LUCID SYNCHRONE : une extension fonctionnelle de LUSTRE

Paul Caspi[1] & Marc Pouzet[2]

1 : Laboratoire VERIMAG, Centre Equation,
2 avenue de Vignate, 38610 Gières, France,
caspi@imag.fr

2 : Laboratoire LIP6, Equipe SPI,
4 place Jussieu, Universitéé Paris 6,
75252 Paris cedex 05, France,
Marc.Pouzet@lip6.fr

Résumé

Nous présentons ici les progrès réalisés dans la conception, la sémantique et l'implantation de LUCID SYNCHRONE, extension fonctionnelle de LUSTRE construite au dessus d'OBJECTIVE CAML, qui combine l'expressivité de la programmation fonctionnelle en flots paresseux avec l'efficacité due au synchronisme. Ces progrès sont fondés sur la caractérisation du synchronisme en terme de co-itération et sa généralisation à l'ordre supérieur.

1. Introduction

Nous avons déjà présenté dans des éditions précédentes [18, 5] l'historique et les motivations de ce travail. Rappelons-les ici brièvement : la programmation de flots synchrones à la LUSTRE et SIGNAL [7, 2] se trouve à la confluence de trois courants d'idées : des questions (anciennes) de sémantique : LUCID, réseaux de Kahn [1, 12], la transcription d'automatismes du continu à l'échantillonné, et la déforestation [22].

Or LUSTRE est un langage de premier ordre. Dans ce qui précède, seul le deuxième point explique cette restriction. L'idée de chercher à la lever correspond à deux types de questions : d'abord, le temps-réel et les automatismes peuvent-ils bénéficier des facilités de programmation procurées par l'ordre supérieur ? et d'autre part, la programmation en général peut-elle bénéficier de l'efficacité apportée par le synchronisme ?

Le langage LUCID SYNCHRONE a été conçu comme outil d'exploration de ces questions. Dans nos précédentes présentations, le langage était fondé sur trois principes : une sémantique à la HASKELL [10], une généralisation de la sémantique opérationnelle de LUSTRE à l'ordre supérieur, et une généralisation du calcul d'horloge de LUSTRE à l'ordre supérieur.

Les progrès réalisés sont de deux ordres : une maquette plus conséquente du langage est réalisée ; elle est disponible à l'adresse : `ftp://spi.lip6.fr` et cette maquette est fondée sur une sémantique par co-itération inspirée de techniques co-algèbriques [11] qui nous semble beaucoup plus claire que la précédente.

Nous décrirons donc le langage en insistant sur ces points nouveaux.

2. Lucid Synchrone et sa sémantique

2.1. Le langage de premier ordre

C'est un langage fonctionnel avec une syntaxe très classique, à ceci près que, comme en LUSTRE, tous les types de base sont implicitement étendus à des flots :

- `int` représente le type des flots d'entiers,,
- `1` représente le flot infini de valeurs 1,
- `+` additionne point à point ses deux flots d'entrée : on peut voir cela comme un circuit additionneur, de même que `&` peut être vu comme une porte `and`.

`c`	t	f	t	$\ldots$
`x`	x_0	x_1	x_2	$\ldots$
`y`	y_0	y_1	y_2	$\ldots$
`if c then x else y`	x_0	y_1	x_2	$\ldots$

De plus, il y a trois opérateurs spéciaux :

- `fby` est un délai unitaire :

`x`	x_0	x_1	x_2	$\ldots$
`y`	y_0	y_1	y_2	$\ldots$
`x fby y`	x_0	y_0	y_1	$\ldots$

  servant à reboucler des circuits, par exemple un intégrateur :

```
let sum x = s
    where rec s = x + (0 fby s);;
```

ou à construire un détecteur de front haut :

```
let edge c = c & not (false fby c);;
```

Lorsque ce texte est fourni au compilateur avec l'option `-i`, on obtient la réponse :

```
node sum :  int -> int
node sum :: 'a -> 'a
node edge :  bool -> bool
node edge :: 'a -> 'a
```

Pour chacune de ces fonctions, appelées `node` comme en LUSTRE, nous obtenons les inférences de type *et d'horloge* qui seront décrites aux sections 3 et 4.

- `when` est un échantillonneur qui permet à des processus rapides de communiquer avec des processus lents en extrayant des sous-flots de flots en fonction d'une condition, c'est à dire un flot booléen.

c	f	t	f	t	...
x	x_0	x_1	x_2	x_3	...
x when c		x_1		x_3	...

