A Co-iterative Characterization of Synchronous Stream Functions

Paul Caspi

VERIMAG
Centre Equation
2 avenue de Vignate
38610 GIERES France
e-mail: `caspi@imag.fr`

Marc Pouzet

LIP6
couloir 45 – 55
4 place Jussieu
Université Paris VI
75252 Paris cedex 05 France
e-mail: `pouzet@spi.lip6.fr`
`http://www.cadillac.lip6.fr/~pouzet`

VERIMAG Research Report 97–07

October 1997

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 package built on top of an ML-like language is briefly described.

Contents

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 [10, 1]. In this framework, co-iteration has recently emerged as a central concept for reasoning and proving properties about stream systems [16, 14] 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 [6].

In this framework, higher order 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. Some simplification can be brought here 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 this concept of 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 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 how we implemented this work in our synchronous stream package built on top of "Caml Light" [11] (section 6).

1.2 Related issues

1.2.1 Synchrony and reactive systems

The concept of synchrony has emerged quite naturally from the field of so-called "reactive systems" [7] as a way of characterizing parallel processes progressing in a locked-step mode [13]. 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 [9]. 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 [6] and SIGNAL [2]. But none of these languages encompass notions of higher order.

1.2.2 Synchrony and deforestation

Furthermore, the idea of synchronous stream programming has been recently [5] linked to the ones of "listlessness" and "deforestation" [18, 19], which have, in turn, been extensively studied for a while [17]. Yet, the study of higher-order deforestation still remains in its infancy.

1.2.3 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 far less practical than the combinator approach proposed here. As a by-product, the compiling method proposed at that time was also far less clear than it is by now. However, 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.

2 Co-iteration

For instance, consider the following stream functions, as defined in HASKELL [8][1]:

- the stream type definition:

```
data Stream t = t : Stream t
```

- the unit delay function, with initial value `v`:

```
pre v (x:xs) = v : pre x xs
```

[1] In this paper, programs will in general be written in HASKELL. This language is used for its ability to express both programs to be compiled (which should manage infinite data-structures and hence be lazy) and strict compiled programs, and also for its type-system which is used as a (partial) proof-checker.

- an adder:

  ```
  plus (x:xs) (y:ys) = (x + y) : (plus xs ys)
  ```

- the function that erases every item of odd index of its input stream:

  ```
  even (x:(x':xs)) = x:(even xs)
  ```

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

```
type  F t s = (t , s)

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

i.e. made of:

- a transition function of type : `s -> F t s` associating to a state, a value and a new state, and
- an initial state of type `s`:

Thanks to co-iteration, the previous examples can be translated (compiled) into [2]:

- ```
  co_pre v (Co tx ix) = Co (\(s,sx)->let (v1, sx1) = tx sx
                                      in  (s, (v1,sx1))
                           ) (v,ix)
  ```

- ```
  co_plus (Co tx ix) (Co ty iy)=
  Co (\(sx,sy)->let  (vx, sx') = tx sx
                     (vy, sy') = ty sy
                in   ((vx + vy), (sx',sy'))
     ) (ix,iy)
  ```

- ```
  co_even (Co tx ix) = Co (\sx->let (vx1, sx1) = tx sx
                                    (vx2, sx2) = tx sx1
                                in  (vx2, sx2)
                          ) ix
  ```

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

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

[2] In HASKELL, the abstraction $\lambda x.e$ is noted `\x -> e` and a curried function $\lambda x, y.e$ is noted `\x y-> e`
```

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 [9], the `run` function is the unique homomorphism relating the co-algebra `tx : s -> F t s` to the terminal co-algebra of streams `hd,tl :` $\texttt{t}^N$ `-> F t` $\texttt{t}^N$ with the usual definitions:

```
hd (s)    =  s(0)
tl (s)(n) = s(n+1)
```

such that the following diagram commutes:

$$
\begin{tikzcd}
\texttt{s} \arrow[r, "\texttt{run}"] \arrow[d, "\texttt{tx}"] & \texttt{t}^N \arrow[d, "\texttt{hd},\texttt{tl}"] \\
\texttt{F t s} \arrow[r, "\texttt{F id run}"] & \texttt{F t t}^N
\end{tikzcd}
$$

2.1 Stream functions

As we said above, higher order co-iteration can be quite complex. In general, the definition of a function from streams to streams needs defining an object of type:

```
(s ->(F t s)) -> s ->(s'-> (F t' s')) , s'
```

or, isomorphically:

```
(s ->(F t s)) -> s ->(s'-> (F t' s')) , (s ->(F t s)) -> s -> s'
```

For instance, the `even` function defined above is isomorphic to the pair of functions `teven, seven` defined as:

```
teven tx sx = (\sx ->let (vx1, sx1) = tx sx
                         (vx2, sx2) = tx sx1
                     in (vx2, sx2)
              )

seven tx sx = sx
```

such that

```
co_even1 (Co tx sx) = Co (teven tx sx) (seven tx sx)
```

2.2 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 -> s'-> F t' s'
```

For instance, the `pre v` function can be re-defined thanks to `tpre`:

