A FUNCTIONAL EXTENSION TO LUSTRE

PAUL CASPI

and

MARC POUZET

Laboratoire VERIMAG

Montbonnot, 38330, France

E-mail: {Paul.Caspi,Marc.Pouzet}@imag.fr

ABSTRACT

Lustre is a data-flow programming language for reactive purposes, which has been given a synchronous operational semantic. Static checks called "clock calculus" restrict Lustre programs to those data-flow networks which can be executed synchronously. As a by-product of this restriction, Lustre programs enjoy efficient compiling techniques. Yet, due to its reactive origin, Lustre applies only to static networks. We show here, that both this operational semantic, and clock calculus can be extended toward accounting for general functional features: abstraction, application and recursion. This allows us to give sense to both higher order, and dynamical synchronous data-flow networks, which can thus be expected to share the same efficient compiling techniques.

1. Introduction

1.1. Some milestones in data-flow programming

In the seventies, the Lucid language was proposed[2,1] as a way of overcoming the lack of efficency of functional languages by providing them with stream based iterations. At the same time, Kahn [12] showed that the semantics of networks of asynchronous deterministic processes could be described as a system of recursive equations over streams, very similar to Lucid programs. Then, the concept of lazy evaluation emerged[9,16], thus accounting for finite and infinite data structures like streams, and now, modern lazy functional languages are available, like LazyML and Haskell[3,11] which easily allow to write data-flow programs, by expressing streams as abstract data types, and providing nice features, such as currying and higher order programming. Yet, the problems of inefficiency remained, and Wadler[18] proposed new techniques, which he called "listlessness"[18] so as to try to overcome it. 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. Latter, he generalized it to other recursive data types, and called it "deforestation", [19].

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 had always used systems of recursive equations over streams of values (signals) as a natural formalism for reasoning about their systems, and they often found boring and prone to errors, having to translate these equations into sequential programs. Some of them found that it could be possible to handle this translation automatically, that is to say, to compile them. For instance, some of the programs running on the Airbus A320 aircraft have been obtained in that way. This led computer scientists to propose languages (Lustre, Signal), toolboxes (Ptolemy) and compilers for this field of application [4,10,13] which they called "Synchronous Data-flow".

Those compilers[8] 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 concept is twofold:

- It encompasses some static analysis step, aiming at rejecting those data-flow networks that cannot be listlessly evaluable. This step is sometimes referred to as either a "clock calculus"[10,4] or "consistency checking"[13].
- It encompasses a compiling step which aims at producing listless sequential code for those 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 permit 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 those reactions use bounded memory and bounded reaction time.

The purpose of the paper is to show that the clock constraints rules are not bound to those reactive restrictions and can apply to more general stream languages yielding abstraction, application and general recursion. These constraints can be seen as sufficient conditions allowing a program to be compiled without using more than one item of a given type for each lazy stream of that type.

This is obtained by

- considering a curryfied version of the language, which will help in handling functional formalisms,
- extending the operational semantic *a la* Plotkin of a synchronous reactive data-flow language, namely LUSTRE[8] so as to apply to abstraction, application, and recursion. This semantic allows us to characterize "synchronous data-flow behaviors",
- extending the clock constraints of LUSTRE to these functional features: this is obtained by expressing the clock calculus of Lustre[10] as a type system, which, in turn, allows us to generalize it to functional features,
- showing that programs that can be typed in the preceding sense, can be synchronously evaluated. This will be done stepwisely, first on the reactive fragment of the language, and then on the generalized language.

2. A LUSTRE-like reactive data-flow language

We present here a slightly modified version of LUSTRE. The variations are due to the facts that we need a curryfied version, and for the sake of conciseness, we are not willing to present the whole syntax of the language. Finally, some of the LUSTRE operators have been modified so as to yield an easier generalization; particularly, in contrast with LUCID's tradition, we shall built streams from values: this will be useful when explaining the synchronous operational semantic, which will, ultimately, produce ordinary values..