Ainsi, la même fonction `sum` peut maintenant être utilisée à un rythme plus lent en échantillonnant son flot d'entrée :

```
let sampled_sum x c = sum(x when c);;

node sampled_sum :  int -> bool -> int
node sampled_sum :: 'a -> (x__4 : 'a) -> 'a on x__4
```

- `merge` réciproquement permet à des processus rapides de communiquer avec des processus lents en mélangeant deux flots :

c	f	t	f	t	...
x		x_0		x_1	...
y	y_0		y_1		...
merge c x y	y_0	x_0	y_1	x_1	...

Ainsi, on peut définir un bloqueur (l'opérateur `current` de LUSTRE) qui maintient un flot entre deux échantillons successifs. Ici, `ydef` est une valeur par défaut utilisée lorsqu'aucun échantillon n'a encore été reçu :

```
let hold ydef c x = y
    where rec y =
    merge c x ((ydef fby y) when (not c));;

node hold :  'a -> bool -> 'a -> 'a
node hold :: 'a -> (x__8 : 'a) -> ('a on x__8) -> 'a
```

Cela autorise une forme limitée de sur-échantillonnage, c'est à dire de fonctions qui ont un rythme de sortie supérieur à leurs rythmes d'entrée [1] :

```
let rec half = false fby (not half);;

let over x = hold 1 half x;;

node half :  bool
node half :: 'a
node over :  int -> int
node over :: 'a on half -> 'a
```

Opérateurs dérivés On considère aussi deux opérateurs dérivés utiles :

- `->`, qui permet d'initialiser des expressions :

  ```
  x -> y = if (true fby false) then x else y ;;
  ```

- `pre` correspondant à un `fby` non initialisé :

  ```
  pre x  = dummy fby x ;;
  ```

  où `dummy` est n'importe quelle constante du type de `x`.

2.2. Extension d'ordre supérieur

Jusqu'ici, le langage est plus ou moins équivalent à LUSTRE. Montrons maintenant comment on l'étend à l'ordre supérieur en considérant les fonctions comme des objets de première classe.

Ainsi, on peut définir, à la place de `sum`, une fonction qui reboucle un circuit, c'est à dire qui itère une fonction sur un flot de valeurs :

[1] En ce sens, ce langage est strictement plus puissant que LUSTRE et peut se comparer à SIGNAL. Cela montre qu'il est possible d'obtenir le pouvoir d'expression de SIGNAL dans un cadre purement fonctionnel et avec un calcul d'horloge (cf. 3.5) ne nécessitant pas la résolution d'équations booléennes.

```
let it f yinit x = y
    where rec y = f x (yinit fby y);;

node it :  ('b -> 'a -> 'a) -> 'a -> 'b -> 'a
node it :: ('b -> 'a -> 'a) -> 'a -> 'b -> 'a
```

de sorte que `sum x = it (+) 0 x`

2.3. Extensions récursives

A coté de ces facilités fonctionnelles on peut aussi définir des fonctions récursives telles que :

- la fonction qui ré-initialise un intégrateur sur une condition :

  ```
  let whilenot c = w
      where rec w = true ->  (not c) & (pre w);;

  let rec reset_sum (c,x) =
    let w = whilenot c in
    merge w
          (sum (x when w))
          (reset_sum ((c,x) when (not w)));;

  node whilenot :  bool -> bool
  node whilenot :: 'a -> 'a
  node reset_sum :  bool * int  -> int
  node reset_sum :: 'a * 'a -> 'a
  ```

  Cette fonction utilise la récursion terminale : `w` est initialement vraie et devient fausse dès que `c` devient vrai. Ainsi, `sum x` s'exécute initialement mais, lorsque `c` devient vrai, un nouveau `reset_sum` commence, avec un `w` local vrai, tandis que l'ancien ne sert plus jamais[2].