```
tpre vx sx = (sx, vx)
```

as:

```
co_pre1 v (Co tx sx) =
Co (\(s,sx)->let (vx, sx') = tx sx
                 (v,  s')  = tpre vx s
             in (v, (s', sx'))
   ) (v,sx)
```

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

```
Stream (t -> s' -> (F t' 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 such functions can be generated from this `pre` function as well as usual `map` functions. These `map` functions can in turn be generated from the two following ones:

- the `const` function which transforms an arbitrary argument into an infinite constant sequence:

  ```
  const v = v:const v
  ```

  Its co-iterative definition is:

```
data State s = Nil | St s

co_const v = Co (\s -> (v, Nil)) Nil

co_const :: t -> CStream t (State s)
```

where the auxilliary definition of `State` accounts for introducing undefined or useless states.

- and the `extend` function which applies pointwise a stream of functions to a stream of arguments:

```
extend (f:fs) (e:es) = (f e):extend fs es
```

and can be defined as:

```
co_extend (Co f i) (Co e ie) =
 Co (\(sf,se) -> let (vf, sf') = f sf
                     (ve, se') = e se
                 in  ((vf ve), (sf',se'))
    ) (i,ie)


co_extend :: (CStream (t -> t') s') -> (CStream t s) ->
             CStream t' (s', s)
```

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

```
co_plus1 x y = extend (extend (const (+)) x) y
```

Another interesting function is `fby` (the "followed-by" function coined by the authors of LUCID [1]) which generalizes `pre` by offering a variable initialization. Its stream definition is:

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

and its co-iterative one:

```
(Co tx ix) `co_fby` (Co ty iy) =
 Co (\(init,sx,sy)->let (vx, sx') = tx sx
                        (vy, sy') = ty sy
                    in  case init
                        of Nil  -> (vx, ((St vy),sx',sy'))
                           St v -> (v,  ((St vy),sx',sy'))
    ) (Nil,ix,iy)

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

`fby` has an internal state used to distinguish between the first instant and the others. This state is initialized with the special state value `Nil`.

Remark: It may seem strange that, in this definition, the two input streams are continuously unfold, though, after the first instant, only the second one is needed. The reason for doing this will appear in the next section. In practice, compiler optimization can easily get rid of such useless computations.

3.2 Synchronous application

A striking fact about these examples is that, in each of them, the transition function of the argument 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:

```
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)
```

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 not determined. 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 1 (Synchronous stream function) *A function from streams to streams*

```
f : Stream  t -> Stream t'
```

is synchronous iff there exists a stream

```
f': Stream  t -> s' -> F t' s'
```

such that:

```
f = co_apply  f'
```

Theorem 1 (Synchronous morphisms) *Interesting properties of this definition are:*

i The identity function `id` *is synchronous, and*

ii The composition of two synchronous functions is synchronous.

The proof is given in appendix.

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`:

```
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)
```

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 (`i`) of `f`. It the function is applied for the first time, the transition function (`t`) is applied to its initial state (`i`). Otherwise, it is applied to the current state of the application (`s`).

Clearly, this synchronous abstraction does not hold when its argument is not indeed synchronous. Now we wish to prove the correctness of the abstraction whenever its argument is synchronous, *i.e.* an image of `co_apply`.

Theorem 2 (Abstraction correctness) *For any* `f'` *we have:*

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

The proof can be found in appendix. 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:

```
nat = pre 0 (plus nat (const 1))
```

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

```
co_nat = Co (\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:

```
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')
```

Remembering that in HASKELL `let` constructs are recursive, 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: 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` or `fby`. 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 of higher order 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 (Recursion correctness) *For any* `f` *we have:*

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

The proof is left in appendix.

3.5 Products

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

```
co_product (Co e1 i1) (Co e2 i2) =
Co (\(s1,s2) ->let (v1, s1') = e1 s1
                   (v2, s2') = e2 s2
               in  ((v1,v2), (s1', s2'))
   ) (i1, i2)

co_product :: (CStream t1 s1) -> (CStream t2 s2) ->
              CStream (t1 , t2) (s1 , s2)
```

Thus, a pair of streams can be identified with a stream of pairs: streams equipped with synchronous morphisms form a Cartesian closed category.

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

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

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

where `P` stands for "present" and `S` for "silent ", a silent process being one which only updates its state without ouputing values.

