Electronic Notes in Theoretical Computer Science 11 (1998)
URL: `http://www.elsevier.nl/locate/entcs/volume11.html` 21 pages

A Co-iterative Characterization of Synchronous Stream Functions

Paul Caspi

VERIMAG, Centre Equation, 2 avenue de Vignate, 38610 Gieres, France,
e-mail: `Paul.Caspi@imag.fr`,
web: `www.imag.fr/VERIMAG/PEOPLE/Paul.Caspi`

Marc Pouzet

LIP6, 4 place Jussieu, Université Paris 6,75252 Paris cedex 05, France,
e-mail: `Marc.Pouzet@lip6.fr`,
web: `www-spi.lip6.fr/~pouzet`

Abstract

This paper presents an attempt to characterize synchronous stream functions within the framework of co-iteration and to use this characterization in building a compiler for (higher order and recursive) synchronous data-flow programs. First, length-preserving functions are considered and we show that streams equipped with such functions form a Cartesian-closed category. Then this point of view is extended toward non length-preserving ones and we stress the use of "empty" values in handling this case. Finally, the implementation we did of this material in a synchronous stream language built on top of an ML-like language is briefly described.

1 Introduction

Data-flow languages have proved quite useful in practice in several fields such as hardware description, monitoring and control systems, etc.... and it has been recognized for a long time that their semantics can be expressed in terms of streams [11,1]. In this framework, co-iteration has recently emerged as a central concept for reasoning and proving properties about stream systems [17,15] and even for compiling and optimizing them. It consists of associating to a stream:

- a transition function from states to pairs of value and state,
- and an initial state.

Its interest lies in the fact that it makes possible to handle infinite data types like streams in a strict and efficient way, instead of having to deal with them in a lazy way. This is why it is (implicitly) widely applied in synchronous data-flow compilers like the Lustre one [7].

In this framework, higher order functions can be useful, e.g. for modular compiling and reasoning and for handling dynamic (reconfigurable) data-flow by means of recursion. However, as we shall see later, higher order co-iteration can be quite complex. This can be simplified by noticing that, in many cases, we do not need general higher-order functions, but some particular ones which we call synchronous, and which can be characterized as those functions whose evaluation at a given step does not require knowing their argument's transition function, but only the value yielded by the argument at that step.

1.1 Paper content

In this paper we intend to provide a formal definition of such synchronous stream functions and to explore its use in compiling (higher order and recursive) stream programs. This will be done as follows.

– We first consider length-preserving stream functions, over which we define a useful set of combinators for abstraction, application and recursion (section 3). In particular, streams equipped with such functions form a Cartesian-closed category.

– This set of length-preserving functions is quite poor — for instance it does not allow for the definition of filter–like functions. We extend this synchronous framework to non length-preserving functions (section 4). This calls for means of statically detecting synchronous functions, referred to as a "clock calculus" (section 5).

– Then we describe briefly how we implemented this work in our synchronous stream language built on top of Objective Caml [12] (section 6).

2 Co-iteration

2.1 Streams

Assume we define streams of type T as:

$$\mathit{Stream}\,(T) = T : \mathit{Stream}\,(T)$$

and consider for instance, the following stream functions: (i) an adder, (ii) the unit delay function with initial state value `v` (a "latch" in circuits), and (iii), the function that erases every item of odd index of its input:

(i) $\texttt{plus}\ (x : xs)\ (y : ys) = (x + y) : (\texttt{plus}\ xs\ ys)$

(ii) $\texttt{pre}\ v(x : xs) = v : \texttt{pre}\ x\ xs$

(iii) $\texttt{even}\ (x : (x' : xs)) = x : (\texttt{even}\ xs)$

Co-iteration makes it possible to get rid of recursion in the above definitions by considering concrete streams:

$$\mathit{Cstream}\,(T, S) = \texttt{Co}\ (S \rightarrow T \times S)\ S$$

i.e. made of:

- a transition function of type : $S \rightarrow (T \times S)$ associating to a state, a value and a new state
- an initial state of type S

Thanks to co-iteration, the previous examples can be translated (compiled) into:

- $\texttt{plus}\ (\texttt{Co}\ t_1\ i_1)\ (\texttt{Co}\ t_2\ i_2) = \texttt{Co}\ \lambda(s_1, s_2).\texttt{let}\ v_1, s'_1 = t_1\ s_1\ \texttt{in}$
  $$\begin{aligned} &\texttt{let}\ v_2, s'_2 = t_2\ s_2\ \texttt{in} \\ &(v_1 + v_2), (s'_1, s'_2) \end{aligned}$$
  $$(i_1, i_2)$$
- $\texttt{pre}\ v\ (\texttt{Co}\ t\ i) = \texttt{Co}\ (\lambda(pre, s).(pre, (t\ s)))\ (v, i)$
- $\texttt{even}\ (\texttt{Co}\ t\ i) = \texttt{Co}\ (\lambda s.\texttt{let}\ v_1, s_1 = t\ s\ \texttt{in}\ \texttt{let}\ v_2, s'_2 = t\ s_1\ \texttt{in}\ v_1, s'_2)\ i$

Such concrete streams can be related to the initial ones thanks to the `run` function which unfolds them:

$$\texttt{run}(\texttt{Co}\ t\ s) = \texttt{let}\ v, s' = t\ s\ \texttt{in}\ v : (\texttt{run}\ (\texttt{Co}\ t\ s'))$$

Remarks:

Two concrete streams having the same runs are called bisimilar. Bisimulation is an equivalence relation and we shall not distinguish it from equality.

