Synchronous Kahn Networks

Paul Caspi

VERIMAG
Miniparc-ZIRST
Rue Lavoisier
38330 Montbonnot St-Martin
France
e-mail: **Paul.Caspi@imag.fr**

Marc Pouzet*

School Of Computer Science
McGill University
3486 University Street
Montréal, H3A 2A7
Canada
e-mail: **pouzet@cs.mcgill.ca**

Abstract

Synchronous data-flow is a programming paradigm which has been successfully applied in reactive systems. In this context, it can be characterized as some class of static bounded memory data-flow networks. In particular, these networks are not recursively defined, and obey some kind of "synchronous" constraints *(clock calculus)*. Based on Kahn's relationship between data-flow and stream functions, the synchronous constraints can be related to Wadler's listlessness, and can be seen as sufficient conditions ensuring listless evaluation. As a by-product, those networks enjoy efficient compiling techniques. In this paper, we show that it is possible to extend the class of static synchronous data-flow to higher order and dynamical networks, thus giving sense to a larger class of synchronous data-flow networks.

This is done by extending both the synchronous operational semantics, the clock calculus and the compiling technique of static data-flow networks, to these more general networks.

1 Introduction

1.1 Some milestones in data-flow programming

In the seventies, the Lucid language was proposed [5, 4] as a way of providing functional languages with stream based iterations. At the same time, Kahn [15] showed that the semantics of networks of asynchronous deterministic processes could be described as systems of recursive equations over streams, very similar to Lucid programs. Then, the concept of lazy evaluation emerged [12, 23], accounting for finite and infinite data structures like streams. Today, modern lazy functional languages such as LML and Haskell [6, 22] easily allow to write data-flow programs, by expressing streams as abstract data types. Such languages provide nice features, such as currying and higher order programming. Yet, these programs are sometimes rather inefficient, and Wadler [25] proposed new techniques, which he called "listlessness" [25] so as to try to overcome the problem. The idea was to avoid constructing intermediate lists in list programs, if the elements of those lists were to be consumed as soon as produced. Later, he generalized this technique to other recursive data types, and called it "deforestation" [26].

*This work has been supported by an INRIA fellowship.

1.2 Synchronous reactive data-flow

Meanwhile, automatic control, and signal processing engineers faced the problem of moving from analog devices to sequential computers. They have always used systems of recursive equations over streams of values (signals) as a natural formalism for reasoning about their systems. They often found translation of these equations into sequential programs as boring and error-prone. Some of them found that it was possible to handle this translation automatically, that is, to compile such programs. For instance, some of the programs running on the Airbus A320 aircraft have been obtained in this way. This led computer scientists to propose languages (LUSTRE, SIGNAL), toolboxes (PTOLEMY) and compilers for application domain [2, 20, 16] which they called "Synchronous Data-flow" (By the same time, this class of programs was recognized to belong to the so-called "synchronous programming" family, whose most characteristic member is ESTEREL [7]).

These compilers [21] behave very much like Wadler's listless transformer, i.e. associate with each lazy stream of a given type, only one item of this type. This is obtained by :

- defining synchronous operational semantics, which allows only listless programs to be evaluated [21],
- providing a static analysis step, aiming at rejecting data-flow networks that cannot be listlessly evaluated. This step is sometimes referred to as either a "clock calculus"[20, 2] or "consistency checking"[16].
- finally, providing compiling techniques which product listless sequential code for programs that have been accepted during the preceding step.

1.3 Synchronous data-flow

However, those synchronous data-flow languages are closely restricted to the domain of reactive systems; for instance, they don't allow the use of recursive definitions of functions. The reasons for these restrictions are quite clear : reactive systems shall continuously interact with their environment, and this can be safely achieved only if their reactions are implemented using bounded memory and bounded reaction time.

The purpose of the paper is to show that the synchronous operational semantics, and the clock constraint rules are not bounded to these reactive restrictions and can apply to more general stream languages (section 3.1) providing abstraction, application and general recursion.

For this, we intend to establish a clear distinction between "synchrony" and "reactivity":

- Reactivity means the ability to react to external events, and, quite obviously, requires bounded memory and reaction time.
- Synchrony is the ability to share a common time scale. In the context of data-flow, it can be interpreted as listlessness in view of the already noted analogy between listless transformers and synchronous data-flow compilers. Synchrony has little to do with reactivity, though it can be useful in order to achieve reactivity. This is why, up to now, these two concepts have been strongly connected.

Thus, we can imagine non reactive synchronous programs. This is the intended use of the proposed extension which allows us to give a synchronous meaning to higher order and dynamical (recursively defined) networks. In particular, truly recursive networks will use unbounded stacks, and thus cannot be reactive. Since we deal with data-flow networks, these stacks are expected to be stacks of lists. Synchrony here means simply that it is possible to replace those stacks with stacks of values (cf example 6), and this is expected to be far more efficient.

1.4 Paper content

The paper will be organized as follows. In section 2 we present a data-flow language based on a LUSTRE-like toy language, i.e. a purely functional [1] synchronous reactive data-flow language. Yet, the language will deviate from LUSTRE in several aspects:

- The primitive constructs will be slightly different and somewhat more general: our primitives allow easily LUSTRE primitives to be expressed.
- The clock calculus is more general in the sense that it allows clocks to be inferred. This helps in extending it to functional features, thanks to the analogy with classical type inference.
- For the same reason, we use a curryfied version of the language.

Section 3 presents the core of the extension work by:

- defining a synchronous operational semantics for a functional data-flow language with abstraction, application, recursion and using the data-flow primitives defined in section 2. This allows us to characterize "synchronous data-flow behaviors" (section 3.2),
- defining a clock calculus for this language: this is obtained by expressing this clock calculus as a type system, which, in turn, allows us to generalize it to functional features (section 3.3),
- showing that programs that can be typed in the preceding sense, can be synchronously evaluated (theorem 1).
- and finally, giving a system of modular compiling rules producing imperative sequential code obeying synchronous semantics (section 3.4).

Section 4 presents implementation matters and section 5 discusses related works.

2 A data-flow language

2.1 Primitive constructs

In this section, we define some primitive constructs over lazy streams in a Haskell-like syntax [22].

- Lazy streams of type a can be defined as:

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

- The `const`primitive allows scalar constants to be transformed into infinite constant streams, by :

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

  Where "`:`"is the ordinary stream constructor. For instance `const true` is the infinite stream:

  $$[\texttt{true}; \texttt{true}; \ldots]$$

- `extend` allows streams of functions to be applied to streams of data :

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

  For instance,

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

  respectively define an inverter and an "and" gate which operate pointwisely over their input streams; thus `notl (const true)` is the infinite stream:

  $$[\texttt{false}; \texttt{false}; \ldots]$$

- `fby` is the *followed-by* operator (or *delay* operator):

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

  It allows recursive expressions (feedback networks or circuits) to be safely built, without exhibiting deadlocks (if the recursion appears on the right of the `fby`); for instance:

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

  has the infinite periodic behavior :

  $$[\texttt{false}; \texttt{true}; \texttt{false}; \texttt{true}; \ldots]$$

- When the first argument of `fby` is a constant stream, we rather use the simpler function `pre`, whose first argument is a scalar:

[1] Though presenting some analogy with Signal, this functional aspect makes it very different, as Signal is *not* a functional language.