- des fonctions non récursives terminales telles le fameux crible d'Eratosthene :

[2] Cela permet de mimer la primitive `loop` d'ESTEREL [3]. Cela montre aussi que le phénomène, dit de réincarnation des compilateurs ESTEREL est très naturel : au même instant, deux incarnations de la variable `w` sont présentes : celle de la fonction appelante, et celle de la fonction appelée.

```
let first x = v
    where rec v = x fby v;;

let rec sieve x =
    let filter = (x mod (first x))<> 0
    in  merge filter
              (sieve (x when filter))
              (true fby false);;

let rec nat = 2 fby (nat + 1);;

let primes = nat when sieve nat;;

node first :  'a -> 'a
node first :: 'a -> 'a
node sieve :  int -> bool
node sieve :: 'a -> 'a
node nat :  int
node nat :: 'a
node primes :  int
node primes :: 'a on sieve nat
```

2.4. Sémantique

Tout ce qui précède peut être facilement simulé dans un langage fonctionnel paresseux à la HASKELL [3], et cette simulation nous procure une sémantique dénotationnelle à peu de frais :

- les flots paresseux de type `a` sont définis par :

  ```
  data  Stream a = Pre a  Stream a
  ```

- la fonction `const` permet de transformer des constantes en flots infinis :

  ```
  const i = Pre i  (const i)
  ```

  Ainsi,

  ```
  true  = const Prelude.true
  ```

[3] Dans tout ce qui concerne les questions de sémantique, nous utilisons une syntaxe à la HASKELL. Cela nous permet de vérifier nos expressions sémantiques à l'aide d'un compilateur HASKELL. En particulier, nous associons systématiquement aux expressions les types fournis par le compilateur. Ceci dit, on peut regretter que la famille des langages fonctionnels se soit divisée en autant de syntaxes differentes.

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

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

  Par exemple,

  ```
  not x  = extend  (const Prelude.not) x
  x & y  = extend (extend (const Prelude.and) x) y
  ```

- `fby` est simplement le constructeur de flots :

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

- `when` permet d'extraire des sous-flots :

  ```
  (Pre x  xs) when (Pre true cs) = Pre x  (xs when cs)
  (Pre x  xs) when (Pre false cs) = xs when cs
  ```

- Réciproquement, `merge` permet de reconstruire des flots :

  ```
  merge(Pre true cs) (Pre x xs) y  = Pre x (merge cs xs y)
  merge(Pre false cs) x (Pre y ys) = Pre y (merge cs x ys)
  ```

HASKELL étant exécutable, ces quelques définitions pourraient servir non seulement pour fournir une sémantique opérationnelle, mais aussi pour exécuter des programmes. Cependant, du fait de la paresse, ces exécutions sont en général peu efficaces. La section suivante a pour objet de montrer comment on peut transformer ces définitions en expressions totalement déforestées, donc efficaces.

3. Une sémantique opérationnelle par co-itération

La co-itération est apparue récemment comme un concept central pour raisonner et prouver des propriétés de systèmes de flots [19, 15] et aussi pour les compiler et les exécuter. Son intérêt vient de ce qu'elle permet de manipuler des structures comme les flots infinis de façon stricte et efficace. C'est pourquoi elle est (implicitement) largement utilisée en flots de données synchrones. Notre propos est ici de montrer comment on peut l'étendre à l'ordre supérieur.

3.1. Co-itération

Considérons les définitions de la section 2.4. La co-itération permet de se débarrasser de la récursion en considérant des flots concrets :

```
type  F t s = (t , s)
data CStream t s = Co (s -> F t s) s
```

c'est à dire formés d'une fonction de transition de type `s -> F t s` associant à un état une valeur et un nouvel état, et un état initial de type `s` :

Ainsi, certain des exemples précédents peuvent être traduits en[4] :

- ```
  co_const v = Co (\s -> (v, Nil)) Nil
  co_const :: t -> CStream t (State s)
  ```

  où `Nil` est une constante indéfinie pouvant être ajoutée à un type :

  ```
  data State s = Nil | St s
  ```

- ```
  co_extend (Co f if) (Co e ie) =
  Co (\(sf,se) -> let (vf, sf') = f sf
                      (ve, se') = e se in
                  ((vf ve), (sf',se'))
        ) (if,ie)
  co_extend :: (CStream (t -> t') s') -> (CStream t s)
               -> CStream t' (s', s)
  ```

- ```
  co_Pre v (Co tx ix) = Co(\(s,sx) ->(s, (tx sx))) (v,sx)

  co_Pre :: t  -> (CStream t s) -> CStream t (t, s)
  ```

Ces flots concrets peuvent être associés aux flots initiaux grâce à la fonction `run` qui les déroule :

```
run (Co tx sx) = let (vx, sx') = tx sx
                 in Pre vx (run (Co tx sx'))
```