Then a transition function for `even` could be:

```
co_even2 (Co tx ix) = Co (\(e,sx)->
                           case tx sx
                           of S sx'    -> S (e, sx')
                              P vx sx' -> if e
                                          then P vx (False,sx')
                                          else S (True,sx')
                          ) (True,ix)
```

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 (Co-algebra of clocked streams) *The terminal co-algebra associated to the functor*

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

has

- *as ground set the set of streams of values complemented with an empty value:* $\texttt{(Val t)}^N$ *where* `data Val t = E | V t`
- *and as destructor:*

```
dest (v : vs) = case v of E   -> S vs
                          V v -> P v vs
```

Proof: Given `tx : s -> F t s` a transition function, let us denote by **next** the iterated next state function:

```
next s = case tx s of S s' -> s'; P v s' -> s'
next n s = if n = 0 then  (next s) else next (n-1) (next s)
```

Any function **run** which makes the diagram

$$\begin{array}{ccc}
\texttt{s} & \xrightarrow{\texttt{run}} & (\texttt{Val t})^N \\
\Big\downarrow{\scriptstyle \texttt{tx}} & & \Big\downarrow{\scriptstyle \texttt{dest}} \\
\texttt{F t s} & \xrightarrow{\texttt{F id run}} & \texttt{F t}\,(\texttt{Val t})^N
\end{array}$$

commute yields:

```
dest(run s) = case ((run s) 0) of E   -> S \n.(run s (n+1))
                                  V v -> P v \n.(run s (n+1))
            = case t s of S s'   -> S (run s')
                          P v s' -> P v (run s')
```

i.e.

```
run s 0     = case t s of S s'   -> E ; P v s' -> V v
run s (n+1) = case t s of S s'   -> run s' n
                          P v s' -> run s' n
            = run (next s) n
```

This uniquely defines `run` as:

```
run s n = case t (next n s) of S s'  -> E; P v s' -> V v
```

Definition 2 (Clocks) *The clock of a clocked stream* `s : (Val T)`^{N} *is the boolean stream:*

```
clock s n = case s n of E -> False ; V v -> True
```

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[3]. This allows us to easily reuse our results of section 3. We thus will adopt this point of view in the sequel, by taking:

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

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 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 [6] and to be consistent with deforestation, we choose the third solution and write:

```
co_extend (Co f i) (Co e ie) =
Co (\(sf,se) ->
     case ((f sf),(e se))
     of  ((E, sf),(E, se))       -> (E, (sf,se))
         ((V vf, sf),(V ve, se)) -> (V (vf ve) , (sf,se))
   ) (i,ie)

co_extend :: (CStream (Val (t -> t')) s') -> (CStream (Val t) s)
                -> CStream (Val t') (s', s)
```

[3]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.

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:

```
copre v (Co e ie) =
Co (\(pre,se) ->case e se
                of  (E, se')    -> (E, (pre,se'))
                    (V v , se') -> (V pre , (v,se'))
   )(v,ie)

copre :: t -> (CStream (Val t) s) -> CStream (Val t) (t , s)
```

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

```
xx = co_rec (co_lambda (\x -> copre 0 x))
```

implementing the stream program:

```
x = pre 0 x
```

leads to a deadlock (the system's answer is `Control stack overflow`). This is due to the fact that the input of `pre` is connected to the 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:

```
co_pre v (Co e ie) (Co cl icl) =
Co (\(pre,se,scl) ->
     case cl scl
     of (False, scl') ->
          (E, (let (E, se') = e se in (pre,se',scl')))
        (True, scl')  ->
          (V pre, (let (V v, se') = e se in (v,se',scl')))
   )(v,ie,icl)

co_pre :: t -> (CStream (Val t) s) -> (CStream Bool s') ->
          CStream (Val t) (t , s , s')
```

This time, programs are deadlock free provided 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 to infer 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:

```
co_const1 v (Co tc ic) =
Co (\s -> case tc s of (True, s')  -> (V v, s')
                       (False, s') -> (E, s')
   ) ic

co_const1 :: t -> (CStream Bool s) -> CStream (Val t) s
```

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

- `fby` To avoid deadlocks, `fby` can be defined as:

```
(Co tx ix) `co_fby` (Co ty iy) =
 Co (\(init,sx,sy) ->
      case tx sx
      of (E, sx')    -> (E, let (E, sy') = ty sy
                                in (init,sx',sy'))
         (V vx, sx') -> ((case init
                            of Nil  -> V vx
                               St v -> V v),
                            let (V v, sy') = ty sy
                            in  ((St v),sx',sy'))
     ) (Nil,ix,iy)

co_fby :: (CStream (Val t) s) -> (CStream (Val t) s') ->
        CStream (Val t) ((State t) , s , s')
```

As before, this definition is correct provided the two arguments have the same clock.

- `when`: The co-iterative definition for the filter is as follows, assuming its two arguments share the same clock:

```
(Co tx ix) `co_when` (Co tc ic) =
 Co (\(sx,sc)->
       case (tx sx ,(tc sc))
       of ((E, sx'),(E, sc'))          -> (E, (sx',sc'))
          ((V vx, sx'),(V True,  sc')) -> (V vx, (sx',sc'))
          ((V vx, sx'),(V False, sc')) -> (E, (sx',sc'))
    ) (ix,ic)

co_when :: (CStream (Val t) s) -> (CStream (Val Bool) sc)
          -> CStream (Val  t) (s , sc)
```