2.1. Primitive constructs

- 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 :

$$[True; True; \ldots]$$

- `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 :

$$[False; False; \ldots]$$

- `fby` is the delay operator :

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

It allows recursive expressions (feedback networks or circuits) to be safely built, without exhibiting deadlocks; for instance :

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

has the infinite periodic behavior :

$$[True; False; True; False; \ldots]$$

- 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, if `x` is the stream $[x_0; x_1; \ldots x_n; \ldots]$, `x when half` is the stream $[x_0; x_2; \ldots x_{2n}; \ldots]$

- 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, if `x` and `y` are the streams $[x_0; x_1; \ldots x_n; \ldots]$ and $[y_0; y_1; \ldots y_n; \ldots]$, `merge half x y` is the stream

$$[x_0; y_0; x_1; y_1; \ldots x_n; y_n; \ldots]$$

These are essentially the primitive functions of the programming language LUSTRE[10], which has been successfully used in several large scale control software and hardware projects.

2.2. A kernel language

The following grammar corresponds to a kernel of the LUSTRE language. Expressions are ranged over e and are built from stream variables x, the primitive constructs presented above and recursive definitions $(rec\ \ x.e)$. Scalar constants (sc) are imported via the `const` primitive and range over scalar types: — (i) denotes integers, (`t`) and (`f`) denote classical true and false constants — as well as scalar functions and operations: $(+)$, the integer addition, (`not`) and (`and`) classical boolean operators. Others classical scalar constants can be added to the language.

$$
\begin{array}{rcl}
e & ::= & \texttt{const}\ sc \mid \texttt{merge}\ \ e\ e\ e \mid e\ \texttt{fby}\ \ e \\
 & & \mid \texttt{extend}\ \ e\ e \mid \texttt{pre}\ \ sc\ e \mid e\ \texttt{when}\ \ e \mid \texttt{notl}\ \ e \\
 & & \mid x \mid rec\, x.e \\
sc & ::= & + \mid \texttt{and}\ \ \mid \texttt{not}\ \ \mid i \mid \texttt{f}\ \ \mid \texttt{t}\ \ \mid \ldots
\end{array}
$$

This recursive language, indeed allows only static networks to be defined: the reason is that recursion ranges only over 0 order expressions since there is no mean of abstracting them.

2.3. Its synchronous operational semantic

Such a recursive language can be executed in a call-by-need way, using the definition given previously for the primitives. The program is then executed as a program managing lists, leading here, to a poor implementation. We shall see that such a language can be executed without managing lists but only scalars. For this, we define a *synchronous operational semantics* where no list structures is needed during the computation. This semantic is twofold:

First behaviors are derived from instantaneous behaviors by the rule:

$$
\frac{\sigma \vdash e \xrightarrow{v} e'\ ,\ \Sigma \vdash e' \stackrel{w}{\Longrightarrow} e''}{\sigma @ \Sigma \vdash e \stackrel{v@w}{\Longrightarrow} e''}
$$

which can be stated as:

If, on the assumption σ on the instant behavior of its free variables, expression e yields the value v and rewrites as e', and if, on the assumption Σ of the behaviors of the same variables, e' yields w and rewrites as e'', then on the assumptions $\sigma@\Sigma$, e yields $v@w$ and rewrites as e''.

Here, "@"is the list concatenation, and it is extended to assumptions by:

$$
\sigma @ \Sigma = \{x \stackrel{v@w}{\Longrightarrow} x'' \mid \exists x'.\ x \xrightarrow{v} x' \in \sigma \ and\ x' \stackrel{w}{\Longrightarrow} x'' \in \Sigma\}
$$

The operational semantics is given at table 1 for each construction.

- The first axiom says simply that an assumption can be used.
- The `const` primitive produces a list of constants thus one step of the `const` v program produces v and the continuation `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 a list of scalar functions to a list of arguments. The semantics states that the two arguments must be present in parallel. If none of them is present, the code stays the same and produces no value. Note here that there is no way to execute the expression when only one argument is present.