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

Thus we could have written:

```
half = pre false (notl half)
```

- The `when` primitive allows sub-streams to be extracted from streams:

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

For instance, we have:

$$\begin{array}{lcl} \texttt{x} & = & [x_0; x_1; \ldots x_n; \ldots] \\ \texttt{x when half} & = & [x_1; x_3; \ldots x_{2n+1}; \ldots] \end{array}$$

- Conversely, `merge` allows streams to be built from sub-streams :

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

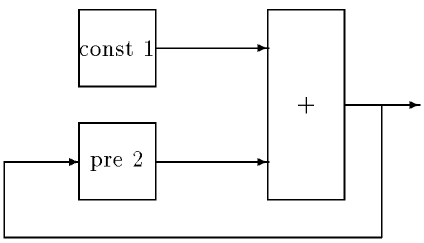
For instance, we have:

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

Example 1 (A program) A program is a set of mutually recursive equations defining streams of scalar values. Consider the following expression:

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

It defines the list of positive integers starting from 2. This program has the following Kahn network representation.



□

Lazy evaluating this program is costly: intermediate lists are allocated and deallocated by the Garbage Collector during execution. On the contrary, the synchronous data-flow compilers translate it into a sequential program with bounded memory and response time. One could say that those compilers transforms the call-by-need evaluation into a call-by-value one.

2.2 Why synchrony?

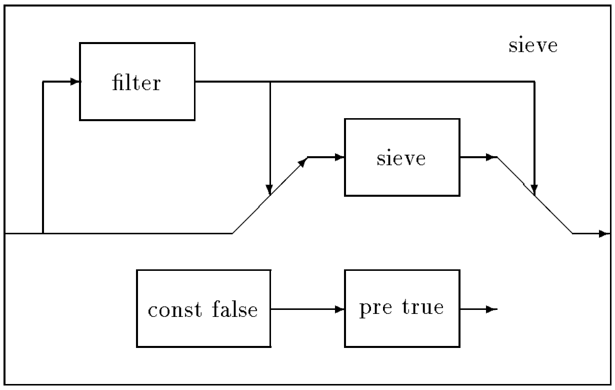
Let us consider the example of figure 1 which displays a program, the corresponding network, and a typical evolution of the streams involved in the program.

In this example, the same input is duplicated and one version goes through an `odd` function[2], whose effect is to discard one item out of two from its input stream. Then both streams go through an `andl` function, which consumes one item of each stream at a time. Clearly, such a network cannot be executed without using an intermediate list. Furthermore, as time goes on, this storage grows, and will sooner or later overflow. This explains why this kind of example, which is called "non synchronous" has to be rejected when dealing with reactive applications.

2.3 Recursive functions

For the same obvious reasons of reactivity, actual synchronous data-flow tools reject recursively defined functions. A typical example of such a rejected program is the following Eratosthenes sieve, whose recursive data-flow network is depicted below.

In this figure, data flow from left to right. For the first input item, the filter records the input value and returns the value false. In this case, the switches are low and the returned value is true. For the next inputs, the filter box returns true when the current entry is not a multiple of its recorded value. In this case, the switches are high. The first time this happens, a new sieve machine is created and fed with an initial input. The next times this happens, this machine is fed with the corresponding inputs. Otherwise, the switches are low and the returned value is always false.



Such a data-flow graph can be represented in a functional and recursive way as follows:

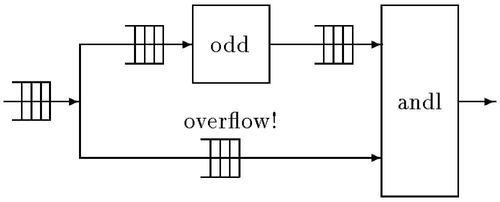
```
let first x =
      let v = x fby v
      in  v

    sieve x =
       let filter = (first x) not_divides_l x
       in  merge filter
                (sieve (x when filter))
                (pre true (const  false))

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

[2] In terms of control theory and hardware x when half is a half-frequency sampler.

