essentiellement que si le terme $t$ a le type $\{x : a, \ldots\}$, alors $t$ "ne dépend pas" de $x$.

Pour un flot $t$, "ne pas dépendre" de $x$ signifie que $[\![t]\!]_v\ s$, la valeur instantanée de $t$, peut être calculée sans avoir besoin de la valeur instantanée de $x$. Ou, encore plus concrètement, $x$ n'apparaît pas dans la forme normale de $[\![t]\!]_v\ s$ [1].

Toujours informellement, nous définissons l'ensemble des termes $t$ qui ne dépendent pas de $x$ comme l'interprétation du type $\{x : a, \ldots\}$. Évidemment, avec notre définition de "ne pas dépendre", l'appartenance d'un terme $t$ à cette interprétation ne dépend que de la forme normale de $[\![t]\!]_v\ s$.

La présence de polymorphisme dans le système nous oblige à définir une notion de "candidat", comme dans [8]. Un candidat est un ensemble de termes qui est de la bonne forme pour être l'interprétation d'un type. Notre notion de candidat se trouvera assez naturellement être "un ensemble auquel l'appartenance d'un terme $t$ ne dépend que d'une propriété de la forme normale de $[\![t]\!]_v\ s$".

Définitions :
- Un candidat $C$ est un ensemble de termes tel que la propriété "$t \in C$" peut être exprimée sous la forme "$[\![t]\!]_v$ et $[\![t]\!]_s$ n'utilisent pas de récursion et, pour tout état $s$, $t_1$ étant la forme normale de $[\![t]\!]_v\ s$, $\mathcal{R}(t_1)$", pour un certain prédicat arbitraire $\mathcal{R}$.
- Un candidat de rangée $Q$ est un ensemble de termes $t$ tel que la propriété "$t \in Q$" peut être exprimée sous la forme :

  "$[\![t]\!]_v$ et $[\![t]\!]_s$ n'utilisent pas de récursion et
  Pour tout état $s$, si $t_1$ est la forme normale de $[\![t]\!]_v\ s$, alors $\mathcal{R}(t_1)$"

[1] Dans cette explication informelle, nous faisons référence à "la" forme normale de $[\![t]\!]_v\ s$. En fait, l'existence d'une forme normale est démontrée en même temps que la correction de l'analyse de causalité. En d'autres termes, nous prouvons que la récursion causale sur les flots peut être traduite sans récursion dans le langage cible.

où $\mathcal{R}$ est un prédicat qui peut être exprimé sous la forme :

$$\mathcal{R}(t) \equiv \bigwedge_{x \in X} (x \text{ n'apparaît pas dans } t)$$

Ces définitions préliminaires appellent quelques remarques :
- La définition d'un candidat est plus simple que celle, par exemple, de la preuve de normalisation forte du système F[8]. Ceci est dû au fait que la propriété que nous cherchons à démontrer est elle-même plus simple.
- D'un autre coté, notre système comporte des rangées, qui nécessitent leur propre notion de candidat. Il fallait s'y attendre, à la lumière du fait que, comme nous l'avons remarqué, les règles pour `pre` et `rec` ne peuvent être instanciées qu'avec des rangées, et deviennent incorrectes quand instanciées avec des types arbitraires.
- On peut remarquer que la définition des interprétations de rangées est compatible avec la théorie équationnelle. En d'autres termes, n'importe laquelle des représentations syntaxiques d'une même rangée donne la même interprétation.
- Un candidat de rangée est un candidat. En fait, la phrase précédente est l'image dans le monde des interprétations de la phrase "une rangée est un type".

Nous nous voulons maintenant associer des candidats aux variables de types, et des candidats de rangées aux variables de rangées et de présence. Nous appelons une telle association une valuation et nous la noterons $V$.

Étant donnée une valuation $V$, nous noterons $V \sqcup \alpha \mapsto C$ la valuation qui associe $\alpha$ à $C$, et toute autre variable $\beta$ à $V(\beta)$.

Quand nous ferons référence à une valuation $V$, nous prendrons toujours implicitement l'hypothèse que les ensembles de termes associés aux variables de types (resp. aux variables de rangée et de présence) sont des candidats (resp. des candidats de rangée).