Table 1. The operational semantics of the reactive data-flow language

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

$$\frac{}{\sigma \vdash \texttt{const}\ \ v \xrightarrow{[v]} \texttt{const}\ \ v} \qquad \frac{}{\sigma \vdash \texttt{const}\ \ v \xrightarrow{[]} \texttt{const}\ \ v}$$

$$\frac{\sigma \vdash f \xrightarrow{[]} f_1 \quad \sigma \vdash x \xrightarrow{[]} x_1}{\sigma \vdash \texttt{extend}\ \ f\ x \xrightarrow{[]} \texttt{extend}\ \ f_1\ x_1} \qquad \frac{\sigma \vdash f \xrightarrow{[f_0]} f_1 \quad \sigma \vdash x \xrightarrow{[x_0]} x_1}{\sigma \vdash \texttt{extend}\ \ f\ x \xrightarrow{[f_0 x_0]} \texttt{extend}\ \ f_1\ x_1}$$

$$\frac{\sigma \vdash c \xrightarrow{[]} c_1 \quad \sigma \vdash x \xrightarrow{[]} x_1 \quad \sigma \vdash y \xrightarrow{[]} y_1}{\sigma \vdash \texttt{merge}\ \ c\ x\ y \xrightarrow{[]} \texttt{merge}\ \ c_1\ x_1\ y_1} \qquad \frac{\sigma \vdash c \xrightarrow{[t]} c_1 \quad \sigma \vdash x \xrightarrow{[x_0]} x_1 \quad \sigma \vdash y \xrightarrow{[]} y_1}{\sigma \vdash \texttt{merge}\ \ c\ x\ y \xrightarrow{[x_0]} \texttt{merge}\ \ c_1\ x_1\ y_1}$$

$$\frac{\sigma \vdash c \xrightarrow{[f]} c_1 \quad \sigma \vdash x \xrightarrow{[]} x_1 \quad \sigma \vdash y \xrightarrow{[y_0]} y_1}{\sigma \vdash \texttt{merge}\ \ c\ x\ y \xrightarrow{[y_0]} \texttt{merge}\ \ c_1\ x_1\ y_1}$$

$$\frac{\sigma \vdash c \xrightarrow{[]} c_1 \quad \sigma \vdash x \xrightarrow{[]} x_1}{\sigma \vdash x\ \texttt{when}\ \ c \xrightarrow{[]} x_1\ \texttt{when}\ \ c_1} \qquad \frac{\sigma \vdash c \xrightarrow{[t]} c_1 \quad \sigma \vdash x \xrightarrow{[x_0]} x_1}{\sigma \vdash x\ \texttt{when}\ \ c \xrightarrow{[x_0]} x_1\ \texttt{when}\ \ c_1}$$

$$\frac{\sigma \vdash c \xrightarrow{[f]} c_1 \quad \sigma \vdash x \xrightarrow{[x_0]} x_1}{\sigma \vdash x\ \texttt{when}\ \ c \xrightarrow{[]} x_1\ \texttt{when}\ \ c_1}$$

$$\frac{\sigma \vdash x \xrightarrow{[]} x_1}{\sigma \vdash \texttt{pre}\ \ v\ x \xrightarrow{[]} \texttt{pre}\ \ v\ x_1} \qquad \frac{\sigma \vdash x \xrightarrow{[x_0]} x_1}{\sigma \vdash \texttt{pre}\ \ v\ x \xrightarrow{[v]} \texttt{pre}\ \ x_0\ x_1}$$

$$\frac{\sigma \vdash x \xrightarrow{[]} x_1 \quad \sigma \vdash y \xrightarrow{[]} y_1}{\sigma \vdash x\ \texttt{fby}\ \ y \xrightarrow{[]} x_1\ \texttt{fby}\ \ y_1} \qquad \frac{\sigma \vdash x \xrightarrow{[x_0]} x_1 \quad \sigma \vdash y \xrightarrow{[]} y_1}{\sigma \vdash x\ \texttt{fby}\ \ y \xrightarrow{[]} \texttt{pre}\ \ x_0\ y_1}$$

$$\frac{\sigma \vdash x \xrightarrow{[x_0]} x_1 \quad \sigma \vdash y \xrightarrow{[y_0]} y_1}{\sigma \vdash x\ \texttt{fby}\ \ y \xrightarrow{[x_0]} \texttt{pre}\ \ y_0\ y_1}$$

$$\frac{\sigma, x \xrightarrow{x_0} x_1 \vdash e \xrightarrow{e_0} e_1}{\sigma \vdash rec\ \ x.e \xrightarrow{rec\ x_0.e_0} rec\ \ x_1.e_1[rec\ \ x_0.e_0/x_0]}$$

- The semantics of the `merge` is straightforward : if the condition produces a true value, then the `merge` produces the value produces by the true branch, else, the value produces by the false branch.