In a co-algebraic framework [10], the `run` function is the unique homomorphism relating the co-algebra $tx : S \rightarrow T \times S$ to the terminal co-algebra of streams $hd, tl : T^{\mathbb{N}} \rightarrow T \times T^{\mathbb{N}}$ with the usual definitions:

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

such that the following diagram commutes:

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

2.2 Stream functions

In this framework, higher order functions can be useful, e.g. for modular compiling and reasoning and for handling dynamic (reconfigurable) data-flow by means of recursion. However, higher order co-iteration can be quite complex:

the definition of a function from streams to streams requires defining an object of type:

$$((S \to T \times S) \times S) \quad \to \quad (S' \to T' \times S') \times S'$$

2.3 Synchronous stream functions

In many cases, this is unnecessarily complex. Most functions of interest like `pre` and `plus` are *synchronous* in the sense that their evaluation at a given step does not require knowing their argument's transition function, but only the value yielded by the argument at that step. Such simpler functions would require defining objects of type:

$$(T \times S') \to (T' \times S')$$

However these objects can possess their own state and be themselves dynamical systems. Thus they can be associated to objects of type:

$$\mathit{Stream}\,((T \times S') \to (T' \times S'))$$

that is,

$$\mathit{Cstream}\,((T \times S') \to (T' \times S'), S)$$

Synchronous functions from streams to streams can now be viewed as streams, allowing the extension of the synchronous point of view to any order.

3 Length-preserving functions

3.1 Examples

Typical examples of length-preserving functions are the `plus` and `pre` functions. More generally, many synchronous functions can be generated from this `pre` function as well as usual `map` functions. These `map` functions can in turn be generated from two simple primitives: `const` and `extend`. First, we need to define the type of states:

$$\mathit{State}\,(S) = \mathtt{Nil} + \mathtt{St}\,(S)$$

where `Nil` stands for "empty" states.

- The `const` function transforms an arbitrary argument into an infinite constant stream:

  $$\mathtt{const}\ v = v : \mathtt{const}\ v$$

  Its co-iterative definition is:

  $$\mathtt{const}\ v = \mathtt{Co}\ (\lambda s.(v, \mathtt{Nil}))\ \mathtt{Nil}$$

- The `extend` function applies point-wise a stream of functions to a stream of arguments:

  $$\mathtt{extend}\ (f : \mathit{fs})\ (e : \mathit{es}) = (f\ e)\ :\ \mathtt{extend}\ \mathit{fs}\ \mathit{es}$$

and can be defined as:

$$\begin{aligned}\texttt{extend}\ (\texttt{Co}\ \mathit{tf}\ \mathit{if})(\texttt{Co}\ \mathit{te}\ \mathit{ie}) = \texttt{Co}\ \lambda(\mathit{sf},\mathit{se}).&\texttt{let}\ \mathit{vf},\mathit{sf}' = \mathit{tf}\ \mathit{sf}\ \texttt{in}\\ &\texttt{let}\ \mathit{ve},\mathit{se}' = \mathit{te}\ \mathit{se}\ \texttt{in}\\ &(\mathit{vf}\ \mathit{ve}),(\mathit{sf}',\mathit{se}')\\ (\mathit{if},\mathit{ie})&\end{aligned}$$

For instance, the `plus` function could have been defined as:

$$\texttt{plus}\,x\ y = \texttt{extend}(\texttt{extend}(\texttt{const}\ (+))\ x)\ y$$

3.2 Synchronous application

A striking fact about these examples is that, in each of them, the transition function of the arguments is applied once and only once in the transition function of the result (this is why we call them *length-preserving*). In other words, it suffices to compute one item of the input to get one item of the output. It is then tempting to set apart this application by using a general application combinator:

$$\begin{aligned}\texttt{co\_apply}\ (\texttt{Co}\ \mathit{tf}\ \mathit{if})\ (\texttt{Co}\ \mathit{te}\ \mathit{ie}) = \texttt{Co}\ \lambda(\mathit{st},\mathit{sf},\mathit{se}).&\texttt{let}\ \mathit{vf},\mathit{sf}' = \mathit{tf}\ \mathit{sf}\ \texttt{in}\\ &\texttt{let}\ \mathit{ve},\mathit{se}' = \mathit{te}\ \mathit{se}\ \texttt{in}\\ &\texttt{let}\ v,\mathit{st}' = \mathit{vf}\ \mathit{ve}\ \mathit{st}\ \texttt{in}\\ &v,(\mathit{st}',\mathit{sf}',\mathit{se}')\\ (\texttt{Nil},\mathit{if},\mathit{ie})&\end{aligned}$$

$$\begin{aligned}\texttt{co\_apply} :\ &\mathit{Cstream}\,(a \to \mathit{State}\,(b) \to c \times \mathit{State}\,(b), d)\\ &\to \mathit{Cstream}\,(a,e) \to \mathit{Cstream}\,(c, \mathit{State}\,(b) \times d \times e)\end{aligned}$$

i.e. a step in applying a synchronous function consists in computing one step of the function, computing one step of the argument and applying the function's value to the argument's value (and to a possible internal state of the application) thus yielding the value and new state of the application. In doing this, we have to define an initial state of the application. When the function has not yet been applied, this state is undefined. This explains why we have to use an abstract *State* type having a constant undefined constructor `Nil`.

This allows us to propose the following definition:

Definition 3.1 *A function from streams to streams $f : (Stream\ T) \to (Stream\ T')$ is synchronous iff there exists a stream $f' : Stream\ (T \to S' \to T' \times S')$ such that:*

$$f = \texttt{co\_apply}\ f'$$

Theorem 3.2 (Synchronous morphisms [6]) *Interesting properties of this definition are:*

i The identity function id is synchronous, and

ii The composition of two synchronous functions is synchronous.