Remarques : Deux flots concrets ayant les mêmes déroulés sont dits bisimilaires. La bisimulation est une relation d'équivalence et nous ne la distinguons pas de l'égalité. Dans un cadre co-algébrique [11], la fonction `run` est l'unique homomorphisme appliquant la co-algèbre $tx : S \to T \times S$

[4] En HASKELL, l'abstraction $\lambda x.e$ se note `\x -> e` et une fonction currifiée $\lambda x, y.e$ se note `\x y-> e`
```

à la co-algèbre terminale de flots $hd, tl : T^{\mathbb{N}} \rightarrow T \times T^{\mathbb{N}}$ avec les définitions usuelles :

$$\begin{array}{rcl} hd(s) & = & s(0) \\ tl(s)(n) & = & s(n+1) \end{array}$$

de sorte que le diagramme suivant commute :

$$\begin{tikzcd} S \arrow[r, "run"] \arrow[d, "tx"] & T^{\mathbb{N}} \arrow[d, "{hd, tl}"] \\ T \times S \arrow[r, "id \times run"] & T \times T^{\mathbb{N}} \end{tikzcd}$$

3.2. Fonctions préservant les longueurs

3.2.1. Application synchrone

On peut remarquer, sur les exemples précédents, que, dans chaque cas, la fonction de transition des arguments est appliquée une et une seule fois dans la fonction de transition du résultat. Autrement dit, chaque déroulé du résultat n'utilise qu'un déroulé de chaque argument (c'est pourquoi on dit que ces fonctions préservent la longueur). Cela nous donne l'idée de définir un combinateur d'application qui force cette restriction :

```
co_apply (Co tf sf) (Co te se) =
  Co (\(st,sf,se) -> let (vf, sf') = tf sf
                         (ve, se') = te se
                         (v,  st') = vf ve st
                     in (v, (st',sf',se')))
     (Nil,sf,se)

co_apply ::
(CStream (t -> State s' -> F t' (State s')) sf) ->
         (CStream t s) -> CStream t' ((State s'), sf , s)
```

Un pas d'application d'une fonction synchrone consiste à exécuter un pas de la fonction, un pas de l'argument, et à appliquer la valeur fonction à la valeur argument et éventuellement à une valeur d'état interne, de façon à obtenir la valeur du résultat et le nouvel état interne. Lorsque la fonction n'a pas encore été appliquée l'état interne est indéfini. D'où la définition :

Définition 1 (Fonction synchrone) *Une fonction des flots dans les flots* `f : Stream  t -> Stream t'` *est synchrone s'il existe un flot* `f': Stream  t -> s' -> F t' s'` *tel que* `f = co_apply  f'`.

Théorème 1 (Morphismes synchrones) *Cette définition donne :*

i l'identité `id` *est synchrone et*

ii la composition de deux fonctions synchrones est synchrone.

Les preuves sont dans [6]

3.2.2. Abstraction synchrone

Ce qui précède invite à considérer l'abstraction correspondante. L'idée pour la construire est que la fonction de transition du résultat ne dépend que de la valeur courante de l'argument. Cela permet d'abstraire l'argument au flot constant ayant cette valeur `v` :

```
co_lambda f =
co_const (\v s -> let Co t i   = f (co_const v)
                      s1       = case s of
                                   Nil     -> i
                                   St s2   -> s2
                      (v', s') = t s1
                  in  (v', (St s')))

co_lambda ::
  ((CStream t (State s)) -> CStream t' s') ->
  CStream (t -> State s' -> F t' (State s')) (State sf)
```

Clairement, cette abstraction n'est correcte que si l'argument est synchrone, c'est à dire l'image d'un `co_apply`. D'où la preuve de correction :

Théorème 2 (Correction de l'abstraction) *Pour tout* `f'` *:*

```
co_apply (co_lambda (co_apply f'))  = co_apply f'
```

3.2.3. Récursion

Nous avons vu l'importance des définitions récursives par exemple pour reboucler des circuits :

```
let rec nat = 0 fby (nat +1);;
```

Peut-on éliminer cette récursion ? Clairement, puisqu'on peut écrire :

```
co_nat = Co (\s-> (s, s+1)) 0
```

Mais existe-t-il un combinateur fournissant cette compilation ? La réponse est partielle :

