Semantics and compilation of recursive sequential streams in 81/2

Jean-Louis Giavitto, Dominique de Vito, Olivier Michel

Semantics and Compilation of Recursive Sequential Streams in $8_{1/2}$

Jean-Louis Giavitto, Dominique De Vito, Olivier Michel

LRI u.r.a. 410 du CNRS,
Bâtiment 490, Université Paris-Sud, 91405 Orsay Cedex, France
Tel: +33 1 69 15 64 07 *e-mail:* giavitto@lri.fr

Abstract. Recursive definition of streams (infinite lists of values) have been proposed as a fundamental programming structure in various fields. A problem is to turn such expressive recursive definitions into an efficient imperative code for their evaluation. One of the main approach is to restrict the stream expressions to interpret them as a temporal sequence of values. Such *sequential* stream rely on a *clock analysis* to decide at what time a new stream value must be produced. In this paper we present a denotational semantics of recursively defined sequential streams. We show how an efficient implementation can be derived as guarded statements wrapped into a single imperative loop.
Keywords: stream, clock, compilation of dataflow graphs.

1 Introduction

To simplify the formal treatment of a program, Tesler and Enea [1] have considered single assignment languages. To accommodate loop constructs, they extend the concept of variable to an infinite sequence of values rather than a single value. This approach takes advantage of representing iterations in a "mathematically respectable way" [2] and to quote [3]: "series expressions are to loops as structured control constructs are to gotos". Such infinite sequences are called *streams* and are manipulated as a whole, using filters, transductors, etc.

This approach has led to the development of the stream data structure and the dataflow paradigm, according to a large variety of circumstances and needs. Since the declarative programming language Lucid [4], more and more declarative stream languages have been proposed: Lustre [5] and Signal [6] in the field of real-time programming, Crystal [7] for systolic algorithms, Palasm [8] for the programming of PLD, Daisy [9] for VLSI design, $8_{1/2}$ [10] for parallel simulation, Unity [11] for the design of parallel programs, etc. Moreover, declarative definitions of streams can be a by-product of the data-dependence analysis of more conventional languages like Fortran. In this case, a stream corresponds to the successive values taken by a variable, e.g. in a loop.

1.1 Synchronous Streams

Synchronous streams in Lustre or Signal have been proposed as a tool for programming reactive systems. In these two languages, the succession of elements in a stream is tightly coupled with the concept of time: the evaluation order of the elements in a

stream is the same as the order of occurrence of the elements in the stream [12]. This is not true in Lucid, where the computation of element i in a stream A may require the computation of an element $j > i$ in A. In addition, synchronization between occurrences of events in different streams is a main concern in Lustre and Signal. Lustre and Signal rely on a *clock analysis* to ensure that synchronous expressions receive their arguments at the same time (see [13] and [14] for a general introduction to synchronous programming). For example, the expression $A + B$ where A and B are streams, is allowed in Lustre or Signal only if the production of the elements in A and B takes place at the same instants (and so does the computation of the elements of $A + B$). This requires that the streams A, B and $A + B$ share a common reference in time: a *clock*. Timed flow, synchrony, together with a restriction on stream expressions to ensure bounded memory evaluation [15], make Lustre and Signal especially suitable tools to face real-time applications.

1.2 Sequential Streams

Sequential streams in $8_{1/2}$ share with the previous approach the idea of comparing the order of occurrence of events in different streams. But, in contrast with the previous approach, the expression $A + B$ is always allowed in $8_{1/2}$ emphasizing on a single common global time. The instants of this time are called *ticks*. The $8_{1/2}$ clock of the stream is specified by the sequence of ticks where a computation must occurs to ensure that, at each instant of the global clock, the relationship between the instantaneous values of the streams A, B, and $A + B$ is satisfied. Given streams A and B, it suffices to recompute the value of $A + B$ whenever a change happen to A or to B. The value of a stream can be observed at any time and this value is the value computed at the last change.