```
let   odd   x =
      let   half = pre false (notl half)
      in    x when half
in
            x andl (odd x)
```



`x`	=	x_0	x_1	x_2	x_3	x_4	x_5	...
`half`	=	`false`	`true`	`false`	`true`	`false`	`true`	...
`x when half`	=		x_1		x_3		x_5	...
`x andl (odd x)`	=		x_0 and x_1		x_1 and x_3		x_2 and x_5	...

Figure 1. A non-synchronous example

where `not_divides_l` is the extension to streams of the ordinary operation on integers.

Yet, it seems to us that this example doesn't exhibit the pathological character of the previous non-synchronous one, and that it seems possible to compile it using no intermediate list at all. Yet, since this is a truly recursive function, its compilation should use a stack, but this is not a surprise and we are used to it. Though the theoretical distinction between stacks and lists should deserve greater attention, we propose to leave it beyond the scope of the paper: we only intend to show here that some recursive stream programs appear as reasonable extensions of what has been done in the field of synchronous reactive data-flow, and enjoy the same aptitude to efficient listless compiling. We propose to call such programs "synchronous Kahn networks".

3 Formalization

3.1 A functional stream language

Let us define the following functional and recursive data-flow kernel. Expressions are ranged over e and are built from stream variables (x), primitive constructs presented above, recursive expressions (*rec* $x.e$), abstractions ($\lambda x.e$) and applications ($e_1 e_2$). The language contains scalar primitives which can be extended to streams via the `const` primitive. (i) denotes integers, (`true`) and (`false`) denote classical true and false constants. Classical primitives are available such as (+) for the integer addition, (not) and (and) for classical boolean operators, etc.

$$
\begin{array}{rcl}
e & ::= & x \mid \texttt{rec}\ x.e \mid \lambda x.e \mid e\ e \\
& & \mid \texttt{const}\ e \mid \texttt{merge}\ e\ e\ e \mid e\ \texttt{fby}\ e \\
& & \mid \texttt{extend}\ e\ e \mid \texttt{pre}\ e\ e \mid e\ \texttt{when}\ e \mid \texttt{notl}\ e \\
& & \mid + \mid \textsf{and} \mid \textsf{not} \mid i \mid \texttt{false} \mid \texttt{true} \mid \ldots
\end{array}
$$

This language is a typed language in the sense of the classical Damas-Milner type system [11]. We say that an expression e is of scalar type if all sub-expression (including e) is of scalar type and does not use the stream primitives. The function *scalar* is such that *scalar*(e) is true if e is of scalar type.

Thus, our language is a classical functional language with special primitives whose semantics have been defined previously.

3.2 Synchronous operational semantics

Programs written in such a language can be executed in a call-by-need manner, as programs managing lists, thus leading to a poor implementation: intermediate structures are allocated and then deallocated by the Garbage Collector. We shall see now that a large set of programs from this language can be executed in a *synchronous* way, where no list is managed during the execution. The idea behind synchrony is to execute one step of each subexpression — one step of each node in the network — such that no intermediate structure is buffered in a node: an argument crosses the whole network in one step. Thus, the *synchronous operational semantics* is a restricted relation of the classical lazy semantics: some reductions rules are not allowed.

Definition 1 (Synchronous Semantics) *The synchro nous operational semantics of the language is defined by the transition relation $\sigma \vdash e \xrightarrow{v} e'$ meaning that in the environment σ, one execution step of the expression e produces v and becomes the expression e'.*

$$
\begin{array}{rcl}
\sigma & ::= & [x_1 \xrightarrow{v_1} y_1, \ldots, x_n \xrightarrow{v_n} y_n] \\
v & ::= & [\,] \mid e
\end{array}
$$

x_i and y_i denote variable names and v is either a scalar expression e or empty (denoted by $[\,]$). An execution of e_0 starting in the environment σ_0 can be written as:

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

if $\sigma_0 \vdash e_i \xrightarrow{v_i} e_{i+1}$. The value produced by the expression e_0 is the infinite list $v_0.v_1 \ldots v_n \ldots$ We assume that $[\,]$ is the unit element of list construction ($[\,].l = l$).

The substitution $e[v_1/v_2]$ used here is quite unusual since v_2 may contain somes empty constructs. Thus, the substitution does some pattern matching, and is defined only when the two components are compatible.

The complete definition of $\sigma \vdash e \xrightarrow{v} e'$ is given by the set of rules at figure 2.

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

$$\text{CONST-e}\quad \frac{}{\sigma \vdash \texttt{const}\ e \xrightarrow{e} \texttt{const}\ e}$$

$$\text{CONST-[]}\quad \frac{}{\sigma \vdash \texttt{const}\ e \xrightarrow{[]} \texttt{const}\ e}$$

$$\text{EXT-[]}\quad \frac{\sigma \vdash f \xrightarrow{[]} f_1 \quad \sigma \vdash e \xrightarrow{[]} e_1}{\sigma \vdash \texttt{extend}\ f\ e \xrightarrow{[]} \texttt{extend}\ f_1\ e_1}$$

$$\text{EXT-v}\quad \frac{\sigma \vdash f \xrightarrow{f_0} f_1 \quad \sigma \vdash e \xrightarrow{e_0} e_1}{\sigma \vdash \texttt{extend}\ f\ e \xrightarrow{f_0 e_0} \texttt{extend}\ f_1\ e_1}$$

$$\text{MERGE-[]}\quad \frac{\sigma \vdash c \xrightarrow{[]} c_1 \quad \sigma \vdash d \xrightarrow{[]} d_1 \quad \sigma \vdash e \xrightarrow{[]} e_1}{\sigma \vdash \texttt{merge}\ c\ d\ e \xrightarrow{[]} \texttt{merge}\ c_1\ d_1\ e_1}$$

$$\text{MERGE-t}\quad \frac{\sigma \vdash c \xrightarrow{\texttt{true}} c_1 \quad \sigma \vdash d \xrightarrow{d_0} d_1 \quad \sigma \vdash e \xrightarrow{[]} e_1}{\sigma \vdash \texttt{merge}\ c\ d\ e \xrightarrow{d_0} \texttt{merge}\ c_1\ d_1\ e_1}$$

$$\text{MERGE-f}\quad \frac{\sigma \vdash c \xrightarrow{\texttt{false}} c_1 \quad \sigma \vdash d \xrightarrow{[]} d_1 \quad \sigma \vdash e \xrightarrow{e_0} e_1}{\sigma \vdash \texttt{merge}\ c\ d\ e \xrightarrow{e_0} \texttt{merge}\ c_1\ d_1\ e_1}$$

$$\text{WHEN-[]}\quad \frac{\sigma \vdash c \xrightarrow{[]} c_1 \quad \sigma \vdash d \xrightarrow{[]} d_1}{\sigma \vdash d\ \texttt{when}\ c \xrightarrow{[]} d_1\ \texttt{when}\ c_1}$$

$$\text{WHEN-t}\quad \frac{\sigma \vdash c \xrightarrow{\texttt{true}} c_1 \quad \sigma \vdash d \xrightarrow{d_0} d_1}{\sigma \vdash d\ \texttt{when}\ c \xrightarrow{d_0} d_1\ \texttt{when}\ c_1}$$

$$\text{WHEN-f}\quad \frac{\sigma \vdash c \xrightarrow{\texttt{false}} c_1 \quad \sigma \vdash d \xrightarrow{d_0} d_1}{\sigma \vdash d\ \texttt{when}\ c \xrightarrow{[]} d_1\ \texttt{when}\ c_1}$$

$$\text{PRE-[]}\quad \frac{\sigma \vdash e \xrightarrow{[]} e_1}{\sigma \vdash \texttt{pre}\ v\ e \xrightarrow{[]} \texttt{pre};\ v\ e_1}$$

$$\text{PRE-v}\quad \frac{\sigma \vdash e \xrightarrow{e_0} e_1}{\sigma \vdash \texttt{pre}\ v\ e \xrightarrow{v} \texttt{pre}\ e_0\ e_1}$$

$$\text{FBY-[]}\quad \frac{\sigma \vdash e \xrightarrow{[]} e_1 \quad \sigma \vdash f \xrightarrow{[]} f_1}{\sigma \vdash e\ \texttt{fby}\ f \xrightarrow{[]} e_1\ \texttt{fby} f_1}$$

$$\text{FBY-init}\quad \frac{\sigma \vdash e \xrightarrow{e_0} e_1 \quad \sigma \vdash f \xrightarrow{[]} f_1}{\sigma \vdash e\ \texttt{fby}\ f \xrightarrow{[]} \texttt{pre}\ e_0\ f_1}$$

$$\text{FBY-pre}\quad \frac{\sigma \vdash e \xrightarrow{e_0} e_1 \quad \sigma \vdash f \xrightarrow{f_0} f_1}{\sigma \vdash e\ \texttt{fby}\ f \xrightarrow{e_0} \texttt{pre}\ e_0\ f_1}$$