```
co_rec (Co t i) = Co (\(se,s) -> let (v, s')   = t s
                                     (ve, se') = v ve se
                                 in  (ve,  (se',s')))
                     (Nil,i)

co_rec :: (CStream (t -> (State s) -> F t (State s)) s') ->
           CStream t ((State s) , s')
```

On doit se souvenir ici qu'en HASKELL les `let` sont récursifs. La seule variable récursive ici, `ve`, est la composante valeur de la fonction de transition. Deux cas se présentent :

- il s'agit d'une expression de base : si `ve` dépend de lui-même on a une récursion infinie correspondant à une boucle de causalité. Sinon, l'évaluation réussit. Par exemple l'équation :

  ```
  let rec x = x + 1
  ```

  n'est pas causale car `x` dépend de lui-même. En revanche, lorsque ce n'est pas le cas, en particulier lorsque la récursion est gardée par un `pre` ou `fby`, l'évaluation de la paire `ve,se'` peut être séparée en deux évaluations non récursives. Le compilateur utilise ce fait pour produire du code non récursif comme celui du `nat`.

- l'expression est d'ordre supérieur (cas du crible) et la terminaison dépend de conditions sémantiques que le programmeur doit vérifier.

Lorsque ces conditions sont remplies, il reste à vérifier que ce combinateur est correct :

Théorème 3 (Correction de la récursion) *Pour tout* `f` *:*

```
co_apply  f (co_rec f) = co_rec f
```

3.2.4. Produits et objets terminaux

Il est facile de construire des produits et de montrer qu'il existe des objets terminaux (les singletons de flots constants). Donc, les flots équipés de morphismes synchrones forment une catégorie cartésienne fermée.

3.3. Fonctions ne préservant pas les longueurs

L'objectif est d'étendre ces résultats aux fonctions ne préservant pas les longueurs telles que la fonction `when`. Une idée serait de considérer le foncteur :

```
data F t s = P t s | S s
```

où `P` signifie "présent" et `S` "silencieux", un processus silencieux étant un processus qui ne met à jour que son état sans fournir de valeur. Par exemple une fonction de transition pour la fonction `when` pourrait être :

```
(Co tx ix) co_when (Co tc ic) =
          Co (\(sx,sc) -> let (vx, sx') = tx sx
                              (vc, sc') = tc sc
                          in  case vc of
                                true -> P vx (sf',se')
                                false-> S (sf',se'))
             (ix,ic)
```

Mais il est équivalent de considérer une valeur "absent" à ajouter à chaque type de valeur : en effet,

$$S + T \times S = (1 + T) \times S$$

On définit donc :

```
data Val t = E | V t
type F t s = (t , s)
```

et l'horloge d'un flot avec "absence" :

```
clock (Pre E s)     = Pre false (clock s)
clock (Pre (V v) s) = Pre true (clock s)
```

Remarquons que les horloges sont des flots ordinaires, sans "absence". Moyennant cela, nous pouvons réutiliser les résultats précédents (cf. 3.2)

3.3.1. Application

Cette représentation des flots pose le problème suivant : que faire lorsque une valeur d'argument est présente, tandis que l'autre ne l'est pas ; il y a au moins trois possibilités :

1) stocker la valeur dans une file d'attente jusqu'à l'obtention d'une valeur de l'autre argument,

2) engendrer une erreur à l'exécution,

3) ou rejeter cette situation statiquement.

Pour rester cohérent avec LUSTRE et l'école synchrone nous choisissons la troisième solution et écrivons :

```
extend(Pre E fs)   (Pre E es)    = Pre E      (extend fs es)
extend(Pre(V f) fs)(Pre(V e) es) = Pre(V(f e))(extend fs es)
```

Cette définition est satisfaisante à condition que les deux arguments aient la même horloge. Il reste donc à interdire les erreur dues à un filtrage incomplet et c'est le rôle du calcul d'horloge. Ce calcul sera présenté ultérieurement pour simplifier la présentation.

3.4. Fonctions primitives

Il convient d'étendre cela aux autres primitives :

- `when` : la définition suppose que les deux arguments aient la même horloge,

```
(Pre E es)    when(Pre E cs)         = Pre E (es when cs)
(Pre E es)    when(Pre (V false) cs) = Pre E (es when cs)
(Pre(V e) es) when(Pre (V true) cs)  = Pre(V e)(es when cs)
```

l'horloge du résultat dépend de la condition ; si `cl` est l'horloge des arguments, on dit que `cl on c` est celle du résultat avec :