3.3 Synchronous abstraction

This calls for a converse synchronous abstraction. The idea in building it is that the transition function of the result depends only on the current value of the incoming stream. This allows us to abstract every possible incoming stream to a constant stream having this value v:

$$\begin{aligned}\texttt{co\_lambda}\ f = \texttt{const}(\lambda v, s.&\texttt{let Co}\ t\ s_0 = f\ (\texttt{const}\ v)\ \texttt{in}\\ &\texttt{let}\ s_1 = \texttt{case}\ s\ \texttt{of Nil} \rightarrow s_0 \ \mid\ \texttt{St}\,(s) \rightarrow s\ \ \texttt{in}\\ &\texttt{let}\ v_1, s_2 = t\ s_1\ \texttt{in}\\ &v_1, \texttt{St}\,(s_2))\end{aligned}$$

$$\begin{aligned}\texttt{co\_lambda} :\ &(\mathit{Cstream}\,(a, \mathit{State}\,(b)) \rightarrow \mathit{Cstream}\,(c, d))\\ &\rightarrow \mathit{Cstream}\,(a \rightarrow \mathit{State}\,(d) \rightarrow c \times \mathit{State}\,(d), \mathit{State}\,(e))\end{aligned}$$

As we can see, this illustrates the role played by the initial `Nil` state of the application: the argument function (f) is applied to a constant stream (`const` v) in order to get the transition function (t) and the initial state (s_0) of f. If the function is applied for the first time, the transition function (t) is applied to its initial state (s_0). Otherwise, it is applied to the current state of the application (s).

Clearly, this synchronous abstraction does not hold when the argument is not synchronous. However we can prove that is correct whenever its argument is synchronous, *i.e.* an image of `co_apply`.

Theorem 3.3 (Abstraction correctness [6]) *For any f we have:*

$$\texttt{co\_apply}(\texttt{co\_lambda}(\texttt{co\_apply}\ f)) = \texttt{co\_apply}\ f$$

As a consequence, streams equipped with synchronous morphisms form a category closed with respect to abstraction and application.

3.4 Recursion

Recursion is an important issue in stream processing, for instance when designing feedback control systems and circuits. As an example, the stream of natural numbers can be defined as:

$$\mathit{nat} = \texttt{pre}\ 0\ (\texttt{plus}\ \mathit{nat}\ (\texttt{const}\ 1))$$

Can we get rid of recursion in this definition? Surely we can, since it can be compiled into:

$$nat = \mathtt{Co}\ (\lambda s.(s, s+1))\ 0$$

But this calls for the question: is there a non-recursive combinator achieving this compilation for synchronous functions? The answer is only partial:

$$\begin{aligned}\mathtt{co\_rec}\ (\mathtt{Co}\ \mathit{tf}\ \mathit{if}) = \mathtt{Co}\ \lambda(se, sf).&\mathtt{let}\ vf, sf' = \mathit{tf}\ sf\ \mathtt{in}\\ &\mathtt{let\ rec}\ ve, se' = vf\ ve\ se\ \mathtt{in}\\ &ve, (se', sf')\\ (\mathtt{Nil}, \mathit{if})&\end{aligned}$$

$$\mathtt{co\_rec} : \mathit{Cstream}\,(a \to \mathit{State}\,(b) \to a \times \mathit{State}\,(b), c) \to \mathit{Cstream}\,(a, \mathit{State}\,(b) \times c)$$

we can notice here that the only recursively defined variable is ve, the value component of the transition function. Two cases can happen:

- we deal with a 0-order expression (a stream expression): then either the first element of the pair $vf\ ve\ se$ depends on ve and we have an unbounded recursion — the program contains a *causality loop* —, or it does not and the evaluation succeeds. This means that indeed the recursive evaluation of the pair ve, se' can be split into two non recursive ones. This case appears, for example, when every stream recursion appears on the right of a `pre`. As we shall see later the compiler takes advantage of this in order to produce non recursive code like the co-iterative `nat` expression given above.
- the expression is an higher order one and its boundedness depends on semantic conditions to be checked in each case.

Thus our goal of turning recursive stream definitions into non recursive ones is partially achieved provided we prove that our `co_rec` combinator implements recursion correctly:

Theorem 3.4 (Recursion correctness [6]) *For any f we have:*

$$\mathtt{co\_apply}\ f\ (\mathtt{co\_rec}\ f) = \mathtt{co\_rec}\ f$$

3.5 Products

We consider products of streams. One can associate a co-iterative process to any pair of streams (s_1, s_2) with the following definition:

$$\begin{aligned}\mathtt{product}\ (\mathtt{Co}\ t_1\ i_1)(\mathtt{Co}\ t_2\ i_2) = \mathtt{Co}\ \lambda(s_1, s_2).&\mathtt{let}\ v_1, s_1' = t_1\ s_1\ \mathtt{in}\\ &\mathtt{let}\ v_2, s_2' = t_2\ s_2\ \mathtt{in}\\ &(v_1, v_2), (s_1', s_2')\\ (i_1, i_2)&\end{aligned}$$

$$\mathtt{product} : \mathit{Cstream}\,(a, b) \to \mathit{Cstream}\,(c, d) \to \mathit{Cstream}\,(a \times c, b \times d)$$

Thus, a pair of streams can be identified with a stream of pairs: streams equipped with synchronous morphisms forms a Cartesian-closed category where any constant stream ($\texttt{const}\ i$) defines a terminal object.