- A `when` expression produces the value of its second argument only when the value of its first is true. Else, the value is forgotten.

- The `fby` operation may produces a value only when its first argument produces a value. In this case, it is transformed in a `pre` operation.

- The `pre` operation is close to the classical `cons` constructor in functional languages over lists.

- The *rec* rule is very natural : yet, it should be noted that in LUSTRE we are allowed to get 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. Yet, static deadlock detection is fairly easy in LUSTRE since it suffices to show that each variable bound by a *rec* is within the scope of a `pre` or `fby` .

The substitution $e_1[e_2/y]$ used here is quite unusual since y may be a pattern of the form $[x]$ where x is a variable. In this case the substitution does some pattern matching and we get $e_1[[e]/[x]] = e_1[e/x]$.

Example 1 Let us consider now an example of synchronous evaluation. We shall see that the program

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

computes the stream of positive integers. Here, the $x{+}1$ expression stands for `extend` (`extend` (`const` $+)x)y$ using the classical addition between scalars. 1 stands for a stream of 1 (thus, `const` 1).

We have :

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

which reduces to :

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

It is quite easy to see that the program produces the stream of positive integers.
□

Within the framework, most functions can be executed synchronously. Yet, there are functions that cannot.

2.4. A non synchronous example

Let us consider the following expressions :

```
x andl (x when half)
where
  half = false fby (notl half)
  false = const False
```

The successive values of the variables can be represented like the following:

$$
\begin{array}{lcllllllll}
\texttt{x} & = & x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & \ldots \\
\texttt{false} & = & f & f & f & f & f & f & \ldots \\
\texttt{half} & = & f & t & f & t & f & t & \ldots \\
\texttt{x when half} & = & & x_1 & & x_3 & & x_5 & \ldots \\
\texttt{x andl (x when half)} & = & & x_0 \texttt{and } x_1 & & x_1 \texttt{and } x_3 & & x_2 \texttt{and } x_5 & \ldots
\end{array}
$$

In terms of control theory and hardware, **`x when half`** is a half-frequency sampler. This expression is not synchronous: informally, the `andl` primitive waits for its two arguments and consumes them at the same rate but in the final expression, the first argument has twice the number of items of its second argument. The diagram shows that the currently set of used values from the list `x` grows. To compute the ith value of the final list, about i successive values of `x` are necessary. Thus, it is not possible to replace the list `x` by a scalar value containing only a finite window of useful elements from it.

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 $\sigma \vdash x \texttt{ andl } (x \texttt{ when } half) \xrightarrow{[v_0]} e$.

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

$$
\frac{\sigma \vdash x \xrightarrow{[x_0]} x' \qquad \dfrac{\sigma \vdash x \xrightarrow{[x_0]} x' \quad \sigma \vdash half \xrightarrow{[v]} half'}{\sigma \vdash (x \texttt{ when } half) \xrightarrow{[y_0]} x' \texttt{ when } half'}}{\sigma \vdash x \texttt{ andl } (x \texttt{ when } half) \xrightarrow{[x_0 \texttt{ and } y_0]} x' \texttt{ andl } x' \texttt{ when } half'}
$$

It is possible only if $\sigma \vdash half \xrightarrow{[t]} half'$ which does not hold. Thus, there is no possible execution of the program. $\square$

2.5. A clock calculus for the reactive language

The idea of the LUSTRE clock calculus is to provide statically checkable conditions allowing an expression to be synchronously evaluable. The version presented here is a slight generalization of the ones presented in [8,5], in the sense that it uses unification, instead of fixed points. This will make it easier to prove the soundness theorem 1, as well as to generalize it to functional features as we shall see it in the next section.