```
(Pre  false cl) on(Pre E         cs) = Pre false(cl on cs)
(Pre  true  cl) on(Pre (V false) cs) = Pre false(cl on cs)
(Pre  true  cl) on(Pre (V true)  cs) = Pre true (cl on cs)
```

Conformément à la définition, cette horloge n'a pas de valeur silencieuse.

- `merge` :

```
merge (Pre E  cs) (Pre E xs) (Pre E ys) =
              Pre E (merge cs xs ys)
merge (Pre (V true)  cs) (Pre (V x) xs) (Pre E ys) =
              Pre (V x) (merge cs xs ys)
merge (Pre (V false) cs) (Pre E xs) (Pre (V y) ys) =
              Pre (V y) (merge cs xs ys)
```

Les conditions d'horloges vont d'elles-mêmes et seront explicitées à la section suivante.

Les autres primitives posent quelques problèmes :

- `const` : cet opérateur devient polymorphe en ce sens que le fait qu'il produise ou non dépend de son environnement. De ce fait, on doit lui adjoindre un paramètre d'horloge qui est synthétisé au cours du calcul d'horloge :

```
cl_const v (Pre true  cl) = Pre (V v) (cl_const v cl)
cl_const v (Pre false cl) = Pre E     (cl_const v cl)
```

- `fby` le problème du `fby` est que, bien que son premier argument ne serve qu'au premier instant, on doit exiger qu'il ait la même horloge que le second, sans quoi on a des risques de boucle de causalité dans les définitions récursives telles que :

```
let rec x = 0 fby (x + 1);;
```

On demande donc que les deux arguments aient la même horloge :