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

```
(Co tcl icl) `on` (Co tc ic) =
 Co (\(scl,sc)-> case tcl scl
                of (False, scl')-> let (E, sc') = tc sc
                                   in (False, (scl',sc'))
                   (True, scl') -> let (V vc, sc') = tc sc
                                   in  (vc, (scl',sc'))
    ) (icl,ic)

on :: (CStream Bool sc) -> (CStream (Val Bool) s) ->
       CStream Bool (sc , s)
```

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:

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

and whose co-iterative one is:

```
co_merge (Co tc ic) (Co tx ix) (Co ty iy) =
Co (\ (sc,sx,sy) ->
      case (tc sc, tx sx, ty sy)
      of ((E, sc'),(E, sx'),(E, sy')) ->
                 (E, (sc',sx',sy'))
         ((V True, sc'),((V vx), sx'),(E, sy')) ->
                 (V vx, (sc',sx',sy'))
         ((V False, sc'),(E, sx'),(V vy, sy')) ->
                 (V vy, (sc',sx',sy'))
    ) (ic,ix,iy)

co_merge :: (CStream (Val Bool) sc) -> (CStream (Val t) s) ->
            (CStream (Val t) s') -> CStream (Val t) (sc , s , s')
```

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 instanciating 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 to use it in different contexts [4]:

```
    f = \x -> let nat = pre 0 (nat + (const 1)) in
              nat + x
...
    (f e1) + (merge c (f (e1 when c)) (const 2))
```

Because `pre` and `const` need their input clock, the body of `f` should be first translated into:

```
let nat = pre 0 (nat + (const 1 c)) c
in  nat + x
```

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

where `c` is the clock of `const` and `pre`. This clock can be set to the base clock `const True`. Nonetheless, it then becomes impossible to use the function `f` in its two distinct contexts unless `f` is first inlined. `f` becomes polymorphic by considering `c` as a free variable which is abstracted in `f` and then instanciated with its actual clock at the application point. These transformations correspond exactly to classical generalisation and instanciation steps in type-inference systems. The code of `f` is transformed into:

```
f = \c x -> let nat = pre 0 (nat + (const 1 c)) c
            in  nat + x
```

and the program using `f` is translated into:

```
(f cl1 e1) + (merge c (f (cl1 on c) (e1 when c)) (const 2 cl1))
```

if `cl1` is the clock of `e1`. Finally, 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 conditions allowing an expression to be synchronously evaluated. The version presented here is a generalization of the ones presented in [6], 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 similar to the one presented in [5].

Definition 3 (Clock calculus) *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, a product of clocks or a clock function. Clock expressions are decomposed into clock schemes (σ) and clock instances (cl).*

$$\begin{array}{rcl} cl & ::= & \alpha \mid cl \ \mathbf{on}\ e \mid cl \xrightarrow{x} cl \mid cl \times cl \\ \sigma & ::= & cl \mid \forall \alpha_1 \ldots \alpha_n . cl \end{array}$$

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

$$\begin{array}{ll} Gen_H(cl) = \forall \alpha_1, ..., \alpha_n . cl & \text{if } \alpha_1, ..., \alpha_n \not\in FV(H) \\ & \alpha_1, ..., \alpha_n \in Left(cl) \end{array}$$

$$\begin{array}{rcl} Left(cl_1 \xrightarrow{x} cl_2) & = & FV(cl_1) \cup Left(cl_2) \\ & = & \emptyset \ \ \textit{otherwise} \end{array}$$

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 [12] with a slight modification of the recursion rule, coming from [15]. Nonetheless, it is an unconventional system since clocks contain expressions.

Let us comment here the rules:

- A constant expression matches any clock.
- The clocks of The two arguments of an `extend`, a `fby` or a `when` must have identical clocks.
- The clock of a `when` expression depends on the values of the second argument. This is represented using the `on` construction.
- `not` e is the classical boolean operator extended pointwise to streams. The expression `merge` e_1 e_2 e_3 uses a e_2 item when the value of e_1 is true, else, it uses a e_3 item. Thus, such an expression 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.
- When abstracting over stream variables, we must keep trace of the variable being abstracted, as it can appear in clock expressions.
- 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 in the classical way: 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.

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 figures 2 and 3. The `const` and `pre` receive their clock as a new argument. When a clock is generalized, the clock variables are abstracted and then instantiated with their actual clocks. All the other rules are simple morphisms.

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

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

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

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

$$\text{PRE}\ \frac{H \vdash e\ : cl}{H \vdash \texttt{pre}\ v\ e\ : cl} \qquad \text{FBY}\ \frac{H \vdash e_1\ : cl \quad H \vdash e_2\ : cl}{H \vdash e_1\ \texttt{fby}\ e_2\ : cl}$$

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

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

$$\text{REC}\ \frac{H, x\ : \sigma \vdash e\ : \sigma}{H \vdash \texttt{rec}\ \ x.e\ : \sigma}$$

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

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

Figure 1: The clock calculus

$$\text{CONST}\quad \frac{}{H \vdash \texttt{const}\ v\ : cl \Rightarrow \texttt{const}\ v\, cl}$$

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

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

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

Figure 2: Transforming stream-functions into clocked-stream functions

6 Compiling issues

The compiling principle is to transform any stream program into its co-iterative process by replacing every construction in the program by its co-iterative definition.