The idea of a clock in $8_{1/2}$ corresponds more closely to the time where values are computed rather than to the time when they must be consumed. In addition, a stream value can be accessed at any time. This makes $8_{1/2}$ unable to express real-time synchronization constraints (for example, asserting that two streams must have the same clock, like the *synchro* primitive in Signal), but makes more easy arbitrary combinations of *trajectories* in the simulation of dynamical systems [16]. We call $8_{1/2}$ streams *sequential streams* to stress that they have a strict temporal interpretation of the succession of the elements in the stream (like Lustre and Signal and unlike Lucid) without constraining to synchronous expressions.

1.3 Compiling Recursive Stream Equations into a Loop

The clock of a synchronous stream is a temporal predicate which asserts that the current value of the stream is changing. The inference, at compile time, of the clock of a stream makes the compiler able to check for consistencies (for example no temporal shortcuts between stream definitions) and to generate straight code for the computation of the stream values instead of using a more expensive demand-driven evaluation strategy.

Compiling a set of recursive stream equations consists in generating the code that enumerates the stream values in a strict ascending order. The idea is just to wrap a loop, that enumerates the ticks, around the guarded expression that computes the stream values at a given tick. This is possible because we only admit operators on streams satisfying a preorder restriction. The problem is to derive a static scheduling

of the computations and to generate an efficient code for the guards corresponding to the clocks of the stream expressions.

Structure of the paper. In the following section, we sketch $\mathcal{L}$ a declarative language on sequential streams. In section 3 we give a denotational semantics of $\mathcal{L}$ based on an extension of Kahn's original semantics for dataflow networks [17]. The main difference between our semantics and that of Plaice [18] or Jensen [19] relies in a simpler presentation of clocks. Moreover, our proposition satisfies a property of consistency between clock and stream values: if the clock ticks, the corresponding stream value is defined. Section 4 presents the translation of the clock definitions from the denotational semantics to a boolean expression using `C` as the target language. The process involves the resolution of a system of boolean expressions. Section 5 presents a benchmarks corresponding to the performances of a $8_{1/2}$ program compiled using the previous tools compared to an equivalent hand-coded `C` program: it compares quite well. Finally, section 6 examines related works.

2 Recursively Defined Sequential Streams

Conventions. We adopt the following notations. The value of a stream is a function from a set of instants called *ticks*, to values called *scalar values*. We restrict ourself in this paper to totally ordered unbounded countable set of ticks and therefore we use $\mathbb{N}$ to represent this set ([20] and [21] show possible uses of a partially ordered set of instants). The *current value* of a stream A refers to the scalar value at some tick t and is denoted by $A(t)$. The current value of a stream may be undefined, which is denoted by *nil*. A sequential stream is more than a function from ticks to scalar values: we have to represent the instants where a computation takes place to maintain the relationship asserted by the definition of the stream. The set of ticks characterizing the activity of the stream A is called its *clock* and written $cl(A)$. For $t \neq 0$, if $t \notin cl(A)$, then $A(t) = A(t-1)$ because no change of value occurs and therefore the current value is equal to the previous value. If $0 \notin cl(A)$, then $A(0) = nil$. So, a stream A described by $>\ ;\ ;\mathtt{1};\ ;\mathtt{2};\ ;\mathtt{3};\ \ldots >$ means that $A(0) = nil$, $A(1) = nil$, $A(2) = 1$, $A(3) = 1$, $A(4) = 2$, $A(5) = 2$, $A(6) = 3$, *etc.* The clock of A is the set $cl(A) = \{\, 2, 4, 6 \ldots \}$. With this notation, ticks are separated by "`;`" and a value is given only if the corresponding tick is in the clock of the stream.