4 Non length-preserving stream functions

An obstacle for viewing `even`-like functions as synchronous lies in the fact that the destructor function of the input `hd,tl` has to be applied twice in the transition function of the result. This would also be the case of filter-like functions like `when` defined as:

$$(x : xs)\ \texttt{when}\ (true : cs) \ \ = x : (xs\ \texttt{when}\ cs)$$
$$(x : xs)\ \texttt{when}\ (false : cs) = xs\ \texttt{when}\ cs$$

An obvious idea to overcome the problem and turn these functions into synchronous ones would be to consider the functor F:

$$F_T(S) = \texttt{P}(T \times S) + \texttt{S}(S)$$

where `P` stands for "present" and `S` for "silent", a silent process being one which only updates its state without outputting values. Then a transition function for `even` could be:

$$\begin{array}{l} \texttt{even}\ (\texttt{Co}\ t\ s) = \texttt{Co}\ \lambda(e, s).\texttt{case}\ t\ s\ \texttt{of} \\ \qquad\qquad\qquad\qquad\qquad \texttt{S}(sx) \to \texttt{S}(e, sx) \\ \qquad\qquad\qquad\qquad\qquad |\texttt{P}(vx, sx) \to \texttt{if}\ e\,\texttt{then}\ \texttt{P}(vx, (false, sx)) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{else}\ \texttt{S}(true, sx) \\ \qquad\qquad\qquad (true, sx) \end{array}$$

where e is a boolean state condition telling whether the current step is an even one or not. However, the question is now: does this functor still define streams? An answer to this question is as follows:

4.1 The co-algebra of clocked streams

Theorem 4.1 (Co-algebra of clocked streams) *The terminal co-algebra associated to the functor $F_T(S) = \texttt{P}(T \times S) + \texttt{S}(S)$ has*

- *as ground set the set of streams of values complemented with an empty value* `E`*:*

$$(\texttt{E} + \texttt{V}\,(T))^{I\!N}$$

- *and as destructor,*

$$dest\ (v : vs) = \texttt{case}\ v\ \texttt{of}\ \texttt{V}\,(v) \to (v, vs) \mid \texttt{E} \to vs$$

Proof. Given $t : S \to F_T(S)$ a transition function, let us denote by nxt the next state function:

$$nxt(s) = \texttt{case}\ t(s)\ \texttt{of}\ \texttt{P}(v,s) \rightarrow s \mid \texttt{S}(s) \rightarrow s$$

Any function *run* which makes the diagram

$$\begin{array}{ccc}
S & \xrightarrow{run} & (\texttt{E}+\texttt{V}\,(T))^{I\!N} \\
\Big\downarrow t & & \Big\downarrow dest \\
F_T(S) & \xrightarrow{F_T(run)} & F_T((\texttt{E}+\texttt{V}\,(T))^{I\!N})
\end{array}$$

commute yields:

$$\begin{aligned}
dest(run(s)) &= \texttt{case}\ run(s)(0)\ \texttt{of}\ \ \texttt{V}\,(v) \rightarrow (v, \lambda n.run(s)(n+1)) \\
&\qquad\qquad\qquad\qquad \mid \texttt{E} \rightarrow \lambda n.run(s)(n+1) \\
&= \texttt{case}\ t(s)\ \texttt{of}\ \texttt{P}(v,s) \rightarrow (v, run(s)) \mid \texttt{S}(s) \rightarrow (run(s))
\end{aligned}$$

$$\begin{aligned}
\textit{i.e.}\ \ run(s)(0) &= \texttt{case}\ t(s)\ \texttt{of}\ \texttt{P}(v,s) \rightarrow \texttt{V}\,(v) \mid \texttt{S}(s) \rightarrow \texttt{E} \\
run(s)(n+1) &= run(nxt(s))(n)
\end{aligned}$$

This uniquely defines *run* as:

$$run(s)(n) = \texttt{case}\ t(nxt^n(s))\ \texttt{of}\ \texttt{P}(v,s) \rightarrow v \mid \texttt{S}(s) \rightarrow \texttt{E}$$

□

Definition 4.2 *The clock of a clocked stream* $s : (\texttt{V}\,(T)+\texttt{E})^{I\!N}$ *is the boolean stream:*

$$clock(s) = \lambda n.\texttt{case}\ s(n)\ \texttt{of}\ \texttt{V}\,(v) \rightarrow true \mid \texttt{E} \rightarrow false$$

Note that clocks are just ordinary streams, *i.e.* without `E` elements.

Yet this result shows also that we can as well assimilate clocked streams with ordinary streams with "empty" values [1]. This allows us to easily reuse our results of section 3. We thus will adopt this point of view in the sequel, by taking:

$$\begin{aligned}
Value\,(T) &= \texttt{E}+\texttt{V}\,(T) \\
Clocked_stream\,(T,S) &= Cstream\,(\,Value\,(T), S)
\end{aligned}$$

4.2 Application

We can now revisit our previously defined operators as well as create new ones. But when defining binary operators, like `extend` we now find the following

[1] This quite obvious result has been used and rediscovered many times since the pioneering work of F. Boussinot [3]. Yet, the above proof may bring some insight about the need for "empty" values.

problem: what to do if one argument yields a value while the other one does not? At least three possibilities are open:

1) store the value in a state variable implementing a FIFO queue, until it matches an incoming value of the other argument,
2) generate an execution error,
3) or statically reject this situation.