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:

$$H ::= [e_0 : cl_0, ..., e_n : cl_n]$$

A clock cl is either a clock variable α, or a sub-clock of a clock, $cl \texttt{ on } e$ monitored by some boolean stream expression e:

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

The axioms and inference rules are based on the syntax. They are given in the table 2. The meaning of each rule is the following:

- An assumption can be used.
- A constant expression match any clock. Thus, it has a polymorphic clock α.
- The clock of the two arguments of an `extend` must be the same.

Table 2. A clock calculus for the reactive synchronous language

$$\frac{}{H \vdash \texttt{const}\ \ v : \alpha} \qquad \frac{}{H, x : cl \vdash x : cl}$$

$$\frac{H \vdash e_1 : cl \quad H \vdash e_2 : cl}{H \vdash \texttt{extend}\ \ e_1\ e_2 : cl} \quad \frac{H \vdash e_1 : cl \quad H \vdash e_2 : cl}{H \vdash e_1\ \texttt{when}\ \ e_2 : cl\ \texttt{on}\ \ e_2} \quad \frac{H \vdash e : cl}{H \vdash \texttt{pre}\ \ v\ e : cl}$$

$$\frac{H \vdash e_1 : cl \quad H \vdash e_2 : cl\ \texttt{on}\ \ e_1 \quad H \vdash e_3 : cl\ \texttt{on}\ \ (\texttt{notl}\ \ e_1)}{H \vdash \texttt{merge}\ \ e_1\ e_2\ e_3 : cl}$$

$$\frac{H \vdash e_1 : cl \quad H \vdash e_2 : cl\ \texttt{on}\ \ (\texttt{notl}\ \ e_1) \quad H \vdash e_3 : cl\ \texttt{on}\ \ e_1}{H \vdash \texttt{merge}\ \ e_1\ e_2\ e_3 : cl}$$

$$\frac{H \vdash e_1 : cl \quad H \vdash e_2 : cl}{H \vdash e_1\ \texttt{fby}\ \ e_2 : cl} \quad \frac{H \vdash e_1 : cl \quad H \vdash e_2 : cl\ \texttt{on}\ \ e}{H \vdash e_1\ \texttt{fby}\ \ e_2 : cl\ \texttt{on}\ \ e}$$

$$\frac{H, x : cl \vdash e : cl}{H \vdash rec\ \ x.e : cl}$$

- 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 one is faster than the one of the second one.
- The `pre` expression conserves the clock of its argument.
- The clock rule for the *rec* is identical to the very classical typing rule for *rec* expressions.

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) $e' : cl'$.

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`. Then the fact that the continuation is clockable guaranties that execution will proceed. □

3. A functional extension

We shall see now a functional extension of the language, keeping the same constructions and adding abstraction over stream arguments, $(\Lambda x.e)$, abstraction over scalar $(\lambda x.e)$ and application $(e\ e)$, where we don't distinguish between streams and scalars :

$$e \ ::= \lambda x.e \mid \Lambda x.e \mid e\ e \mid \ \ldots$$

3.1. Synchronous operational semantics

The operational semantics is still defined by the same relation. It is given at table 3.

Table 3. The synchronous operational semantics of the functional extension

$$\frac{\sigma, x \xrightarrow{x_0} x_1 \vdash f \xrightarrow{f_0} f_1}{\sigma \vdash \Lambda x.f \xrightarrow{\lambda x_0.f_0} \lambda x_0.\Lambda x_1.f_1} \qquad \frac{\sigma \vdash f \xrightarrow{f_0} f_1}{\sigma \vdash \lambda x.f \xrightarrow{\lambda x.f_0} \lambda x.f_1} \qquad \frac{\sigma \vdash f \xrightarrow{f_0} f_1 \quad \sigma \vdash e \xrightarrow{e_0} e_1}{\sigma \vdash fe \xrightarrow{f_0 e_0} (f_1 e_0) e_1}$$

The rule of abstraction 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. When an abstraction is over a scalar value then the execution may produce a new abstraction if the body produces something new. The rule of application matches the rule of abstraction.