$$\text{REC}\quad \frac{\sigma, x \xrightarrow{x_0} x_1 \vdash e \xrightarrow{v_0} e_1}{\sigma \vdash rec\ x.e \xrightarrow{rec\ x_0.v_0} rec\ x_1.e_1[rec\ x_0.v_0/x_0]}$$

$$\text{ABST}\quad \frac{\sigma, x \xrightarrow{x_0} x_1 \vdash e \xrightarrow{v_0} e_1}{\sigma \vdash \lambda x.e \xrightarrow{\lambda x_0.v_0} \lambda x_0.\lambda x_1.e_1}$$

$$\text{APP}\quad \frac{\sigma \vdash f \xrightarrow{f_0} f_1 \quad \sigma \vdash e \xrightarrow{v_0} e_1}{\sigma \vdash f\ e \xrightarrow{f_0\ v_0} f_1\ v_0\ e_1}$$

$$\text{abst}\quad \frac{\sigma \vdash e \xrightarrow{v_0} e_1 \quad scalar(x)}{\sigma \vdash \lambda x.e \xrightarrow{\lambda x.v_0} \lambda x.e_1}$$

$$\text{app}\quad \frac{\sigma \vdash f \xrightarrow{f_0} f_1 \quad scalar(v)}{\sigma \vdash f\ v \xrightarrow{f_0\ v} f_1\ v}$$

Figure 2. The synchronous operational semantics

- `const` produces a list of constants. Thus one step of the $\texttt{const}\, v$ program produces v and the continuation $\texttt{const}\, v$. Such an operation may produce nothing if no operation is waiting for its value. In this case, we call it an "empty rule".

- The `extend` primitive applies pointwisely a list of scalar functions to a list of arguments. The semantics states that both arguments must be either present or absent in parallel. If none of them is present, the code remains the same and produces no value.

  Remark: The synchronous aspect of these semantics lies in the absence of some rules: if we take into account all the cases of production and non production of values of the arguments, we should consider four cases. Here, we have only two. Thus, there is no way of evaluating the expression when only one argument is present nor there is a way of storing intermediate results in temporary lists. This is *the key difference* with respect to classical operational semantics.

- Similarly, `merge` should need eight rules, but synchronous semantics consider only three of them.

- `fby` may produces a value only when its first argument produces a value. In this case, it transforms into a `pre`operation.

- Operationally, a `pre` operator acts as a latch: it puts its recorded value on the output and stores its input.

- The (REC) rule is very natural: yet, in doing this, we must carefully distinguish the cases where *rec* applies to ground expressions and those where it applies to functional expressions: in the former case, (the only one which arises in reactive data-flow) it is allowed to use fix-points like $rec\, x_0.e_0$ only if x_0 is not free in e_0. Otherwise, this should be considered as a deadlock. Consider, for example, the program $rec\, x.(plus\; x\; (\texttt{const}\, 1))$ where *plus* stand for the classical addition extended to streams. The produced scalar expression is $recx.x + 1$ which infinitely loops. Yet, static deadlock detection is fairly easy since it suffices to show that each variable bound by a *rec*is within the scope of a `pre` or `fby` (consider example 1).

- The rule of abstraction over streams (rule ABST) says that a $\lambda x.f$ expression produces a scalar function $\lambda x_0.f_0$ and returns a new function $\lambda x_0.\lambda x_1.f_1$ which can be both a function of the instant value and of the continuation of its argument. For instance, this arises when the instant value modifies the state of the function (e.g, a (recursive) function containing some `fby` or `pre` primitives). When an abstraction is over a scalar value (rule abst) then the execution may produce a new abstraction if the body produces something new. Application rules match abstraction rules: an application $f\, e$ over streams produces an application $f_0\, e_0$ over scalars and the continuation $(f_1\, e_0)\, e_1$. Thus, next returned values by the application may depend on e_0.

Example 2 (A synchronous program) Let us consider again example 1 that computes the set of positive integers and let us write it (for the sake of clarity) as follows:

$$\texttt{rec}\; x.\texttt{pre}\, 0\; (x + 1)$$

where we assume that 1 denotes the infinite stream of 1 and + is the classical addition extended to lists.

One execution step of this program is represented in the following proof tree:

$$\dfrac{\dfrac{\dfrac{x \xrightarrow{x_0} x' \vdash x \xrightarrow{x_0} x' \quad x \xrightarrow{x_0} x' \vdash 1 \xrightarrow{1} 1}{x \xrightarrow{x_0} x' \vdash x + 1 \xrightarrow{x_0+1} x' + 1}}{x \xrightarrow{x_0} x' \vdash \texttt{pre}\, 0\, (x+1) \xrightarrow{0} \texttt{pre}\, (x_0+1)\, (x'+1)}}{\texttt{rec}\; x.\texttt{pre}\, 0\, (x+1)}$$

$$\vdash \qquad \xrightarrow{rec\, x_0.0}$$

$$\texttt{rec}\; x'.\texttt{pre}\, (x_0+1)\, (x'+1)[rec\, x_0.0/x_0]$$

The resulting expression reduces to the following one:

$$\vdash \texttt{rec}\; x.\texttt{pre}\, 0\, (x+1) \xrightarrow{0} \texttt{rec}\; x'.\texttt{pre}\, 1\, (x'+1)$$

It is quite easy to see that the program produces the stream of integers. Moreover, this program is *reactive*: the program needs bounded memory and bounded response time. This is due to the fact that the size of the expression and the size of the execution proof are bounded. This can be seen as a reactivity property. In particular, this property is verified in synchronous static data-flow. □

Within the framework, most functions can be executed synchronously. Yet, some functions cannot.

Example 3 (A non synchronous example) Let us come back to the non synchronous program given at figure 1. We saw previously that the set of buffered values of x increased during execution. We can verify that there is no possible execution step of this expression with the given synchronous operational semantics. For the sake of clarity, we prefer to use directly `andl` instead of its complete definition.

Fact: There is no x_0 and expression e such that $\sigma \vdash x \xrightarrow{x_0} x_1$ and $\sigma \vdash x\, \texttt{andl}\, (x\, \texttt{when}\, half) \xrightarrow{v_0} e$.

Proof: Suppose we have a transition. The first step in the proof would be :

$$\dfrac{\sigma \vdash x \xrightarrow{x_0} x' \quad \dfrac{\sigma \vdash x \xrightarrow{x_0} x' \quad \sigma \vdash half \xrightarrow{e} half'}{\sigma \vdash (x\, \texttt{when}\, half) \xrightarrow{y_0} x'\, \texttt{when}\, half'}}{\sigma \vdash x\, \texttt{andl}\, (x\, \texttt{when}\, half) \xrightarrow{x_0\, \texttt{and}\, y_0} x'\texttt{andl}\, x'\, \texttt{when}\, half'}$$

This is possible only if $\sigma \vdash half \xrightarrow{\texttt{true}} half'$ which does not hold. Thus, there is no possible execution of the program. □

The adequacy of the synchronous operational semantics with the classical lazy semantics is straightforward: the synchronous semantics is a subset of the lazy semantics.

3.3 Clock Calculus

The idea of the LUSTREclock 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 [21, 9], in the sense that it infers the clock and uses unification instead of fix-points. Moreover, the clock calculus is extended to functional expressions.

Definition 2 (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 is such that:

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

A clock cl is either a clock variable α, or a sub-clock of a clock, cl on *monitored by some boolean stream expression e, or a clock function. Clock expressions are decomposed into clock schemes (σ) and clock instances (cl).*