In this section, we briefly present the method that has been implemented for an experimental language named LUCID SYNCHRONE[5]. The tool can be accessed at
`http://cadillac.ibp.fr/~pouzet`.

This implementation (about 7000 lines of Caml Light [11] code) takes as input, a Caml Light program[6] with synchronous data-flow primitives but without type definitions nor imperative features and produces for every function definition, its corresponding transition function. These transition functions are valid Caml Light (or Objective Caml) programs.

The implemented language is more ambitious than the kernel presented here in the sense that it allows the definition of mutually recursive expressions (streams or functions) and has tuple-patterns. Moreover, a simple preprocessor allows to extend implicitly classical scalar primitives and constants to streams. For example, the stream constant `const 1` is simply written `1` and `extend (extend (const (+)) x) y` is simply written `x+y`.

The compilation is composed of several steps that are discussed below.

6.1 Type inference, clock calculus and causality analysis

- The program is first typed using a classical typing inference. There is no special treatment in this step: the system gives the same types as HASKELL (or any other

[5]Name built from the LUCID language [1] and from the French word synchrone.
[6]Note that at this point we move to a Caml Light syntax

$$\text{EXT}\quad \frac{H \vdash e_1 \ : cl \Rightarrow c_1 \quad H \vdash e_2 \ : cl \Rightarrow c_2}{H \vdash \texttt{extend}\ e_1\ e_2 \ : cl \Rightarrow \texttt{extend}\ c_1\ c_2}$$

$$\text{WHEN}\quad \frac{H \vdash e_1 \ : cl \Rightarrow c_1 \quad H \vdash e_2 \ : cl \Rightarrow c_2}{H \vdash e_1\ \texttt{when}\ e_2 \ : cl\ \texttt{on}\ c_2 \Rightarrow c_1\ \texttt{when}\ c_2}$$

$$\text{MERGE-1}\quad \frac{\begin{array}{c} H \vdash e_3 \ : cl\ \texttt{on}\ (\texttt{not}\ c_1) \Rightarrow c_3 \\ H \vdash e_1 \ : cl \Rightarrow c_1 \quad H \vdash e_2 \ : cl\ \texttt{on}\ c_1 \Rightarrow c_2 \end{array}}{H \vdash \texttt{merge}\ e_1\ e_2\ e_3 \ : cl \Rightarrow \texttt{merge}\ c_1\ c_2\ c_3}$$

$$\text{MERGE-2}\quad \frac{\begin{array}{c} H \vdash e_3 \ : cl\ \texttt{on}\ c_1 \Rightarrow c_3 \\ H \vdash e_1 \ : cl \Rightarrow c_1 \quad H \vdash e_2 \ : cl\ \texttt{on}\ \texttt{not}\ c_1 \Rightarrow c_2 \end{array}}{H \vdash \texttt{merge}\ (\texttt{not}\ e_1)\ e_2\ e_3 \ : cl \Rightarrow \texttt{merge}\ (\texttt{not}\ c_1)\ c_2\ c_3}$$

$$\text{FBY}\quad \frac{H \vdash e_1 \ : cl_1 \Rightarrow c_1 \quad H \vdash e_2 \ : cl_1 \Rightarrow c_2}{H \vdash e_1\ \texttt{fby}\ e_2 \ : cl_1 \Rightarrow c_1\ \texttt{fby}\ c_2}$$

$$\text{REC}\quad \frac{H, x \ : \sigma \vdash e \ : \sigma \Rightarrow c}{H \vdash \texttt{rec}\ \ x.e \ : \sigma \Rightarrow \texttt{rec}\ \ x.c}$$

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

$$\text{APP}\quad \frac{H \vdash f \ : cl \xrightarrow{x} cl' \Rightarrow c_f \quad H \vdash e \ : cl \Rightarrow c_e}{H \vdash f\ e \ : cl'[c_e/x] \Rightarrow c_f\ c_e}$$

Figure 3: Transforming stream-functions into clocked-stream functions

ML-like language) would give.

- The program is clocked and every expression is annotated with its clock. The implementation of the clock calculus has been done in the spirit of classical implementations of ML with the following restrictions:

  - In this system, unification is different from the classical one since clocks may contain expressions. In this implementation, we restrict the unification between expressions in clocks to the syntactical equality.
  - Recursive expressions should be clocked in an iterative way, as noted in [15]. In this implementation, only two iteration steps are performed; programs are rejected if a fixed point is not reached so far.
  - Substitution of `let` definitions is not done to avoid code size increase in clocks. For example, the simple function:

    ```
    let f x = let x = x + 1
                  half = pre true (not half)
              in  x when half
    ```

    is rejected. The program should be re-written into:

    ```
    let f x = let x = x+1
              in  x when (rec half. pre true (not half))
    ```

    Note that programs can be rewritten to be accepted by the system.
  - In the current implementation, the arrow is decomposed into two categories: the dependent clock $cl \xrightarrow{x} cl'$ when x appear in cl or cl' and the independent clock $cl \rightarrow cl'$ otherwise. Then, the classical restriction of the Milner-type system applies: valid clocks are those where a dependent clock does not appear on a left of an arrow (prenex form). The implemented rules for abstraction and application are given in figure 4.