As an extension of the previous work done with Lustre [7] and to be consistent with deforestation, we choose the third solution and write:

$$
\begin{array}{l}
\texttt{extend}\ (\texttt{Co}\ \mathit{tf}\ \mathit{if})(\texttt{Co}\ \mathit{te}\ \mathit{ie}) = \\
\qquad \texttt{Co}\ \lambda(\mathit{sf}, \mathit{se}).\texttt{case}\ (\mathit{tf}\ \mathit{sf}), (\mathit{te}\ \mathit{se})\ \texttt{of} \\
\qquad\qquad\qquad (\texttt{E}, \mathit{sf}), (\texttt{E}, \mathit{se}) \rightarrow \texttt{E}, (\mathit{sf}, \mathit{se}) \\
\qquad\qquad\qquad |(\texttt{V}\ (\mathit{vf}), \mathit{sf}), (\texttt{V}\ (\mathit{ve}), \mathit{se}) \rightarrow \texttt{V}\ (\mathit{vf}\ \mathit{ve}), (\mathit{sf}, \mathit{se}) \\
\qquad\quad (\mathit{if}, \mathit{ie})
\end{array}
$$

$$
\begin{array}{l}
\texttt{extend} : \mathit{Clocked\_stream}\,(a \rightarrow b, c) \rightarrow \mathit{Clocked\_stream}\,(a, d) \\
\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow \mathit{Clocked\_stream}\,(b, c \times d)
\end{array}
$$

under the condition that the clocks of the two arguments are the same. Otherwise, the program should raise an execution error (a pattern-matching failure). In order to avoid such errors, we shall associate some static rules in a system named a *clock calculus*. These rules play a critical role in the definition of our set of data-flow primitives but to make the presentation clearer, their definition will be given in the next section.

4.3 Primitive functions

Besides the `extend` function, we propose the following set of primitives

- `pre`

We can wonder whether the previous definition for `pre` extends naturally for programs which do not preserve length. Indeed, we could simply write:

$$
\begin{array}{l}
\texttt{pre}\ v\ (\texttt{Co}\ t\ i) = \texttt{Co}\ \lambda(\mathit{pre}, s).\texttt{case}\ t\ s\ \texttt{of} \\
\qquad\qquad\qquad\qquad\qquad \texttt{E}, s' \rightarrow \texttt{E}, (\mathit{pre}, s') \\
\qquad\qquad\qquad\qquad\qquad |\texttt{V}\ (v), s' \rightarrow \texttt{V}\ (\mathit{pre}), (v, s') \\
\qquad\qquad\qquad (v, i)
\end{array}
$$

Unfortunately, this definition cannot be combined with recursion in a satisfactory way. Running the co-iterative process:

$$\texttt{co\_rec}\ (\texttt{co\_lambda}\ (\lambda x.\texttt{pre}\ 0\ x))$$

implementing the stream program $x = \texttt{pre}\ 0\ x$ leads to a deadlock (corresponding to a "stack overflow" in Haskell). This is due to the fact that the input of `pre` is connected to its output and `pre` emits a value iff its input emits a value. This deadlock can be eliminated by adding an extra argument — an input clock — to `pre` controlling the production. The new definition becomes:

$$
\begin{array}{l}
\texttt{pre}\ v\ (\texttt{Co}\ t\ i)\ (\texttt{Co}\ cl\ icl) = \\
\quad \texttt{Co}\ \lambda(pre, s, scl).\texttt{case}\ cl\ scl\ \texttt{of} \\
\qquad\qquad false, scl \rightarrow \texttt{E}, \texttt{case}\ t\ s\ \texttt{of} \\
\qquad\qquad\qquad\qquad \texttt{E}, s' \rightarrow (pre, s', scl) \\
\qquad\qquad |true, scl \rightarrow \texttt{V}\,(pre), \texttt{case}\ t\ s\ \texttt{of} \\
\qquad\qquad\qquad\qquad \texttt{V}\,(v), s' \rightarrow (v, s', scl) \\
\qquad (v, i, icl)
\end{array}
$$

This time, programs are deadlock free if recursions appear on the right of a `pre`. The use of this new `pre` instead of the previous one is satisfactory if it is possible to built a system inferring the clock. This will be considered later.

`- const`
This operator is polymorphic in the sense that it may produce or not depending on its environment. For this reason, `const` should have an extra argument giving its clock. The co-iterative definition for `const` is:

$$
\begin{array}{l}
\texttt{const}\ v\ (\texttt{Co}\ cl\ icl) = \texttt{Co}\ \lambda scl.\texttt{case}\ cl\ scl\ \texttt{of} \\
\qquad\qquad true, s \rightarrow \texttt{V}\,(v), s \\
\qquad\qquad |false, s \rightarrow \texttt{E}, s \\
\qquad icl
\end{array}
$$

Like for the `pre` operator, we shall later see how the clock can be inferred.

`- when`
The co-iterative definition for the filter is as follows, supposing that its two

arguments are on the same clock:

$$
\begin{array}{l}
(\texttt{Co}\ tx\ ix)\ \texttt{when}\ (\texttt{Co}\ tc\ ic) = \\
\texttt{Co}\ \lambda(sx, sc).\texttt{case}\ (tx\ sx), (tc\ sc)\ \texttt{of} \\
\qquad\qquad (\texttt{E}, sx), (\texttt{E}, sc) \to \texttt{E}, (sx, sc) \\
\qquad\qquad |(\texttt{V}\,(vx), sx), (\texttt{V}\,(true), sc) \to \texttt{V}\,(vx), (sx, sc) \\
\qquad\qquad |(\texttt{V}\,(vx), sx), (\texttt{V}\,(false), sc) \to \texttt{E}, (sx, sc) \\
\quad (ix, ic)
\end{array}
$$

The clock of the result depends on the boolean condition. If the clock of the two arguments is $(\texttt{Co}\ tcl\ scl)$, we say that the clock of the result is $(\texttt{Co}\ tcl\ scl)$ on $(\texttt{Co}\ tc\ sc)$ with the following definition for on.