$$\begin{array}{lll} cl & ::= & \alpha \mid cl \text{ on} e \mid cl \xrightarrow{x} cl \\ \sigma & ::= & cl \mid \forall \alpha_1 \dots \alpha_n . cl \end{array}$$

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

$$\begin{array}{ll} Gen_H(cl) = \forall \alpha_1 ... \alpha_n . cl & \text{if } \alpha_1, ..., \alpha_n \notin FV(H) \\ & \alpha_1, ..., \alpha_n \in Left(cl) \end{array}$$

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

The axioms and inference rules of the clock system are given at figure 3.

The clock system has been done in the spirit of the classical Damas–Milner type system [11] with a slight modification of the recursion rule, coming from [19]. Nonetheless, it is an unconventional system since clocks contain expressions.

- A constant expression matches any clock. Thus, it has a polymorphic clock $\forall \alpha . \alpha$.
- The clocks of the two arguments of an `extend` must be identical.
- The clock of a `when` expression depends on the values of the second argument. This is represented using the on construction.
- 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. We also have a symmetrical rule.
- A `fby` expression is well clocked when, either the clock of the two arguments are the same, or the clock of the first is faster than the second.
- The `pre` expression preserves the clock of its argument.
- In abstracting over stream variables, we must keep trace of the variable being abstracted, as it can appear in clock expressions.
- 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. Abstraction and application over scalar variables don't modify clocks.
- A variable in a clock expression can be generalized if it is free in the environment H and if it appears on the left of a functional clock. This constraint is unusual but has an intuitive explanation: in general, the clock of an expression can never 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. The instantiation rule is the classical one for type systems.
- 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.

This allows us to state the main result of this section, namely that every clockable expression can be synchronously executed.

Theorem 1 (Soundness) *For all e and cl, if $\vdash e : cl$ then it exists v e' and cl' such that*

- *i)* $\vdash e \xrightarrow{v} e'$, *and*
- *ii)* $\vdash e' : cl'$.

Proof: The proof will be roughly sketched as follows: to each clock proof one can always associate at least one execution tree by building a morphism ϕ from clock proof nodes to execution proof nodes, which preserves some consistency property, namely that all expressions sharing the same clock should either yield a value or evaluate to "empty", and any sub-clocked expression should yield a value if and only if its clock evaluates to `true`Furthermore, it is easy to check that the inference axioms and rules of synchronous operational semantics preserve clockability. □

This theorem says that when an expression can be clocked, then, it can undergo a synchronous execution step, and then rewrites as a clockable expression. Thus, execution can proceed.

3.4 Compilation

The goal, here, is to transform a well clocked (thus synchronous) program over streams into an (efficient) transition function over scalars. Whereas the initial program has to be executed in a call-by-need manner, the final program will be executed in a call-by-value manner.

This compilation process is based on the synchronous operational semantics: indeed, a general execution of e is $e = e_0 \xrightarrow{v_0} e_1 \xrightarrow{v_1} e_2 \xrightarrow{v_2} \dots$ stating that one execution step of e_0 produces the value v and e_0 is transformed into e_1, etc. The goal of the compilation of e is to produce a transition function — which can be seen as one execution step of a machine — whose iteration will produce the successive values v_i. The transition can be decomposed into two parts: a *code* function producing the scalar output (v_i) and a *modif-state* function modifying some internal *memory* (or state) in the machine (in order to express the fact that e_i becomes e_{i+1}). A *memory* is a set of links between values and names. Thus, we shall construct a general function *trans*:

$$trans() \stackrel{\triangle}{=} \begin{array}{l} \textit{let v=code(s)} \\ \textit{in s:=modif-state(s);v} \end{array}$$

$$\text{CONST} \quad H \vdash \mathtt{const}\ v : \forall\alpha.\alpha$$

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

$$\text{EXT} \quad \frac{H \vdash e_1 : cl \quad H \vdash e_2 : cl}{H \vdash \mathtt{extend}\, e_1\, e_2 : cl}$$

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

$$\text{MERGE-1} \quad \frac{H \vdash e_1 : cl \quad H \vdash e_2 : cl\ \mathtt{on} e_1 \quad H \vdash e_3 : cl\ \mathtt{on}(\mathtt{notl} e_1)}{H \vdash \mathtt{merge}\, e_1\, e_2\, e_3 : cl}$$

$$\text{PRE} \quad \frac{H \vdash e : cl}{H \vdash \mathtt{pre}\, v\ e : cl}$$

$$\text{MERGE-2} \quad \frac{H \vdash e_1 : cl \quad H \vdash e_2 : cl\ \mathtt{on}(\mathtt{notl} e_1) \quad H \vdash e_3 : cl\ \mathtt{on} e_1}{H \vdash \mathtt{merge}\, (\mathtt{notl} e_1)\, e_2\, e_3 : cl}$$

$$\text{FBY-1} \quad \frac{H \vdash e_1 : cl \quad H \vdash e_2 : cl}{H \vdash e_1\, \mathtt{fby}\, e_2 : cl}$$

$$\text{FBY-2} \quad \frac{H \vdash e_1 : cl \quad H \vdash e_2 : cl\ \mathtt{one}}{H \vdash e_1\, \mathtt{fby}\, e_2 : cl\ \mathtt{one}}$$

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

$$\text{GEN} \quad \frac{H \vdash e : cl \quad \sigma = Gen_H(cl)}{H \vdash e : \sigma}$$

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

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

$$\text{abst} \quad \frac{H \vdash e : cl \quad scalar(x)}{H \vdash \lambda x.e : cl}$$

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

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

Figure 3. The clock calculus

such that each call returns a value and modifies some memory. Then, once the memory s has been properly initialized, successive calls to $trans()$ will yield the sequence $v_0, v_1, \ldots$ More formally,

Definition 3 (Compilation method) *The compilation method is defined by the relation* $\sigma \models e : \langle c, m, s\rangle$. *It means that in the environment* σ, *compiling* e *produces the code* c, *the sequence of instructions* m *modifying the memory and an initial memory* s. *These two codes are expressed in any ML-like language with side effects (the adopted syntax is close to the one of Caml-light [17]). An environment* σ *is such that:*

$$\sigma ::= [x_1 : \langle c_1, m_1, s_1\rangle, \ldots, x_n : \langle c_n, m_n, s_n\rangle]$$

A memory expression s *(or internal state) can be an empty memory* ($[]$)*, a memory name* (x)*, a set of links between names and either memories or scalar values, a recursive memory* ($rec\, x.s$) *or a function of memories* ($s \rightarrow s$).

$$\begin{array}{rcl} s & ::= & [] \mid [a_1/x_1, \ldots, a_m/x_m] \\ & & \mid x \mid rec\, x.s \mid s \rightarrow s \\ a & ::= & s \mid v \end{array}$$

This structure will be explained in more details further.

The relation $\models$ *is only defined for clockable expressions with functional order less or equal to one and where the clocks of* ($\mathtt{const}\, v$) *sub-expressions are clock variables. If* e *has order zero and* $s = [v_1/x_1; ...; v_n] \rightarrow []$, *then the final executed code can be written in the following way:*