Définition de l'interprétation $\mathcal{P}_V$ d'un type (relative à une valuation $V$) :
- $t \in \mathcal{P}_V(\{x\!:\!a, \varphi\})$ si $[\![t]\!]_v$ et $[\![t]\!]_s$ n'utilisent pas de récursion, et la forme normale de son application à un état ne comporte pas $x$, et si $t \in \mathcal{P}_V(\{\varphi\})$.
- $\mathcal{P}_V(\{x\!:\!p, \varphi\}) = \mathcal{P}_V(\{\varphi\})$.
- $t \in \mathcal{P}_V(\{x\!:\!\delta, \varphi\})$ si $t \in V(\delta)$ et $t \in \mathcal{P}_V(\{\varphi\})$
- $\mathcal{P}_V(\rho) = V(\rho)$
- $\mathcal{P}_V(\tau_1 * \tau_2)$ est défini comme l'ensemble des termes $t$ tels que `fst` $t \in \mathcal{P}_V(\tau_1)$, et `snd` $t \in \mathcal{P}_V(\tau_2)$
- $\mathcal{P}_V(\tau_1 \to \tau_2)$ : l'ensemble des termes $t$ tels que $[\![t]\!]_v$ et $[\![t]\!]_s$ n'utilisent pas de récursion, et tels que pour tout $t_1 \in \mathcal{P}_V(\tau_1)$, $(t\ t_1) \in \mathcal{P}_V(\tau_2)$
- $\mathcal{P}_V(\alpha) = V(\alpha)$

Il nous faut généraliser notre interprétation à des environnements non vides. Classiquement, nous exigeons du terme qu'il satisfasse la propriété d'interprétation sous toutes les substitutions possibles de termes (appartenant à l'interprétation du type qui leur est assigné dans l'environnement) aux variables libres.

Plus formellement, nous commençons par étendre la notion d'interprétation aux schémas de types :

$t \in \mathcal{P}_V(\forall \alpha, \sigma)$ si pour tout candidat $C$, $t \in \mathcal{P}_{V \sqcup \alpha \mapsto C}(\sigma)$.

$t \in \mathcal{P}_V(\forall \rho, \sigma)$ si pour tout candidat de rangée $Q$, $t \in \mathcal{P}_{V \sqcup \rho \mapsto Q}(\sigma)$.

$t \in \mathcal{P}_V(\forall \delta, \sigma)$ si pour tout candidat de rangée $Q$, $t \in \mathcal{P}_{V \sqcup \delta \mapsto Q}(\sigma)$.

Énonçons maintenant quelques propriétés relatives aux interprétations et aux candidats :

**Propriété 1** *Pour toute valuation $V$, $\mathcal{P}_V(\{\varphi\})$ est un candidat de rangée.*

Preuve par induction sur $\varphi$.

**Propriété 2** *Pour toute valuation $V$, $\mathcal{P}_V(\tau)$ est un candidat.*

Preuve par induction sur $\tau$.

Comme on pouvait s'y attendre, les interprétations des schémas de types sont incluses dans les interprétations de leurs instances. Cela peut être énoncé :

**Propriété 3** *Pour toute valuation $V$, si $t \in \mathcal{P}_V(\sigma)$ alors pour tout $\tau < \sigma$, $t \in \mathcal{P}_V(\tau)$.*

La preuve se fait par induction sur $\sigma$.

**Théorème 1** *Si $\Gamma \vdash t : \tau$, alors pour toute valuation $V$, pour toute substitution $S$ adaptée à $\Gamma$ (sous $V$) [2], $S(t) \in \mathcal{P}_V(\tau)$.*

La partie principale de la preuve est une induction sur l'arbre d'inférence de l'analyse de causalité. Nous ne présentons que quelques cas intéressants.