2.1 A Sequential Stream Algebra

The language $\mathcal{L}$ represents the core of $8_{1/2}$ w.r.t the definition of streams. The set of expressions in $\mathcal{L}$ is given by the grammar:

$$e ::= c \mid \texttt{Clock}\, n \mid x \mid e_1\, op\, e_2 \mid \$e \mid e_1\ \texttt{when}\ e_2$$

where c ranges over integer and boolean constants (interpreted as constant streams), n ranges over $\mathbb{N}$, x over the set of variables Id and op over integer and boolean operations such as $+, \wedge, ==, <$ etc.

Constant streams. Scalar constants, like 0 or *true*, are overloaded to denote also a constant stream with clock reduced to the singleton $\{0\}$ and current value always equal to the scalar c: $c(t) = c$. A construct like `Clock` n represents a predefined boolean stream with current value always equal to *true* and with an unbounded clock (the precise clock is left unspecified).

Arithmetic expressions. An expression like $e_1 \, bop \, e_2$ extends the scalar operator *bop* to act in a point-wise fashion on the elements of the stream: $\forall t$, $(A \, bop \, B)(t) = A(t) \, bop \, B(t)$. The clock of $A \, bop \, B$ is the set of ticks t necessary to maintain this assertion (in a first approximation, it is the union of the clocks of A and B, Cf. section 3).

Delay. The delay operator \$, is used to shift "in time" the values of an entire stream. It gives access to the previous stream value. This operator is the only one that does not act in a point-wise fashion. Consequently, only past values can be used to compute a current stream value, and references to past values are relative to the current one. So, only the last p values of a stream have to be recorded where p is a constant computable at compile time. This restriction enables a finite memory assumption and enforces a temporal interpretation of the sequence of elements in a stream.

Sampling. The `when` operator is a trigger, corresponding to the temporal version of the `if then else` construct. It appears also in Lustre and Signal. The values of the stream A `when` B are those of A sampled at the ticks where B takes the value *true* (Cf. Tab. 1).

Table 1. Some examples of streams expression.

$1 :>$	1;	;	;	;	; ... >
$true :>$	true;	;	;	;	; ... >
`Clock` $2 :>$	true;	;	true;	;	true; ... >
1 `when Clock` $2 :>$	1;	;	1;	;	1; ... >
\$1 $:>$	;	;	;	;	; ... >
\$(1 `when Clock` 2) $:>$	;	;	1;	;	1; ... >

2.2 Recursively Defined Sequential Streams

A stream definition in $\mathcal{L}$ is given through an equation $x = e$ where x is a variable and e a stream expression. This definition can be read as an equation being satisfied between x and the stream arguments of e.

A definition can be guarded to indicate that it is valid only for some ticks:

$$A@0 = 33, \quad A = (\$A + 1) \texttt{ when Clock } 0 \ .$$

The first equation is guarded by @0 which indicates that this equation is only valid for the first tick in the clock of A (that is, the first tick of A is also the first tick of the constant stream 33, which is the tick $t = 0$). The second equation is "universally" quantified and defines the stream when no guarded equation applies. In this paper, the only language we consider for temporal guards is $@n$ where n is an integer which denotes the nth tick in a clock. A $\mathcal{L}$ program is a set of such definitions (i.e. guarded or non-guarded equations). For a given identifier x, there can only be a single universally quantified equation and at most one equation quantified by n.

An example of a reactive system using sequential streams. A "wlumf" is a "creature" whose behavior (mainly eating) is triggered by the level of some internal state (see [22] for such model in ethological simulation) More precisely, a wlumf is *hungry* when its *glycaemia* subsides under the level value 3. It can *eat* when there is some *food* in its environment. Its metabolism is such that when it eats, the glycaemia goes up to the level 10 and then decreases to zero at a rate of one unit per time step. Essentially, a wlumf is made of counters and flip-flop triggered and reset at different rates. The operator $\{\ldots\}$ is used to group sets of logically related stream definitions but we shall not be concerned with this aspect of the language for the rest of the paper .

```
System wlumf = {
    hungry@0 = false
    hungry = glycemia < 3
    glycemia@0 = 6
    glycemia = if eat then 10
      else max(0, $glycemia−1)when(Clock 0) fi
    eat@0 = false
    eat = $hungry && environment.food
}
```

```
System environment = {
    t@0 = 0
    t = $t + 1 when Clock − 4
    food = (0 == (t%2))
}
```

Fig. 1. The dynamical behaviour of an artificial creature, the "wlumf". The operator % is for modulo and == for testing equality. So $food$ is $true$ or $false$ depending on the parity of the counter t which progresses randomly at an average rate of 1/4. The operator && is the logical and.