$$
\begin{array}{l}
(\texttt{Co}\ tcl\ icl)\ \texttt{on}\ (\texttt{Co}\ tc\ ic) = \texttt{Co}\ \lambda(scl, sc).\texttt{case}\ tcl\ scl\ \texttt{of} \\
\qquad\qquad\qquad false, scl' \to \texttt{let E}, sc' = tc\ sc\ \texttt{in} \\
\qquad\qquad\qquad\qquad false, (scl', sc') \\
\qquad\qquad\qquad |true, scl' \to \texttt{let V}\,(vc), sc' = tc\ sc\ \texttt{in} \\
\qquad\qquad\qquad\qquad vc, (scl', sc') \\
\qquad\qquad (icl, ic)
\end{array}
$$

$$\texttt{on} : Cstream\,(bool, a) \to Clocked\_stream\,(bool, b) \to Cstream\,(bool, a \times b)$$

Note that, according to the definition, a clock is an ordinary stream which has no "silent" move.

`- merge`

The converse of `when` whose abstract definition is:

$$
\begin{array}{l}
\texttt{merge}\ (false : cs)\ xs\ (y : ys) = y : (\texttt{merge}\ cs\ xs\ ys) \\
\texttt{merge}\ (true : cs)\ (x : xs)\ ys\ \ = x : (\texttt{merge}\ cs\ xs\ ys)
\end{array}
$$

and whose co-iterative one is:

$$
\begin{array}{l}
\texttt{merge}\ (\texttt{Co}\ tc\ ic)\ (\texttt{Co}\ tx\ ix)\ (\texttt{Co}\ ty\ iy) = \\
\quad \texttt{Co}\ \lambda(sc, sx, sy).\texttt{case}\ (tc\ sc), (tx\ sx), (ty\ sy)\ \texttt{of} \\
\qquad\qquad (\texttt{E}, sc), (\texttt{E}, sx), (\texttt{E}, sy) \to \texttt{E}, (sc, sx, sy) \\
\qquad\qquad |(\texttt{V}\,(true), sc), (\texttt{V}\,(vx), sx), (\texttt{E}, sy) \to \texttt{V}\,(vx), (sc, sx, sy) \\
\qquad\qquad |(\texttt{V}\,(false), sc), (\texttt{E}, sx), (\texttt{V}\,(vy), sy) \to \texttt{V}\,(vy), (sc, sx, sy) \\
\quad (ic, ix, iy)
\end{array}
$$

This definition does not raise any execution error if the true branch produces a value when the false branch produces no value and the condition is true, and conversely, the true branch does not produce any value when the false branch produces its value and the condition is false.

The definitions for application, abstraction and recursion remain unchanged.

4.4 Introducing and instantiating clock variables

Finally, we would like to have code that can be used in any context, *i.e.* clock polymorphic code. For example, we want to define a function f and use it in different contexts [2] :

$$\texttt{let } f = \lambda\ x.\texttt{pre}\ 0\ (x + (\texttt{const}1))$$

`pre` and `const` need their input clock cl. This clock can be set to the "base" clock (the constant stream $true$). Nonetheless, it then becomes impossible to use the function f in different contexts (with different clocks) unless f is in-lined. A solution is to parameterize f by its input clock:

$$\texttt{let } f = \lambda cl, x.\texttt{pre}\ 0\ (x + (\texttt{const}\ 1\ cl))\ cl$$

f is then polymorphic and compiled only once. The function will be called with its effective clock at every application point.

Thus, by interpreting classical stream operators, application, abstraction and recursion in their co-iterative versions, we obtain the transition function of the whole program in a very modular way.

5 Inferring the clock

The idea of the clock calculus is to provide statically checkable sufficient conditions on the synchrony of expressions. The version presented here is a generalization of the ones presented in [7], in the sense that it infers the clock, uses unification instead of fix-points and has polymorphic clocks. Moreover, it is extended to full functionality. This system is taken from [5] [3] .

Definition 5.1 *The goal of the clock calculus is to assert the judgment $H \vdash e : cl$ meaning that "expression e has clock cl in the environment H". An environment H is a list of assumptions on the clocks of free variables of e: $H ::= [x_0 : cl_0, ..., x_n : cl_n]$. A clock cl is either a clock variable α, a sub-clock of a clock, cl `on` e monitored by some boolean stream expression e or a clock function. Clock expressions are decomposed into clock schemes (σ) and clock instances (cl).*

$$cl ::= \alpha \mid cl\ \texttt{on}\ e \mid cl \xrightarrow{x} cl$$

$$\sigma ::= cl \mid \forall \alpha_1 \ldots \alpha_n . cl$$

[2] For the sake of clarity, usual scalar operators are extended implicitly to streams.

[3] The clock calculus is reminded in the present paper in order to have a self-contained presentation and because of its use in the semantics of the language.

The system is used with a generalization function. Its definition is:

$$Gen_H(cl) = \forall \alpha_1, ..., \alpha_n.cl \;\; if\, \alpha_1, ..., \alpha_n \notin FV(H) \\ \alpha_1, ..., \alpha_n \in Left(cl)$$

$$\begin{aligned} Left(cl_1 \xrightarrow{x} cl_2) &= FV(cl_1) \cup Left(cl_2) \\ Left(\sigma) &= \emptyset \end{aligned}$$

The axioms and inference rules of the clock system are given in figure 1.

The clock system has been done in the spirit of the classical Damas–Milner type system [13] with a slight modification of the recursion rule, coming from [16]. Nonetheless, it is an unconventional system since clocks contain expressions.