1. Le cas de `pre` est intéressant, même s'il n'est pas difficile :
   $\mathcal{P}_V(\{\varphi'\})$ est un candidat de rangée. Puisque $[\![\texttt{pre}\ c\ t]\!]_v = \lambda(v,s).v$, évidemment `pre` $c$ $t$ appartient à cet ensemble.
2. La règle de `rec` :
   Fixons $V$ et $S$.
   L'interprétation $\mathcal{P}_V(\{y:\pi_1,\varphi_1\})$ est un candidat de rangée. Donc, il existe $X$ un ensemble de variables de récursion tel que $t$ appartient à $\mathcal{P}_V(\{y:\pi_1,\varphi_1\})$ si et seulement si $t$ ne dépend d'aucune des variables de $X$.
   On peut distinguer deux cas :

   (a) $y \in X$
   Dans ce cas, l'hypothèse d'induction pour la première prémisse donne que ni $x$ ni $y$ n'apparaissent dans la forme normale de $[\![(S \sqcup x \mapsto x \sqcup y \mapsto y)(t_1)]\!]_v\ s$ (i.e. $[\![S(t_1)]\!]_v\ s$).
   Cela veut dire que le code pour la fonction de valeur $[\![\texttt{rec}\ x = t_1\ \texttt{and}\ y = t_2]\!]_v$ aurait pu être simplement être implémenté par [3] :

$$\begin{array}{l}\lambda(s_1,s_2).\\ \ \texttt{let}\ x_v = [\![t_1]\!]_v\ s_1\ \texttt{in}\\ \ \texttt{let}\ y_v = [\![t_2]\!]_v\ s_2\ \texttt{in}\\ \ (x_v,y_v)\end{array}$$

   (b) $y \notin X$
   Propriété : Pour tout $s$, $x$ n'apparaît pas dans la forme normale de $[\![t_2]\!]_v\ s$.
   En effet, supposant qu'il en soit autrement, nous pouvons substituer à $x$ un terme qui dépend de $y$ (par exemple $y$ lui-même) puisqu'il appartient à l'interprétation de $\{x:p,\varphi_1\}$.
   L'application de l'hypothèse d'induction pour la seconde prémisse avec $(S \sqcup x \mapsto y \sqcup y \mapsto y)(t_2)$ montre que si $x$ apparaissait dans $[\![t_2]\!]_v\ s$, alors $t_2$ ne pourrait avoir le type $\{y:a,\varphi_2\}$.
   en conséquence, la fonction de valeur de `rec` $x = t_1$ `and` $y = t_2$ est équivalente à :

[2] C'est-à-dire, pour toute substitution $S$ associant à chaque variable $x$ de $\Gamma$ un terme $t$ tel que $t \in \mathcal{P}_V(\Gamma(x))$

[3] En fait, nous utilisons cette notation pour laisser croire au lecteur qu'il n'y a plus de définition de valeurs récursives, et qu'un langage fonctionnel strict serait suffisant pour implémenter cette récursion. Nous trichons : nous avons seulement prouvé que $x$ et $y$ n'apparaissent pas dans la forme normale de $[\![t_1]\!]_v\ s_1$, pas qu'ils n'apparaissent pas dans $[\![t_1]\!]_v\ s_1$. Merci de nous pardonner notre mauvaise foi.

$$\begin{array}{l}\lambda(s_1, s_2).\\ \quad \texttt{let}\ y_v = [\![t_2]\!]_v\ s_2\ \texttt{in}\\ \quad \texttt{let}\ x_v = [\![t_1]\!]_v\ s_1\ \texttt{in}\\ \quad (x_v, y_v)\end{array}$$

Dans les deux cas, l'opérateur de récursion `rec` peut être représenté dans récursion dans le langage cible, et cette représentation alternative vérifie clairement l'invariant d'induction.

Toutefois, il faut remarquer que, par exemple, au moment de générer le code de la fonction $\lambda f.\lambda g.\texttt{rec}\ x = f(y)\ \texttt{and}\ y = g(x)$, il n'est pas encore possible de décider quel cas s'appliquera. En fait, comme le schéma de type de cette fonction est polymorphe, les deux versions peuvent être nécessaires, si la fonction est appliquée plusieurs fois avec des instances de type différentes.

# 6. Comparaison à d'autres travaux

## 6.1. À propos de causalité

Il existe plusieurs systèmes dont le but est de vérifier la productivité de définitions récursives. Le plus proche de celui que nous venons de présenter est décrit dans [18] : il s'agit d'une analyse de causalité pour une extension de Signal avec des fonctions d'ordre supérieur. Bien que les règles pour `pre` dans les deux systèmes soient très similaires, la règle pour la récursion dans [18] nous semble être la règle classique (de prémisse $x : \tau \vdash e : \tau$). Les programmes non causaux doivent alors être éliminés par le biais de conditions externes. Dans le système que nous proposons, ces conditions sont exprimées à l'intérieur du système de types. C'est ce qui permet d'avoir la propriété du type principal. En corollaire, les types et les erreurs de typage sont plus faciles à afficher.