- A causality analysis is done on the program. It checks that every recursive use of a stream appear on the right of a `pre` or a `fby`.

6.2 Production of co-iterative code and scheduling

After the clock computation, the compiler produces a Caml-light program containing a transition function for every definition in the source program. This compilation is syntax directed. In the current implementation, the transition function is a pure ML program where new states are returned by the transition function.

$$\text{ABST-dependent}\quad \frac{H, x : cl \vdash e : cl' \quad x \notin FV(H) \quad x \in FV(cl) \cup FV(cl')}{H \vdash \lambda x.e : cl \xrightarrow{x} cl'}$$

$$\text{ABST-independent}\quad \frac{H, x : cl \vdash e : cl' \quad x \notin FV(H) \quad x \notin FV(cl) \cup FV(cl')}{H \vdash \lambda x.e : cl \rightarrow cl'}$$

$$\text{APP-independent}\quad \frac{H \vdash f : cl \rightarrow cl' \quad H \vdash e : cl}{H \vdash f\ e : cl'}$$

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

Figure 4: The implemented rules for abstraction and application

6.2.1 Co-iterative code

The compilation method is obtained from the semantics of length preserving functions, adding two optimizations coming from the following observations:

- In the `Co tx sx` structure, the `tx` function has type `s -> t * s` and `sx` has type `s`. Thus, `tx` can be decomposed into an argument `sig` of type `s` and an expression `let def in (v,st)` such that

  ```
  f = \sig. let def in (v,st)
  ```

  The sequence `def` is used by both the value (`v`) and the state part (`st`) of the result of `tx`. The interest of this representation is that it is preserved by composition. It makes it possible to optimize the produced code by avoiding all useless intermediate function applications which appear in the definition of co-iterative combinators.

- The abstraction combinator is an optimized version of the one we have presented since names are known at compile-time. This combinator `new_co_lambda x e` is semantically equivalent to `co_lambda (\x -> e)`. The same principle is applied to the fix-point combinator.

- The representation of empty values and non-empty ones in the transition function is useless since every primitive knows (with the clock) if an argument is present or not. It allows to eliminate the pattern matchings testing whether a value is present or not. Thus, we consider that an expression always produce a value but state modifications — in `fby` and `pre` — become effective only when the value is present (the clock is true). These two primitives are the only one to take their clock as input. Thus, the `const` generator always produce a scalar value as well as `e when c` which acts exactly as the expression `e`.

- States are flattened to be represented as tuples (vectors or records). This is obtained by applying a special combination between states. $s_1 \circ s_2$ is the composition of two states (instead of the pair (s_1, s_2)). The flattening is obtained by applying rules like::

$$(e_1, ..., e_n) \circ (e_{n+1}, ..., e_k) \quad = \quad (e_1, ..., e_k)$$

These tuples can be represented with records. This representation will make it possible to do state modifications "in-place".

6.2.2 Scheduling and causality loops

Recursive definitions (possibly mutual) of synchronous streams are transformed into non recursive ones. This is done by defining a mutual combinator `let_rec` doing some scheduling between definitions and uses. For example, with the simple recursion:

```
let rec nat = pre 0 (nat+1)
```

the compiler first produces something like:

```
Co (\pre->(pre,nat+1)) 0
```

for the body `pre 0 (nat+1)`, then it produces:

```
Co (\pre -> (let rec nat,s = pre,nat+1)) 0
```

which is then transformed by a scheduling step into the non recursive program:

```
Co (\pre -> (let nat = pre in (pre,nat+1))) 0
```

Mutually recursive definitions are treated in the same way by finding a sequential order between computations that satisfies the def-use chains. Such an ordering is impossible in case of *causality loop* like in:

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

since the compiler cannot eliminate recursion. Other examples can be found at the Web site given above.

7 Conclusion

In this paper we have presented a characterization of synchrony in terms of co-iteration and we hope to have illustrated its use in building a (higher-order and recursive) stream function package on top of a ML-like language.

This language is very young and many problems still have to be solved. In particular, synchronous data-flow networks with dynamical (recursive) process creation (like the celebrated Eratosthenes sieve) are not yet fully taken into account in the compiler, though many elements for handling it correctly have already been provided (it remains to find convenient rules for avoiding the creation of an infinite number of useless recursive processes). Also, 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).

Otherwise 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 [14, 16] 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.

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

[7] 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.

[8] 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.

[9] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 1997. to appear, available at `http://www.cs.kun.nl/ bart/`.

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

[11] X. Leroy. The caml light system : release 0.7 : documentation and user's manual. Technical report, INRIA, 1995.

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

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

[14] 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.

[15] 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.

[16] 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/`.

[17] 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.