Let us comment here the rules:

- A constant expression matches any clock.
- Arguments of `extend` and `when` must have identical clocks.
- `not` e is the classical boolean operator point-wisely extended to streams. The expression `merge` e_1 e_2 e_3 is well clocked when the clock of e_2 is the one of e_1 restricted to the case where e_1 is true and the clock of e_3 is the one of e_1 restricted to the case where e_1 is false.
- The `pre` expression preserves the clock of its argument.
- The clock of a `when` expression depends on the values of the second argument. This is represented using the `on` construction.
- When abstracting over stream variables, we must keep track of the variable being abstracted, as it can appear in clock expressions. Moreover, the abstracted (expression) variable must not be free in any clock of the hypothesis.
- The application rule is consistent with the preceding one: the clock of the application is the clock returned by the function instantiated with its actual argument.
- `let` expressions are clocked as usual: the first argument is clocked and its clock is generalized and put in the environment. A clock expression can be generalized if it is free in the environment H and if it is the clock of an input. This constraint is unusual but has an intuitive explanation: in general, the clock of an expression can not be generalized (if a value is present, it cannot be absent at the same time!) unless the expression is a function where the clock of the result depends on the clock of the argument.
- Finally, the generic clock of a variable can be instantiated.

$$\text{CONST}\ \frac{}{H \vdash \texttt{const}\ v\ : cl} \qquad \text{EXT}\ \frac{H \vdash e_1\ : cl \quad H \vdash e_2\ : cl}{H \vdash \texttt{extend}\ e_1\ e_2\ : cl}$$

$$\text{MERGE-1}\ \frac{H \vdash e_1\ : cl \quad H \vdash e_2\ : cl\ \texttt{on}\ e_1 \quad H \vdash e_3\ : cl\ \texttt{on}\ (\texttt{not}\ e_1)}{H \vdash \texttt{merge}\ e_1\ e_2\ e_3\ : cl}$$

$$\text{MERGE-2}\ \frac{H \vdash e_1\ : cl \quad H \vdash e_2\ : cl\ \texttt{on}\ (\texttt{not}\ e_1) \quad H \vdash e_3\ : cl\ \texttt{on}\ e_1}{H \vdash \texttt{merge}\ (\texttt{not}\ e_1)\ e_2\ e_3\ : cl}$$

$$\text{PRE}\ \frac{H \vdash e\ : cl}{H \vdash \texttt{pre}\ v\ e\ : cl} \qquad \text{WHEN}\ \frac{H \vdash e_1\ : cl \quad H \vdash e_2\ : cl}{H \vdash e_1\ \texttt{when}\ e_2\ : cl\ \texttt{on}\ e_2}$$

$$\text{ABST}\ \frac{H, x\ : cl \vdash e\ : cl' \quad x \notin FV(H)}{H \vdash \lambda x.e\ : cl \xrightarrow{x} cl'} \qquad \text{APP}\ \frac{H \vdash f\ : cl \xrightarrow{x} cl' \quad H \vdash e\ : cl}{H \vdash f\ e\ : cl'[e/x]}$$

$$\text{REC}\ \frac{H, x\ : \sigma \vdash e\ : \sigma}{H \vdash \texttt{rec}\ \ x.e\ : \sigma} \qquad \text{LET}\ \frac{H \vdash e_1\ : cl_1 \quad H, x : Gen_H(cl_1) \vdash e_2 : cl_2}{H \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : cl_2[e_1/x]}$$

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

Fig. 1. The clock calculus

$$\text{CONST}\ \frac{}{H \vdash \texttt{const}\ v\ : cl \Rightarrow \texttt{const}\ v\, cl} \qquad \text{PRE}\ \frac{H \vdash e\ : cl \Rightarrow c}{H \vdash \texttt{pre}\ v\ e\ : cl \Rightarrow \texttt{pre}\ v\ c\ cl}$$

$$\text{LET}\ \frac{H \vdash e_1\ : cl_1 \Rightarrow c_1 \quad H, x : \forall \alpha_1, ..., \alpha_n.cl_1 \vdash e_2 : cl_2 \Rightarrow c_2}{H \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : cl_2[e_1/x] \Rightarrow \texttt{let}\ x = \lambda(\alpha 1, ...\alpha_n).c_1\ \texttt{in}\ c_2}$$

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

Fig. 2. Transforming stream-functions into clocked-stream functions

5.1 Transforming streams into clocked streams

Finally, we use the clock calculus to transform operators over streams into operators over clocked streams. This translation follows exactly the clock calculus and is obtained by asserting the judgment: $H \vdash e\ : cl \Rightarrow e'$ meaning that the expression e with clock cl can be transformed into the expression e'. The predicate is defined in figure 2. The `const` and `pre` receive their clock as a new argument. When a clock is generalized, the clock variables are abstracted by adding extra arguments. The clocks are then instantiated with the actual clock at every use of the variable. In this way, declarations are compiled once for all. All other rules are simple morphisms and will be omitted.

6 Compiler overview

The principles we have presented in this paper are implemented in an experimental language named LUCID Synchrone [4] , a functional data-flow language combining the features of ML and the synchronous primitives of Lustre and built on top of Objective Caml.

The compiler takes a program made of a sequence of declarations and produces a transition function (written in Objective Caml) for every declaration. The implemented language is more ambitious than the kernel presented here

[4] Name built from the LUCID language [1] and from the French word synchrone. The implementation can be accessed by anonymous `ftp` at `spi.lip6.fr` in the directory `/pub/lucid-synchrone` or thru the author's web page.

in severy aspects:

- Programs are written in an ML style (precisely in a subset of Objective Caml) where every operator and constant is (implicitely) extended to streams (e.g, `1` is an infinite constant stream)
- Expressions (streams or functions) can be mutually recursives and arbitrarily nested
- Tuple patterns and products can be defined.
- Some simple module facilities are provided for importing scalar values from Objective Caml and stream values from other LUCID Synchrone modules.
- A real-time mode is provided where the compiler checks that the input program is not dynamic (it does not contain any recursive function) and thus can be compiled into a "real-time" transition function (it answers with a bounded response time and uses a bounded memory).