```
def_type([v1/x1; ...; vn]);;
let trans =
  let xs = new([v1/x1; ...; vn]) in
  fun () → match xs with
           Env(x1, ..., xn) → try let v = c in m; print(v)
                              with fail → m
  in loop trans() end
```

The definition of the compilation rules, the def_type and new functions are given at figure 4.

Compilation rules follow exactly semantic rules: the code part produces the value and the *memory* part modifies a state to take into account the fact that the resulting expression has changed. Let us comment it:

- `const` returns the value of its argument. The *code* part of the compilation of $\mathtt{const}\, v$ is v. The `const`machine needs no internal state, thus nothing has to be modified. The *modif-state* part is the empty instruction $()$.
- `merge` produces a simple `if` statement. *modif-state* instructions come from the three parts of the `merge` and the *memory* part is the union of the three arguments.
- `when` produces a conditional which may raise the exception `fail` when the test is false. Here also, the *modif-state* is the composition of the two *modif-state* and the *memory* part is the union.
- Operationally, `pre` acts as "latch": it outputs its recorded value and then records its input. This is implemented by a variable. Thus, the `pre` construction

CONST $\quad \sigma \models \mathtt{const}\, v : \langle v, (), [] \rightarrow [] \rangle$ $\qquad$ TAUT $\quad \sigma, x : \langle c, m, s \rangle \models x : \langle c, m, s \rangle$

$$\text{MERGE}\quad \frac{\sigma \models e_1 : \langle c_1, m_1, s_1 \rightarrow [] \rangle \quad \sigma \models e_2 : \langle c_2, m_2, s_2 \rightarrow [] \rangle \quad \sigma \models e_3 : \langle c_3, m_3, s_3 \rightarrow [] \rangle}{\sigma \models \mathtt{merge}\, e_1\ e_2\ e_3 : \langle \mathtt{if}\ c_1\ \mathtt{then}\ c_2\ \mathtt{else}\ c_3, m_1; m_2; m_3, (s_1 \cup s_2 \cup s_3) \rightarrow [] \rangle}$$

$$\text{WHEN}\quad \frac{\sigma \models e_1 : \langle c_1, m_1, s_1 \rightarrow [] \rangle \quad \sigma \models e_2 : \langle c_2, m_2, s_2 \rightarrow [] \rangle}{\sigma \models e_1 \mathtt{when}\, e_2 : \langle \mathtt{if}\ c_2\ \mathtt{then}\ c_1\ \mathtt{else}\ \mathtt{fail}, m_1; m_2, (s_1 \cup s_2) \rightarrow [] \rangle}$$

$$\text{PRE}\quad \frac{\sigma \models e : \langle c, m, s \rightarrow [] \rangle \quad pre\_x \notin Dom(s) \quad FV(v) = \emptyset}{\sigma \models \mathtt{pre}\, v\, e : \langle !pre\_x, \begin{array}{l}\mathtt{try}\ pre\_x := c \\ \mathtt{with\ fail} \rightarrow ()\end{array}, (s \cup [v/pre\_x]) \rightarrow [] \rangle}$$

$$\text{FBY}\quad \frac{\sigma \models e_1 : \langle c_1, m_1, s_1 \rightarrow [] \rangle \quad \sigma \models \mathtt{pre}\, v\, e_2 : \langle c_2, m_2, [v/pre\_x] \cup s_2 \rightarrow [] \rangle}{\sigma \models e_1 \mathtt{fby}\, e_2 : \langle \mathtt{if}\ !init\ \mathtt{then}\ c_1\ \mathtt{else}\ !pre\_x, m_1; m_2, (s_1 \cup s_2 \cup [\mathtt{true}/init] \cup [?/pre\_x] \rightarrow [] \rangle}$$

$$\text{EXTEND}\quad \frac{\sigma \models e_1 : \langle c_1, m_1, s_1 \rightarrow [] \rangle \quad \sigma \models e_2 : \langle c_2, m_2, s_2 \rightarrow [] \rangle}{\sigma \models \mathtt{extend}\, e_1\ e_2 : \langle (c_1\ c_2), m_1; m_2, (s_1 \cup s_2) \rightarrow [] \rangle}$$

$$\text{ABST}\quad \frac{\sigma, x : \langle xc, (), [] \rightarrow [] \rangle \models e : \langle c, m, [a_1/x_1; ...; a_n/x_n] \rightarrow s \rangle}{\sigma \models \lambda x.e : \langle \lambda xs, xc.\ \begin{array}{l}\mathtt{match}\ !xs\ \mathtt{with} \\ \mathtt{Env}(x_1, ..., x_n) \rightarrow c\end{array}, \lambda xs, xc.\ \begin{array}{l}\mathtt{match}\ !xs\ \mathtt{with} \\ \mathtt{Env}(x_1, ..., x_n) \rightarrow m\end{array}, [] \rightarrow [a_1/x_1; ...; a_n/x_n] \rightarrow s \rangle}$$

$$\text{ABST-[]}\quad \frac{\sigma, x : \langle xc, (), [] \rightarrow [] \rangle \models e : \langle c, m, [] \rightarrow s \rangle}{\sigma \models \lambda x.e : \langle \lambda xc.c, \lambda xc.m, [] \rightarrow [] \rightarrow s \rangle}$$

$$\text{APP}\quad \frac{\sigma \models e_1 : \langle c_1, m_1, s_1 \rightarrow [a_1/x_1; ...; a_n/x_n] \rightarrow s_3) \rangle \quad \sigma \models e_2 : \langle c_2, m_2, s_4 \rightarrow [] \rangle}{\sigma \models e_1\ e_2 : \langle c_1\ x\ c_2 \rangle, \begin{array}{l}\mathtt{try}\ m_1\ x\ c_2\ m_2 \\ \mathtt{with\ fail} \rightarrow m_2\end{array}, (s_1 \cup s_4 \cup [[a_1/x_1; ...; a_n/x_n]/x]) \rightarrow s_3}$$

$$\text{APP-rec}\quad \frac{\sigma \models e_1 : \langle c_1, m_1, s_1 \rightarrow rec\, xs.s_2 \rightarrow s_3) \rangle \quad \sigma \models e_2 : \langle c_2, m_2, s_4 \rightarrow [] \rangle}{\sigma \models e_1\ e_2 : \langle \begin{array}{l}\mathtt{match}\ !x\ \mathtt{with} \\ \mathtt{NoEnv} \rightarrow x := new(s_2); c_1\ x\ c_2 \\ |\_ \rightarrow c_1\ x\ c_2\end{array}, \begin{array}{l}\mathtt{match}\ !x\ \mathtt{with} \\ \mathtt{NoEnv} \rightarrow x := new(s_2); m_2 \\ |\_ \rightarrow\ \mathtt{try}\ m_1\ x\ c_2; m_2 \\ \qquad \mathtt{with\ fail} \rightarrow m_2\end{array}, (s_1 \cup s_4 \cup [rec\, xs.s_2/x]) \rightarrow s_3 \rangle}$$

$$\text{APP-[]}\quad \frac{\sigma \models e_1 : \langle c_1, m_1, s_1 \rightarrow [] \rightarrow s_3 \rangle \quad \sigma \models e_2 : \langle c_2, m_2, s_4 \rightarrow [] \rangle}{\sigma \models e_1\ e_2 : \langle (c_1\ c_2), \begin{array}{l}\mathtt{try}\ m_1\ c_2; m_2 \\ \mathtt{with\ fail} \rightarrow m_2\end{array}, (s_1 \cup s_4) \rightarrow s_3 \rangle}$$

$$\text{REC}\quad \frac{\sigma, x : \langle xc, xm, rec\, xs.s \rangle \models e : \langle c, m, rec\, xs.s \rangle}{\sigma \models rec\, x.e : \langle rec\, xc.c, \mathtt{let}\ xc = rec\, xc.c\ \mathtt{in}\ rec\, xm.m, rec\, xs.s \rangle}$$

$$\begin{array}{lcl}
new([]) & = & \mathtt{ref\ NoEnv} \\
new(rec x.s) & = & new(s) \\
new([a_1/x_1; ...; a_n/x_n]) & = & \mathtt{Env}(new(a_1), ..., new(a_n)) \\
new(v) & = & v \\
new(x) & = & \mathtt{ref\ NoEnv}
\end{array}
\qquad
\begin{array}{lcl}
def_type([]) & = & \mathtt{NoEnv} \\
def_type(rec x.s) & = & \mathtt{type}\ x = def_type(s) \mid \mathtt{NoEnv} \\
def_type([a_1/x_1; ...; a_n/x_n]) & = & \mathtt{Env\ of}\ def_type(a_1)\ \mathtt{ref}\ * \\
 & & ... * def_type(a_n)\ \mathtt{ref} \\
def_type(v) & = & type(v) \\
def_type(x) & = & x
\end{array}$$

Figure 4. The compilations rules

produces a code which returns the content of this variable (pre_x). In the *modif-state* part, the new value of pre_x is computed if c produces a value (does not raise any exception). Else, no modification has to be done. The *memory* part associated with `pre`is thus the *memory* part of its argument plus the new link $[v/pre\_x]$. Note in this rule that the introduced variable pre_x must be a new variable (not belonging to the domain of s).

- `fby` is very similar to `pre` except that the initial value of the memory is not known. `fby` has two states: an initial state where it returns the value of its first argument and a general state where it returns the value of its second argument. Two memories are necessary: the *init* memory is first initialized to `true` and becomes `false` forever and the pre_x variable which acts as a "latch". `pre_x` is initialized with any dummy value of the correct type. It is important to notice that only `pre` and `fby` primitives need some memory.

- `extend` produces a simple application with no new memory.

- When dealing with abstraction $\lambda x.e$ we can distinguish two different cases (rules (ABST) and (ABST-[])):

  - e may need some memory. An abstraction may be called from different places in the code and the modifications on the internal state must be different for each call. The solution we propose is to produce an abstraction with a new argument xs representing the internal state. This memory can then be modified by the function. Moreover, the function has to match the structure of the memory it may receive. The memory is written $[] \rightarrow s$ saying that no memory is used to produce the abstraction whose body uses s. We can now explain the structure $s_1 \rightarrow s_2$. It means that the initial memory s_1 is used for creating a value and that value uses s_2. Thus, constant values (like `const`1) have the memory $[] \rightarrow []$.

  - e may not need any memory. In that case, the abstraction $\lambda x.e$ (where x is a stream) corresponds exactly to a scalar abstraction with the same pattern: the returned value of the abstraction does not depend on the previous calls to the function. For example, consider the expression $\lambda x.(\texttt{extend}(\texttt{extend} + (\texttt{const}\,1)x)$. The scalar code associated to this function over lists is simply $\lambda xc.(+1\,xc)$. Thus, the *code* part associated to an abstraction without internal state is also an abstraction with only one parameter.

- Application rules match abstraction rule: There are two cases:

  - e_1 may modify an internal state. In that case, this memory has to be given. Thus we add a link between a new name x and the initial memory, the body is waiting for. We then produce the cases depending on the possible values of x. If the memory is recursive, we add a special case for creating a new memory dynamically in case the entry of the function is empty (rule (APP-rec)). This time, the memory size is not bounded anymore: a new entry is created each time the function is called with an empty entry. We shall discuss this point further, explaining the possible optimization in order to keep "reactivity" constraints.

  - e_1 may not modify an internal state. In that case, the application over lists is similar to the application over scalars.

- Recursion produces three recursive expressions. A recursion over streams becomes a recursion over scalars. Nonetheless, as we shall see on examples, such a recursion $rec\,xc.c$ is not always a true recursion in the sense that xc may not be free in c. In that case, it simplifies to c. This is exactly what happens for LUSTRE programs: they are compiled into non recursive ones. The *modif-state* may behave similarly. The memory part may also be recursive: in fact, this corresponds to the definition of a recursive data type.

- Finally, the *def_type* and *new* constructions translate memories into Caml-light data structures. The *def_type* function has to produce flat definitions from (possibly nested recursive) memory expressions. We only give here the case of non nested memories. Moreover, we restrict the definition to first-order (recursive or not) memories without abstraction or application. The general case is a matter of further improvements.

A lot of optimizations can (and should) be applied on the produced code. For example, several matches can be factorized and empty computation like $x; m$ where x is a variable can be simplified. This is still to be done.

4 Implementation

This section deals with implementation aspects. The final goal is to include a special (efficient) stream structure inside some ML language, based on the application of the clock calculus and the compilation method.

A first prototype tool is under construction. Clock calculus and compilation are applied to a simple functional language as defined in the previous section. Latter, we plan to include it as a front-end of the Caml-light system.

4.1 Clock calculus

The implementation of the clock calculus has been done in the spirit of classical implementations of the Caml-light type system [17] with a slight modification for the recursion rule. The clock calculus is obtained using a destructive unification. Here, the unification is slightly 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 (modulo α-conversion). Recursive expressions should be clocked in an iterative way, as noted in [19]. We have decided to implement a sub-system by restricting the iteration to two. It seems to be sufficient for many examples.

Example 4 Expressions can be defined at top-level. The clock expression is returned for every entry.