La manipulation de graphes de dépendances explicites (comme il est fait en Lustre ou dans [15]) est plus expressive que notre système mais n'est pas modulaire. Nous perdons une certaine information en encodant les graphes de dépendance dans des types (même avec l'expressivité que donnent les rangées) mais chaque partie de programme peut être rejetée, ou acceptée et recevoir un type, indépendamment de son contexte d'utilisation.

La plupart des autres analyses de causalité sont conçues pour des langages dans lesquels les flots peuvent être dé-construits (il y a un opérateur `tail`). Cela complique beaucoup le problème : la dépendance sur $x$ de l'expression `Cons 0 x` est seulement temporairement cachée, elle peut réapparaître si le flot est déconstruit.

Par conséquent, il est nécessaire de compter combien de `Cons` ont été produits dans chaque expression. Cela rend le recours au sous-typage pratiquement inévitable, si l'on veut accepter des expressions telles que `x + (Cons 0 x)` (+ étant l'addition point-à-point sur les flots). Un exemple d'un tel système se trouve dans [7].

Le système proposé par J.Hughes, L.Pareto et A.Sabry dans [11] est beaucoup plus expressif, mais aussi beaucoup plus complexe. Leur système permet d'exprimer des relations linéaires entre les tailles des flots, vérifiant ainsi en même temps la causalité et ce qui correspondrait à l'analyse d'horloges dans un langage synchrone. Ce système utilise le sous-typage.

On peut remarquer que dans le cas de la conception de circuits, notre analyse peut être considérée trop restrictive : des circuits comportant des cycles peuvent cependant être causaux, comme il est montré dans [12]. Le compilateur Esterel a une analyse de causalité[16] visant à accepter de tels programmes. D'autre langages pour la conception de circuits, cette fois utilisant les flots de HASKELL, sont LAVA[17] et HAWK[5].

### 6.2. À propos de flots

Dans cette partie, nous mettons en avant le fait que les flots synchrones ne sont pas seulement un cas particulier des flots paresseux, et que chaque approche a ses avantages et ses inconvénients.

Commençons par une analogie : pour programmer des listes, quand l'évaluation paresseuse n'est pas nécessaire, on a le choix entre SCHEME et ML. Le dernier peut être marginalement plus rapide que le premier, mais ce n'est pas toujours très significatif. La différence principale est que le dernier fait des analyses statiques, et rejette des programmes qui n'ont pas l'air correct vis-à-vis de ces analyses (nécessairement approximatives). Bien sûr, les avis sont divisés sur la question de savoir si cela aide effectivement à programmer ou le contraire.

| pour programmer des listes : | liberté | sécurité |
|---|---|---|
| | Scheme | ML |

Il y a aussi des techniques d'optimisation, parfois inspirées de systèmes de types à la ML, pour rendre SCHEME plus rapide en supprimant une partie des tests de types dynamiques(*soft-typing*). Mais ces optimisations transparentes : si l'analyse échoue, le programmeur a toujours raison et le compilateur se contente de produire un code non optimisé.
Une comparaison similaire peut être faite entre LUCID SYNCHRONE et HASKELL :

| pour programmer des flots : | liberté | sécurité |
|---|---|---|
| | Haskell | Lucid Synchrone |

De la même manière que les tests de types dynamiques peuvent parfois être supprimés dans le cas de SCHEME, les allocations de cellules `Cons` peuvent être supprimées grâce à la déforestation[19]. Exactement comme entre ML et SCHEME, il reste une différence de point de vue fondamentale concernant le devenir des programmes qui ne passent pas les analyses.

## 7. Conclusion et Directions

Nous avons présenté une analyse de causalité pour un langage de flots synchrones. De bonnes propriétés, telles que le typage principal et la facilité d'implémentation, sont héritées de l'utilisation de rangées plutôt que de sous-typage. Bien que la preuve de correction soit spécifique à la sémantique co-itérative de LUCID SYNCHRONE, l'analyse elle-même devrait se généraliser aux cas d'équations mutuellement récursives dans lesquelles il n'y a pas d'opération (telle que `tail`) transformant une expression gardée en expression non gardée.