Besides a classical type inference *a la* Hindley-Milner and the clock calculus already presented, the compiler encompasses

- a *causality analysis* forbidding the definition of causality loops,
- an optimizer for avoiding the creation of an infinite number of useless recursive processes when dealing with dynamic networks of processes, like the celebrated Eratosthenes sieve,
- and the *generation of transition functions* obtained from an optimized version of the co-iterative definitions presented in this paper. In this step, mutual stream recursions are eliminated by statically scheduling the program.

7 Related issues

7.1 Synchrony and reactive systems

The concept of synchrony has emerged quite naturally from the field of so-called "reactive systems" [8] as a way of characterizing parallel processes progressing in a locked-step mode [14]. Its extension toward a stream-based framework can be motivated in two ways:

– reactive systems exhibit infinite behaviors that are best taken into account by notions of co-algebra and co-inductive types [10]. Yet it does not seem that the idea of synchronous function has been identified there;

– data-flow (*i.e.* stream) programming has been recognized for long as an important part of general synchronous programming, as exemplified with the languages Lustre [7] and Signal [2]. But none of these languages encompass notions of higher order computing.

7.2 Synchrony and deforestation

Furthermore, the idea of synchronous stream programming has been recently [5] linked to the ones of "listlessness" and "deforestation" [19], which have, in turn, been extensively studied for a while [18]. Yet, the study of higher order deforestation still remains in its infancy.

7.3 Synchrony and hylomorphisms

Hylomorphisms and associated fusion theorems are closely related to deforestation [9] and look very much like our synchronous functions. However, no higher order theory is provided and furthermore, synchronous functions are not based on natural transformations:

We can write

$$\psi \circ \texttt{pre} = G_T(\texttt{pre}) \circ \eta_{T\times S} \circ \phi$$

where

- $F_T(S) = T \times S$,
- $G_T(S) = T \times S$,
- $\phi : T \times S \to F_T(T \times S)$ is any stream co-algebra,
- $\eta_{T\times S}(x, (v, s)) = v, (x, s)$
- $G_T(\texttt{pre}) = id \times \texttt{pre}$,
- $\psi : T^N \to G_T(T^N) = hd, tl$ is a terminal stream co-algebra.

Furthermore, let $hd, tl : T^N \to F_T(T^N)$ be a terminal co-algebra associated with F_T and let run be the corresponding unique morphism such that

$$(hd, tl) \circ run = F_T(run) \circ \phi$$

Assume now that η is a natural transformation from F_T to G_T. We should find some η_{T^N} such that the following equation hold:

$$\eta_{T^N} \circ F_T(run) = G_T(run) \circ \eta_{T\times S}$$

Now,

$$\begin{aligned} G_T(run) \circ \eta_{T\times S}(x, (v, s)) &= G_T(run)(v, (x, s)) \\ &= v, xs \\ F_T(run)(x, (v, s)) = x, xs & \end{aligned}$$

where $xs : T^N$ is some sequence.

Since the v value is lost we cannot find η_{T^N} such that

$$\eta_{T^N}(x, xs) = v, xs$$

Thus, strictly speaking, hylomorphisms (in fact co-hylomorphisms) are not synchronous functions, but only static state-less ones.

7.4 Previous works

In a previous work [5] we had the same goal as in this paper, *i.e.* to provide functional extensions of synchronous stream programming languages but we mainly founded this extension on a so-called "synchronous operational" semantics based on "SOS" rules which was less practical than the combinator approach proposed here. As a by-product, the compiling method proposed at that time was limited and less clear than it is by now. We have now a general characterization of synchrony and a complete implementation of the language. Thus, this paper can be seen as providing some firmer basis to the previous one as well as continuing it. Furthermore, the approach proposed here may be expected to be useful in other related fields such as verification and proof.

8 Conclusion

In this paper we have presented a characterization of synchrony in terms of co-iteration which has been used for building a (higher-order and recursive) stream function language on top of a ML-like language.

This language is young and some interesting problems still have to be solved. In particular, an exact characterization of reactive stream programs (*i.e.* bounded memory and reaction time ones) is still lacking (it requires characterizing tail-recursive dynamical networks).

Furthermore it may be expected that this theory of synchrony be useful in other domains; for instance most stream functions already considered when applying theorem proving to streams [15,17] are indeed synchronous and the set of combinators proposed here may be useful in handling modular proofs and systematic translations from streams to prover languages. It may also be expected that it be usefully applied to other infinite data types, like trees.

Finally, another interest could be to contribute in bridging the gap between general purpose functional programming and reactive programming, by showing that the latter is just a particular case of the former.

Acknowledgement

We gratefully acknowledge Thérèse Hardin from LIP6 and Catherine Oriat from Imag/LSR for their suggestions about this work.

References

[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] F. Boussinot. Proposition de sémantique dénotationnelle pour des réseaux de processus avec mélange équitable. *Theoretical Computer Science*, 18(173–206), 1982.

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

[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. Verimag Research Report, 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.* ACM SIGPLAN, Philadelphia May 1996.

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

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

[12] X. Leroy. The Objective Caml system release 1.05. Documentation and user's manual. Technical report, INRIA, March 1997.

[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 *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] 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.

[18] M. H. Sørensen, R. Glück, and N. D. Jones. Towards unifying partial evaluation, deforestation, supercompilation and GPC. In *European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 485–500. Springer Verlag, 1993.

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