```
let plus=fun x->fun y->(extend (extend (const +) x) y)

clock: 'a-<>->'a-<>->'a

let filter =
 fun x->let half=rec half.(const true) fby (notl(half))
        in x when half

clock:'a-<>->'a on (rec half.(const true) fby (notl half))
```

```
let natx=fun x ->
  rec natx.pre 0 (plus natx ((const 1) when x))

clock: 'a-<x>->'a on x

fun x -> fun y -> merge (natx x)  y (const 1)

clock: 'a-<x>->('a on x) on (natx x)-<>->'a on x
```

□

4.2 Compilation

The implementation produces a Caml-light program with side effects. It preserves, as much as possible, the functionality and structure of the initial program. We could have chosen to produce directly C code. Yet, our choice is quite natural: the point in our work is to show how to compile efficiently the data part of the language without dealing with the control part. So, by translating our programs into ML, we preserve every other possible optimizations (function calls, closures, ...) related to control. Moreover some efficient C code can then be obtained using ML to C translators.

Example 5 (A recursive value) For instance, the produced Caml-light code for the recursive expression *nat* (example 1) is (after applying some simplifying rules):

```
let trans =
  let prex = ref 2
  in fun () -> let nat_code = !prex in
               let nat_mem = prex:=!prex+1 in
               print_int(nat_code)
;;
```

trans is the transition function. The memory is simply $[0/prex]$ due to the preconstruct. The `nat_code` part returns the current value which is initialized to 2 and `nat_mem` computes the next value and puts it into the accumulator. In this example, all useless instructions have been deleted. □

Example 6 (The Eratosthenes Sieve) Let us see now the case of the Eratosthenes sieve (section 2.3). Its clock is $\forall\alpha.\alpha \rightarrow \alpha$. The compilation process produces the following code (with some hand-made simplifications).

```
type env = NoEnv
         | Env of (bool ref) * (int ref) *
                  (bool ref) * (env ref)
;;
let new() = Env(ref true,ref (-10),ref true,ref NoEnv)
(* -10 is a dummy integer value *)
;;
let first_code (init,prev) x = if !init then x else !prev
;;
let first_mem (init,prev) x =
            (prev:=(first_code (init,prev) x);
             init:=false)
;;
let rec sieve_code e x =
 match !e with
   Env(init,prev,prex,e) ->
     let filter_code =
           (x mod (first_code(init,prev) x))!=0
     in if filter_code
        then match !e with
               NoEnv -> e:=new();sieve_code e x
             | _ -> sieve_code e x
        else !prex
;;
let rec sieve_mem e x =
  match !e with
    Env(init,prev,prex,e) ->
       begin
        let filter_code =
             (x mod (first_code(init,prev) x))!=0
        in first_mem(init,prev) x;
           if filter_code
           then match !e with
                  NoEnv -> e:=new()
                | _ -> sieve_mem e x
           else prex:=false
       end
;;
let e = ref (new());;
let trans =
    let prex = ref 2
    in fun () -> let nat = !prex in
                 if sieve_code e nat
                 then print_int(nat)
                 else ();
                 sieve_mem e nat;
                 prex:=nat+1
;;
```

`code()` first returns 2, then 3, then nothing (`()`) since the fourth integer is not a prime integer, etc. This corresponds exactly to the compilation of `nat when sieve(nat)` that produces an output only when the condition is true. During the execution of this program, the memory `e` grows. A son is created each time the entry is a prime number so the internal memory has the form of a stack that contains the previous prime numbers. □

Example 7 (A tail-recursive function) Let us consider a recursive function computing the list of positive integers. It can be written as:

$$\texttt{rec}\ f.(\lambda x.x\,\texttt{fby}\,(f(plus\,x\,(\texttt{const}\,1))))$$

where *plus* stands for the addition over streams. $f(\texttt{const}0)$ produces the list of positive integers. The clock of f is $\forall\alpha.\alpha \rightarrow \alpha$. The compilation method will produce the following Caml-light recursive code.