```
x fby y = fby1 true  x y

fby1 true  (Pre E xs)    (Pre E ys)    =
                         Pre E    (fby1 true  xs ys)
fby1 true  (Pre(V x) xs) (Pre(V y) ys) =
                         Pre(V x) (fby1 false xs ys)
fby1 false (Pre E xs)    (Pre E ys)    =
                         Pre E    (fby1 false xs ys)
fby1 false (Pre(V x) xs) (Pre(V y) ys) =
                         Pre(V y) (fby1 false xs ys)
```

Les définitions de l'application, abstraction et récursion restent inchangées.

3.5. Inférence d'horloge

Le calcul d'horloge ayant déjà été décrit ailleurs [18, 5], il sera simplement évoqué et nous donnerons ses règles sans les commenter à la table 1. L'idée est de fournir des conditions vérifiables statiquement pour que les programmes ne connaissent pas d'échec au filtrage. La version présentée ici généralise celle de LUSTRE, en ce sens qu'il y a inférence d'horloge au lieu d'une simple vérification et que, comme nous l'avons vu, cela autorise un certain sur-échantillonnage. D'autre par, il supporte la pleine fonctionnalité. Ceci est obtenu en le considérant comme un système de typage classique à la Damas–Milner [13] avec une modification de la règle de la récursion due à [16]. L'autre modification vient du fait que les expressions d'horloges contiennent des expressions booléennes et qu'il s'agit donc de types dépendants. Cette difficulté a été résolue en restreignant l'unification sur ces expressions à la simple égalité syntaxique. Nous pouvons alors démontrer que ce calcul garantit l'exécution sans échec au filtrage. Plus précisément, nous définissons, par induction sur la structure des expressions d'horloge les classes d'équivalence $C(\sigma)$

- $ce \in C(\alpha)$, si
  - ce s'évalue en un flot fini ou infini sans échec au filtrage et
  - pour tout $ce' \in C(\alpha)$, et pour tout n pour lequel ce et ce' sont définis, $ce(n) = \mathtt{E}$ ssi $ce'(n) = \mathtt{E}$ et $ce(n) = \mathtt{V}\,(v)$ ssi il existe v' tel que $ce'(n) = \mathtt{V}\,(v')$
- $ce \in C(cl \ \mathtt{on}\ c)$ ssi
  - ce et c s'évaluent en flots finis ou infinis sans échec au filtrage et
  - pour tout n, tels que ce et c sont définis, $ce(n) = \mathtt{E}$ ssi $c(n) = \mathtt{E}$ ou $c(n) = (\mathtt{V}\,(false))$
- $cf \in C(cl \xrightarrow{x} cl')$ ssi $\forall\ ce \in C(cl)$, $cf\ ce$ est dans $C(cl'[ce/x])$
- $ce \in C(\forall\alpha_1, \ldots, \alpha_n.cl)$ ssi pour tout $cl_1, \ldots cl_n$ dont les variables libres diffèrent de celles de cl, $ce \in cl[cl_1/\alpha_1, \ldots cl_n/\alpha_n]$

Etant donné une liste d'hypothèses sur les horloges de variables distinctes, on définit l'interprétation (surchargée) :

$$\begin{array}{rcl} I(\emptyset) & = & true \\ I(H, x : \sigma) & = & I(H)\ \&\ x \in C(\sigma) \\ I(H \vdash e : \sigma) & = & I(H) \Rightarrow e \in C(\sigma) \end{array}$$

Cela nous permet d'énoncer le théorème :

Théorème 4 (consistance) *Pour tout H, e, σ, si $H \vdash e : \sigma$ est prouvable, alors $I(H \vdash e : \sigma)$ est vérifiée*

$$\textsc{const} \quad H \vdash \texttt{const}\ v\ : cl$$

$$\textsc{taut} \quad H, x : \sigma \vdash x : \sigma$$

$$\textsc{fby} \quad H \vdash \texttt{fby} : \forall\alpha.\alpha \to \alpha \to \alpha$$

$$\textsc{extend} \quad H \vdash \texttt{extend} : \forall\alpha.\alpha \to \alpha \to \alpha$$

$$\textsc{merge} \quad H \vdash \texttt{merge} : \forall\alpha.\alpha \xrightarrow{x} \alpha\ \ \texttt{on}\ x \to \alpha\ \ \texttt{on not}\ x \to \alpha$$

$$\textsc{when} \quad H \vdash \texttt{when} : \forall\alpha.\alpha \to \alpha \xrightarrow{x} \alpha\ \ \texttt{on}\ x$$

$$\textsc{abst} \quad \frac{H, x\ : cl \vdash e\ : cl' \quad x \notin fv(H)}{H \vdash \lambda x.e\ : cl \xrightarrow{x} cl'}$$

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

$$\textsc{rec} \quad \frac{H, x\ : \sigma \vdash e\ : \sigma \quad x \notin fv(H)}{H \vdash \texttt{rec}\ \ x.e\ : \sigma}$$

$$\textsc{let} \quad \frac{H \vdash e_1\ : \sigma_1 \quad x \notin fv(H) \quad H, x : \sigma_1 \vdash e_2 : cl_2}{H \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : cl_2[e_1/x]}$$

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

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

Tableau 1 : Le calcul d'horloge

4. Notes d'implantation

Le langage est bâti au dessus de OBJECTIVE CAML, dont il emprunte la syntaxe et l'environnement d'exécution. Un mécanisme d'importation permet d'acquérir des types et des expressions de OBJECTIVE CAML, qui sont implicitement étendues à des flots. Le même mécanisme est utilisé pour les communications entre modules LUCID SYNCHRONE.

Un programme est formé de déclarations qui peuvent être mutuellement récursives et arbitrairement imbriquées.

Le filtrage se limite pour le moment aux tuples.

Un mode temps-réel est fourni dans lequel le compilateur vérifie les conditions d'emploi en temps-réel (temps de transition et mémoire bornées).

La compilation est divisée en deux étapes : la phase d'analyse statique s'occupe du typage, calcul d'horloge, analyse de causalité et d'initialisation. La génération de code produit une fonction de transition écrit en OBJECTIVE CAML. Celle-ci peut alors être liée à d'autres modules OBJECTIVE CAML puis exécutée. En ce sens le compilateur agit comme un pre-processeur.

5. Travaux similaires

5.1. Systèmes réactif et synchronisme

Le concept de synchronisme a émergé assez naturellement dans le domaine des systèmes réactifs [8] comme une façon de caractériser les processus progressant de façon coordonnée [14]. Ce concept s'étend aux flots pour les raisons suivantes :