3.2. A clock calculus for the functional extension

As usual in the Hindley–Milner framework[14], clock expressions are decomposed into clock schemes (σ) and clock instances (cl).

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

$cl \xrightarrow{x} cl'$ is the clock expression of a function with argument x. Clock schemes are constructed in the classical way of type schemes. It is an unconventional type expression (with respect to Hindley- Milner) since clocks contains expressions.

Table 4. The clock calculus of the functional extension

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

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

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

The clock calculus of the functional extension is given at table 4:

- Clock variables which are free in H can be generalized: $FV(H)$ stands for the free clock variables in H.

- Clocks schemes can be instantiated, yielding truly polymorphic clocks. Similarly, $FV(cl)$ stands for the free clock variables of the clock expression cl, and we must care for name clashes in instantiation.

- In abstracting over stream variables, we must keep trace of the variable being abstracted, as it can serve in a clock expression.

- The rule of application is consistent with the preceding one.

- Abstraction and application over scalar variables don't modify the clocks.

We can now state the main result of the paper :

Theorem 2 (soundness) *Theorem 1 holds for the extended language.*

The proof is done by showing that the family of morphisms defined in theorem 1 extends to the present case.$\square$

3.3. Applications

In previous papers[6,7], we have already discussed the interest of this extension :

In the first one, which presents a non curryfied version of this framework, several examples are provided, both from reactive and non reactive domains. The latter deals more specifically with reactive examples in a graphic programming style : the idea was to extend the classical data-flow "Block Diagram" approach of control theory, toward yielding recursive networks, while preserving bounded memory and bounded reaction time properties. This was achieved thanks to so-called "tail recursion". The reader is referred to those papers for more details.

3.4. Related works

Apart from those previously cited papers, this work is clearly related to the topics of "listlessness"[18] and "deforestation"[19]. Yet, the probably closest work is due to Vree and Hartel[17]. However, their approach cannot deal with non length-preserving functions, like `merge` , which constitutes the core of our approach.

4. Conclusion

Defining synchrony in stream-based programs as those programs that can be evaluated streamlessly, we have extended the synchronous operational semantic of a first-order stream language like LUSTRE toward higher order and recursive features. Then we have extended the clock calculus of LUSTRE, i.e. static checks ensuring that a program behaves synchronously, so as it applies to the extended language.

This work allows us to give a precise meaning to our proposals of recursive synchronous data-flow networks which have been previously published[6,7]. Its interest lies in the fact that it allows modularity and expressive power to be gained, without any loss in efficiency. This leads us to forecast some synchronous extension to an ML-like language[15], in the form of a synchronous infinite (thus lazy) stream type.

5. References

1. E. A. Ashcroft and W. W. Wadge. *Lucid, the data-flow programming language.* Academic Press, 1985.
2. E.A. Ashcroft and W.W. Wadge. Lucid, a formal system for writing and proving programs. *SIAM j. Comp.*, 3:336–354, 1976.

3. L. Augustsson and T. Johnsson. *LazyML user's manual Version 0.999.4*. Chalmers University of Technology, Göteborg, Sweden, 1993.
4. 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.
5. P. Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94:125–140, 1992.
6. P. Caspi. Lucid synchrone. In *Actes du colloque INRIA OPOPAC, Lacanau*. HERMES, novembre 1993.
7. P. Caspi. Towards recursive block diagrams. In *Proc. 19th IFAC/IFIP Workshop on real-time programming, Isle of Reichenau, Germany*. IFAC, June 1994.
8. P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. Lustre : a declarative language for programming synchronous systems. In *Proc. 14th ACM Symposium on Principles of Programming Languages*. ACM, 1987.
9. D.P. Friedman and D.S. Wise. CONS should not evaluate its arguments. In *ICALP'76*, pages 257–284, 1976.
10. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
11. 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.
12. G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74 Congress*. North Holland, Amsterdam, 1974.
13. 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.
14. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–375, 1978.
15. R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, 1989.
16. D.A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, 9:31–49, 1979.
17. W.G. Vree and P.H. Hartel. Communication lifting : fixed point computation for parallelism. *J. Functional Programming*, 1(1):1–33, 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.