```
type env = NoEnv
         | Env of (bool ref) * (int ref) * (env ref)
;;
let new() = Env(ref true,ref (-10),ref NoEnv)
(* -10 is a dummy integer value *)
;;
let rec f_code e x =
  match !e with
    Env(init,prex,e) -> if !init then x else !prex
;;
let rec f_mem e x =
  match !e with
    Env(init,prex,e) -> match !e with
                          NoEnv ->e:=new();init:=false;
                                  prex:=f_code e (x+1)
                        | _ ->f_mem e (x+1);init:=false;
                               prex:=f_code e (x+1)
;;
let code =
  let e = ref (new())
  in fun () -> let v = f_code e 1
               in f_mem e 1;print_int(v)
;;
```

The memory is recursive since the function is recursive. It contains the successive init values of the nested machines. Of course, this function is not truly recursive and can be implemented in a finite way (all the recorded values in the structure are always the same). We call it a *tail-recursive* function in the usual sense since the body of the function dies when it creates a new son. This kind of recursion is not treated efficiently by the current compilation method that focused on truly recursive functions. To be implemented efficiently, these recursions should be rewritten in a non recursive and non functional way. For example, the function f should be rewritten as:

$$\lambda x.x\, \texttt{fby}\, (\texttt{rec}\, \mathit{ff}.x\, \texttt{fby}\, \mathit{plus}\, \mathit{ff}\, (\texttt{const}\, 1))$$

which could then be compiled into a bounded transition function. The recognition of tail recursive functions is a matter of further work. □

4.3 Preliminary results

For comparison purposes, we report here executions times (in seconds on a Sparc 10 workstation) of two examples, a reactive and a truly recursive one, using three different approaches:

- the source program is compiled with the LML compiler.
- the compiled program is compiled using caml-light 0.7 compiler.
- the compiled program is directly compiled into C

Examples	`lmlc`	`camlc`	C code (`cc -O4`)
nat (100000)	2s	1.4s	.05s
nat (1000000)	24.1s	14s	.26s
sieve (10000)	15.8s	66s	2.4s
sieve (100000)	1116s	3931s	146s

Whereas the implementation is still limited, this very preliminary comparison shows that, when implemented in C, our method is more efficient for real-time programs and some truly recursive functions than usual lazy semantics methods, represented here by the LML compiler. Note also that our approach compares favorably with respect to code length: LML code is about 500K bytes.

5 Related works

This work is clearly related to the topics of "listlessness"[25] and "deforestation"[26]. There have been extended works on the subject, and it is quite difficult to summarize all of them. Yet, roughly speaking, these can be classified into two groups:

- Particular methods, based on some set of special purpose primitives, among which we can cite deforestation shortcuts [3], communication lifting [24], and the one presented here. It is quite difficult to compare them, because each has its proper set of primitives and constructs as well as its own goal.

  The communication lifting method [24] is very similar to that can be achieved with LUSTRE, and thus doesn't intend to deal with dynamical networks. Furthermore, it hardly deals with non length preserving functions like `filter`. On the contrary, our `when` primitive is a filter-like, non length preserving, function, which is central to our approach[3]

  Deforestation shortcuts [3], based on `foldr-build` transformation, is fairly general; yet it seems to us that it fails in addressing the problem of functions having several stream arguments, which are also central to our approach.

- General purpose methods like the deforestation algorithm [26], which applies to any recursive data type. With respect to these methods, ours is less general, but better fits its particular domain of application. For instance, it is well-adapted to non linear (in the sense of [26]) expressions, for which the deforestation algorithm is not guaranteed to terminate.

  For instance, it can be checked that deforestation doesn't terminate with:

```
y = merge c
          (x when c)
          ((pre 0 y) when (notl c))
```

  while we easily deforest it.

  Moreover, we can treat recursive functions (as sieve) without considering them as macros.

Finally, Boussinot[8] has defined an execution scheme for process networks, consisting in synchronously executing one step of each process at a time, and this yields an operational semantic very similar to ours. Yet, he doesn't define corresponding clock calculus and compilation schemes. Thus his approach doesn't allow communications to be efficiently compiled.

6 Conclusion

Finally, since the early days of Kahn and Lucid models, several synchronous data-flow languages have been proposed, for programming reactive systems [20, 2, 16], for programming parallel machines [14, 13], for describing and synthesizing hardware ([18] and the data-flow part of VHDL [1]). Yet, none of these proposals seem to have considered recursion. By characterizing synchrony in data-flow languages as the possibility of restricting to listlessly compilable programs, we intended to show that this is not contradictory with recursion. We have provided a synchronous operational semantics, clock constraints and compiling rules for handling recursion as well as higher order programming. Thus, both features can be added to those languages yielding a large increase in expressive power, possibly even reaching the expressive power of general purpose languages.

A possible achievement can consist of applying the techniques developed in this paper in order to add a lazy synchronous stream type on the top of a strict ML-like language.

In doing this, it will be easy to characterize a reactive subset of this language: reactive programs will be those whose only recursive types are streams, and which don't use recursive functions.

[3] In [24], it is claimed that Lustre can be simulated using communication lifting; this is true, but it is only a simulation: instead of being rejected at compile time thanks to some clock calculus, non listless programs yield execution errors.

Yet, this reactive subset can be enriched by considering tail recursive functions as in example 7. As already noted in [10] tail recursion is expected to help in clarifying and simplifying programs, by providing control structures. Thus efficient tail recursion handling will be a matter of future work.

References

[1] IEEE standard VHDL reference manual. Technical report, 1988.

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

[3] J. Launchbury A. Gill and S.L. Peyton Jones. A short cut to deforestation. In *6th Functional programming languages and computer architecture*, pages 223–232, Copenhagen, June 1993. ACM.

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

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

[6] L. Augustsson and T. Johnsson. Lazyml user's manual version 0.999.4. Technical report, Chalmers University of Technology, Goteborg, Sweden, 1993.

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

[8] F. Boussinot. Réseaux de processus réactifs. Technical Report 1588, INRIA Sophia-Antipolis, janvier 1992.

[9] P. Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94:125–140, 1992.

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

[11] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Conference on Principles of Programming Languages*, 1982.

[12] D.P. Friedman and D.S. Wise. CONS should not evaluate its arguments. In *Proceding of the ICALP'76 Conference*, pages 257–284, 1976.

[13] J.L. Giavitto. A synchronous data-flow language for massively parallel computers. In *Parallel Computing'91*, London, 1991.

[14] D. C. Cann J. T. Feo and R. R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computation*, 10:349–366, 1990.

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

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

[17] Xavier Leroy. The caml light system : release 0.7 : documentation and user's manual. Technical report, INRIA, INRIA, Domaine de Voluceau-Rocquencourt, 78153 Le Chesnay Cedex, France, July 1995.

[18] C. Maurras. *Alpha, un langage équationnel pour la conception d'architectures parallèles synchrone*. PhD thesis, Université de Rennes 1, France, 1989.

[19] Alan Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *Proceedings of the Sixth International Symposium on Programming, Toulouse*, pages 217–228. Springer-Verlag LNCS 167, April 1984.

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

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

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

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

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

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

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