Une analyse de causalité fine rend impossible d'ordonnancer complètement le code à la compilation. De plus, on peut s'attendre à ce que les résultats de l'analyse de causalité puissent être utilisés pour une optimisation dirigée par les types du code généré.

## Remerciements

Nous sommes reconnaissants à Michel Mauny et Didier Rémy pour leur aide avec les premières versions de cet article. François Pessaux avait utilisé un système de types avec rangées pour détecter statiquement les exceptions non rattrapées, et ses explications nous ont été d'un grand secours.

## Références

[1] E. A. Ashcroft and W. W. Wadge. Lucid : A non-procedural language with iteration. *Communications of the ACM*, 20,7 :519–526, 1977.

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

[3] P. Caspi and M. Pouzet. The Lucid Synchrone distribution. `http ://www-spi.lip6.fr/~pouzet/lucid-synchrone/`.

[4] P. Caspi and M. Pouzet. A co-iterative characterisation of synchronous stream functions. Technical Report 07, VERIMAG, October 1997. Workshop on Coalgebraic Methods in Computer Science (CMCS'98), Electronic Lecture Notes in Theoretical Computer Science, Portugal, Lisbon (28-29 March 1998)
`http ://www-spi.lip6.fr/~pouzet/personal_bib/rr97-07.ps.gz`.

[5] B. Cook, J. Launchbury, and J. Matthews. Specifying superscalar microprocessors in Hawk. In *1998 Workshop on Formal Techniques for Hardware*, 1998.

[6] A. Frey, G. Berry, P. Bertin, F. Bourdoncle, and J. Vuillemin. The Jazz distribution. `http ://www.cma.ensmp.fr/jazz/`.

[7] E. Giménez. Structural recursive definitions in type theory. In *ICALP'98*, LNCS series no. 1443, July 1998. `ftp ://ftp.inria.fr/INRIA/Projects/coq/Eduardo.Gimenez/RRICALP98.ps.gz`.

[8] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991. `http ://www-verimag.imag.fr/PEOPLE/Nicolas.Halbwachs/lustre :ieee.html`.

[10] 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.

[11] J. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *ACM Principles of Programming Languages*, St Petersburg, Florida, January 1996. `http ://www.cs.chalmers.se/~rjmh/Papers/popl-96.ps`.

[12] S. Malik. Analysis of cyclic combinational circuits. *IEEE Transactions on Computer Aided Design*, 13(7) :950–956, July 1994.

[13] D. Rémy. Extending ML type system with a sorted equational theory. Research Report 1766, Institut National de Recherche en Informatique et Automatique, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1992. `ftp ://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/eq-theory-on-types.ps.gz`.

[14] D. Rémy. Type inference for records in a natural extension of ML. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design.* MIT Press, 1993. `ftp ://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/taoop1.ps.gz`.

[15] J. Saraiva, D. Swiersrta, M. Kuiper, and M. Pennings. Strictification of lazy functions. Technical report, 1996. `http ://www.cs.ruu.nl/docs/research/publication/TechList2.html`.

[16] T. R. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *Proc. International Design and Testing Conference*, Paris, 1996. `ftp ://ftp-sop.inria.fr/meije/esterel/papers/ConstructiveAnalysis.ps`.

[17] S. Singh and M. Sheeran. Designing FPGA circuits in Lava. Technical report. `http ://www.dcs.gla.ac.uk/~satnam/lava/lava_intro.ps`.

[18] J.-P. Talpin and D. Nowak. A synchronous semantics of higher-order processes for modeling reconfigurable reactive systems. *Conference on Foundations of Software Technology and Theoretical Computer Science*, 1998. `http ://www.irisa.fr/prive/talpin/fsttcs98.ps.gz`.

[19] P. Wadler. Deforestation : transforming programs to eliminate trees. *Theoretical Computer Science, (Special issue of selected papers from 2'nd European Symposium on Programming)*, 73 :231–248, 1990. `http ://www.cs.bell-labs.com/~wadler/papers/deforest/deforest.ps.gz`.

[20] M. Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93 :1–15, 1991. Preliminary version appeared in *Proc. 4th IEEE Symposium on Logic in Computer Science* (1989), 92–97.
`ftp ://ftp.ccs.neu.edu/pub/people/wand/papers/ic-91.dvi`.