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

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

A Some language extensions

A.1 Resetting a function

`reset` takes a stream function `f` as its first argument, a boolean stream `c` and the input `x` of the function. `reset f c x` returns the stream `f x` while `c` is false and resets the states of `f` when `c` is true, etc. Its definition is:

```
reset (Co f fi) (Co c ci) (Co x xi) =
Co (\ (r,fs,cs,xs)->let (vf, fs') = f fs
                        (vc, cs') = c cs
                        (vx, xs') = x xs
                    in if vc
                       then let (vf', r') = vf vx Nil
                            in (vf', (r',fs',cs',xs'))
                       else let (vf', r') = vf vx r
                            in (vf', (r',fs',cs',xs'))
   ) (Nil,fi,ci,xi)

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

The state of `f` is re-initialized by sending `Nil` to the function, each time the condition `c` is true. This ability of resetting a function in a modular way (without knowing the code of the function to be reset) is not provided in LUSTRE. We have already stressed elsewhere [4] the interest of such a function.

A.2 Abstract data types

Another language extension, which can be forecast, concerns abstract data types. Abstract data types raise no problems at the scalar level; given a type definition:

```
data Type x y...= C1 t1,t1'... | C2 t2, t2'... |...Cn tn, tn'...
```

we can always lift this type to the stream level by lifting both constructors and destructors:

```
Csi  = extend (const Ci)
Case = extend (const \x.case x of C1 x1, x1... ..-> e1
                                  ....
                                  Cn xn, xn'...  -> en)
```

However this choice would be quite awkward and hardly consistent with the choice made of lifting any language construct to the stream level. This, in turn, raises a clock problem. What are the clock constraints of expressions such as

$$\text{CONST}\ \frac{H \vdash e_1\ : cl \quad H \vdash e_2\ : cl \ldots}{H \vdash C_1\ e_1\ e_2 \ldots : cl}$$

$$\text{CASE}\ \frac{\begin{array}{l} H, x_1 : cl \ \mathtt{on}\ C_1?\ e, x'_1 : cl \ \mathtt{on}\ C_1?e \ldots \vdash e_1\ : c_l \ \mathtt{on}\ C_1?e \\ \ldots \\ H, x_n : cl \ \mathtt{on}\ C_n?e, x'_n : cl \ \mathtt{on}\ C_n?e \ldots \vdash en\ : cl \ \mathtt{on}\ C_n?e \end{array}}{\begin{array}{ll} H \vdash \mathtt{case}\ e\ \mathtt{of} & C_1 x_1, x'_1 \ldots - > e_1 \\ & \ldots \\ & C_n x_n, x'_n \ldots - > e_n\ :\ cl \end{array}}$$

Figure 5: Clock calculus for abstract data types

```
case e of C1 x1, x1...  -> e1
          ....
          Cn xn, xn'... -> en
```

and `C1 e`?

This problem can be solved thanks to the two rules shown at figure 5, where, given a data type constructor `C`, the boolean condition `C?` is defined as:

```
C? (Co e ie) = Co \s-> case e s
                        of E,s'    -> E,s'
                           (V v),s'->case v
                                     of C x,x'... -> (V True) ,s'
                                        otherwise -> (V False),s'
                  ie
```

B Proof of proposition 1

i) It is easy to check that

`id' = const (\ve->\sfe-> P ve sfe)`

yields `apply id' (Co te se) = Co te se`:

```
co_apply id' (Co te se) =
        Co \(sfe,sf,se) ->
             let ve, se' = te se
             in ve, (sfe,Nil,se')
            (Nil,Nil,se)
```

The two first components of the state are stuck to `Nil` and can be withdrawn, yielding the announced result.

ii) Let us define:

```
compose (Co tf' sf') (Co tf sf) =
        Co \(sf',sf)->
              (\ve sf'fe ->
                 let vf', sf'''  = tf' sf'
                     vf, sf''    = tf sf
                     (sfe, sf'e) = case sf'fe
                                   of Nil           -> Nil,Nil
                                      St (sfe,sf'e) -> sfe, sf'e
                     vfe, sfe'    = vf ve sfe
                     vf'fe, sf'e' = vf' vfe sf'e
                 in  vf'fe, (St (sfe', sf'e'))), (sf'',sf''')
           (sf, sf')
```

on the one hand, we have:

```
co_apply (Co tf' sf') (co_apply (Co tf sf) (Co te se)) =
         Co \(sf'e,sf',(sfe,sf,se)) ->
               let vf', sf''' = tf' sf'
                   vf, sf'' = tf sf
                   ve, se' = te se
                   vfe, sfe' = vf ve sfe
                   vf'fe, sf'e' = vf' vfe sf'e
               in vf'fe, (sf'e',sf''',(sfe',sf'',se'))
           (Nil,sf',(Nil,sf,se))
```

On the other hand we have:

```
co_apply (compose (Co tf' sf') (Co tf sf)) (Co te se)) =
         Co \(sf'fe,(sf',sf),se) ->
               let vf', sf''' = tf' sf'
                   vf, sf''   = tf sf
                   ve, se'    = te se
                   sfe, sf'e  = case sf'fe
                                of Nil           -> Nil,Nil
                                   St (sfe,sf'e) -> sfe, sf'e
                   vfe, sfe' = vf ve sfe
                   vf'fe, sf'e' = vf' vfe sf'e
               in  vf'fe, ((St (sfe', sf'e')), (sf'',sf'''),se')
            (Nil,(sf',sf), se))
```

Applying the first state component change:

```
Nil            -> (Nil,Nil)
(St (sfe,sf'e))-> sfe, sf'e
```

and reordering the state yields the desired result.

C Proof of proposition 2

Taking `f'` as `Co tf sf`, this amounts to compute
`co_apply (Co tf sf) (Co (\se ->(ve,Nil)) Nil)` and to substitute the resulting coiteration to `Co tfe sfe0` in the expression of `Co_lambda f`:

```
co_apply (Co tf sf) (Co (\se ->(ve,Nil)) Nil)   =
         Co \(sfe,sf,se) ->
               let vf, sf' = tf sf
                   vfe, sfe' = vf ve sfe
               in vfe, (sfe',sf',Nil)
             (Nil,sf,Nil)
```

Noting that the third component of the state remains always equal to `Nil`, we can take:

```
tfe = \(sfe,sf) ->
               let vf, sf' = tf sf
                   vfe, sfe' = vf ve sfe
               in  vfe, (sfe',sf')

sfe0 =  (Nil,sf)
```

yielding:

```
co_lambda (co_apply (Co tf sf)) =
          Co \sf ->
               (\ve sfe->
                  let (sfe1,sf1) = case sfe
                                   of Nil          -> Nil,sf
                                      (St (sfe,sf))-> sfe,sf
                      vf, sf' = tf sf1
                      vfe, sfe' = vf ve sfe1
                  in  vfe, (St (sfe',sf'))),
                  sf
             Nil
```

Now we apply this result `f''` so as to compute `co_apply f'' (Co te se)`:

```
co_apply f'' (Co te se)=

         Co \(sfe,sf,se) ->
               let vf, sf' =
                     (\ve \sfe->
                        let sfe1,sf1 = case sfe
                                       of Nil       -> Nil,sf
                                         St (sfe,sf)-> sfe,sf
                            vf, sf' = tf sf1
                            vfe, sfe' = vf ve sfe1
                        in  vfe, (St (sfe',sf'))),
                        sf
                   ve, se' = te se
                   vfe, sfe' = vf ve sfe
               in  vfe, (sfe',sf',se')
             (Nil,Nil,se)

co_apply f'' (Co te se)=
         Co \(sfe,sf,se) ->
               let ve, se' = te se
                   vfe, sfe' =
                      let sfe1,sf1 = case sfe
                                     of Nil         -> Nil,sf
                                        St (sfe,sf) -> sfe,sf
                          vf, sf'   = tf sf1
                          vfe, sfe' = vf ve sfe1
                      in vfe, (St (sfe',sf'))
               in vfe, (sfe',sf, se')
            (Nil,Nil,se)


co_apply f'' (Co te se)=
         Co \(sfe,sf,se) ->
               let ve, se' = te se
                   sfe1,sf1 = case sfe
                              of Nil        -> Nil,sf
                                 St (sfe,sf)-> sfe,sf
                   vf, sf' = tf sf1
                   vfe, sfe' = vf ve sfe1
               in  vfe, (St (sfe',sf'),sf, se')
             (Nil,Nil,se)
```

Here also we can get rid of the state second component which remains stuck to `Nil`:

```
co_apply f'' (Co te se)=

         Co \(sfe,se) ->
               let ve, se' = te se
                   sfe1,sf1 = case sfe
                               of  Nil         -> Nil,sf
                                   St (sfe,sf) -> sfe,sf
                   vf, sf' = tf sf1
                   vfe, sfe' = vf ve sfe1
               in  vfe, (St (sfe',sf'), se')
           (Nil,se)
```

Expanding the definition of the state first component:

```
  Nil          -> Nil,sf
  St (sfe,sf)  -> sfe,sf
```

yields:

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

which is the announced result.

D Proof of proposition 3

Given:

```
co_rec (Co tf sf) = Co \(se,sf) ->
                         let vf, sf' = tf sf
                             ve, se' = vf ve se
                         in  ve, (se', sf')
                       (Nil,sf)
```

we want to compute:

```
co_apply (Co tf sf) (co_rec (Co tf sf)) =
         Co \(sfe,sf,(se, sf1)) ->
               let vf, sf'   = tf sf
                   vf1, sf1' = tf sf1
                   ve, se'   = vf1 ve se
                   vfe, sfe' = vf ve sfe
               in  vfe, (sfe',sf', (se', sf1'))
             (Nil,sf,(Nil,sf))
```

it is then easy to prove that `sfe' = se' and sf' = sf1'` is an invariant as it is true initially and

```
sfe= se and sf = sf1 implies sfe'= se' and sf'= sf1'
```

Thus the system simplifies to:

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

which is the announced result.