- Les systèmes réactifs ont des comportement infinis bien pris en compte par les notions de co-algèbre [11]. Cependant, il ne semble pas que les notions de morphismes synchrones aient été identifiés à ce niveau.

- La programmation par flots de données a fait partie, dès le début, du courant général de la programmation synchrone, ainsi que l'attestent les langages LUSTRE [7] et SIGNAL [2]. Mais ces langages ne comprennent pas de traits d'ordre supérieur[5].

[5]Récemment est apparue une sémantique co-inductive de SIGNAL [17]. Elle s'apparente donc à celle présentée ici (cf 2.4) dont la première publication [18] date d'au moins deux ans.

Boussinot [4] a défini un schéma d'exécution de réseaux de processus consistant à exécuter un pas de chaque processus à chaque instant, et cela produit une sémantique opérationnelle proche de la notre. Cependant il n'y a pas de calcul d'horloge et de compilation des communications.

Dans [20], Pucella propose une extension de Standard ML avec des traits réactifs et comprenant de l'ordre supérieur. Il diffère de notre approche car il se fonde sur un style de programmation à la ESTEREL, donc impératif, ce qui rend la comparaison difficile.

5.2. Synchronisme et déforestation

Nous avons déjà évoqué [5] le lien étroit entre l'approche synchrone et la déforestation de Wadler [21, 22]. Cependant, il ne semble pas que les idées de déforestation, même récentes [9] couvrent entièrement notre approche.

6. Conclusion

Nous avons donc montré ici les progrès accomplis dans notre extension fonctionnelle de LUSTRE, progrès obtenus essentiellement grâce à l'extension, à l'ordre supérieur du concept de co-itération qui nous a permis d'obtenir une bonne et systématique méthode de compilation.

Le langage est cependant encore très jeune et bien des progrès restent à accomplir, tant pratiques (nombre d'optimisations de compilation sont encore possibles) que théoriques. Parmi ces derniers, on peut citer l'obtention d'une caractérisation de la récursion terminale, nécessaire pour bien délimiter les programmes réactifs, et l'introduction de définition de types de données dans le langage, qui pose des problèmes de cohabitation entre algèbres et co-algèbres. De même, étant fondés sur OBJECTIVE CAML, il serait intéressant d'aborder la question des objets.

On peut aussi penser que cette théorie du synchronisme sera utile dans d'autres domaines comme par exemple les relations avec les formalismes de preuve : il est frappant de constater que les exemples de preuves de systèmes à flots sont, semble-t-il, toujours synchrones [15, 19].

Remerciements Thérèse Hardin du LIP6, Catherine Oriat du LSR/Imag et Alberto Pardo de l'Instituto de Computación de Montevideo ont, par leurs suggestions, contribué à ce travail.

Références

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

[2] 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.

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

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

[5] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *Int. Conf. on Functional Programming.* ACM SIGPLAN, Philadelphia May 1996.

[6] P. Caspi and M. Pouzet. A co-iterative characterization of synchronous stream functions. Research Report 97-07, Verimag, October 1997. Available at `http ://www-spi.lip6.fr/~pouzet`.

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

[8] D. Harel and A. Pnueli. On the development of reactive systems. In *Logic and Models of Concurrent Systems*, volume 13 of NATO *ASI Series*, pages 477–498. Springer Verlag, 1985.

[9] Z. Hu, H. Iwasaki, and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. In *Int. Conf. on Functional Programming, Philadelphia.* ACM SIGPLAN, May 1996.

[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] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, (62) :222–259, 1997.

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

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

[14] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267–310, July 1983.

[15] P.S. Miner and S.D. Johnson. Verification of an optimized fault-tolerant clock synchronization circuit. In Mary Sheeran and Satnam Singh, editors, *Designing Correct Circuits*, Electronic Workshops in Computing, Bastad, Sweden, September 1996. Springer-Verlag.

[16] A. Mycroft. Polymorphic type schemes. In *Proceedings of the Sixth International Symposium on Programming, Toulouse*, volume 167 of *Lecture Notes in Computer Science*. Springer, April 1984.

[17] D. Nowak, J.R. Beauvais, and J.P. Talpin. Co-inductive axiomatization of a synchronous language. In *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics*, volume 1479 of *LNCS*, pages 387–399. Springer Verlag, October 1998. http://www.irisa.fr/prive/nowak/publis/tphols98.ps.gz.

[18] P. Caspi et M. Pouzet. Réseaux de Kahn synchrones. In *Journées Francophones des langages applicatifs, Val Morin, Quebec*. INRIA, janvier 1996.

[19] C. Paulin-Mohring. Circuits as streams in Coq, verification of a sequential multiplier. Research Report 95-16, Laboratoire de l'Informatique du Parallélisme, September 1995. Available at `http://www.ens-lyon.fr:80/LIP/lip/publis/`.

[20] R. Pucella. Reactive programming in standard ml. In *IEEE International Conference on Computer Languages*. IEEE Computer Society, 1998.

[21] 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.

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