3 A Denotational Semantics for $\mathcal{L}$

For the sake of simplicity, we assume that guarded equations are only of the form $x@0 = e$. Therefore, we replace a definition $x@0 = e_1,\ x = e_2$ by a single equation $x = e_1$ `fby` e_2 where `fby` is a new operator waiting for the first tick in the clock of e_1 and then switching to the stream e_2. The denotational semantics of $\mathcal{L}$ is based on an extension of Kahn's original semantics for dataflow networks [17]. The notations are slightly adapted from [23].

3.1 Stream Values and Clocks

The basic domain consists of finite and infinite sequences over the sets of integer and boolean values extended with the value nil to represent the absence of a value: $\textsc{ScValue} = Bool \cup Int \cup \{nil\}$ and $\textsc{Value} = \textsc{ScValue}^* \cup \textsc{ScValue}^\infty$. The operation "." denotes the concatenation of finite or infinite sequences. In Value, u approximates v, written $u \preceq v$ if $v = u.w$. This order is chosen against the more general Scott order (e.g. used for defining domains of functions [23]) in accordance with our interpretation of the succession of elements in the stream as the progression in time of the evaluation process.

A first idea to describe timed stream is to associate to the sequence of values, a sequence of boolean flags telling if an element is in the clock of the stream (flag true: $\top$) or not (flag false: $\bot$). In other words, a sequence of booleans $\{\bot, \top\}$ is used to represent $cl()$. For example, the sequence representing the clock of $\texttt{Clock}\,2$ is: $\top\ \bot\ \top\ \bot\ \top\ \ldots$ Thus: $\textsc{ScClock} = \{\bot, \top\}$ and $\textsc{Clock} = \textsc{ScClock}^* \cup \textsc{ScClock}^\infty$. We choose to completely order $\textsc{ScClock}$ by $\bot < \top$. The motivation to completely order the domain $\textsc{ScClock}$ is the following: there is no particular reason for a stream definition evaluating into a sequence of undefined values, not to have a defined clock (with no true values). Moreover, if we cannot evaluate the current value of a clock, we obviously cannot evaluate the current value of the corresponding stream and this is observationally equivalent to the value $\bot$ in the clock sequence.

By convention, if s is a $\textsc{Clock}$, then $t \in s$ means $s(t) = \top$ and $t \notin s$ means $s(t) = \bot$. We extend the logical or $\vee$ by $\veebar$ and the logical and $\wedge$ by $\barwedge$ to operate point-wise on $\textsc{Clock}$: that is, $(s \barwedge s')(t) = s(t) \wedge s'(t)$ and $(s \veebar s')(t) = s(t) \vee s'(t)$). The ordering of clocks is also the prefix ordering.

In the work of Plaice [18] or Jensen [19], the definition of the clock of a stream is loosely coupled with the value of the stream, in the following sense: a tick can be in the clock of a stream while the current value of the stream is undefined. The simplest example is the expression $\$e$ which has the same clock of e but with an undefined value for the first tick in $cl(e)$. On the contrary, we ask for a denotational semantics that ensures that:

$$t \in cl(e) \Rightarrow e(t) \neq nil \tag{1}$$

A property like (1) is natural and certainly desirable but cannot be directly achieved. This is best shown on the following example. Consider the stream defined by:

$$A = 1\ \texttt{fby}\ ((\$A + 1)\ \texttt{when}\ (\texttt{Clock}\,0)) \tag{2}$$

which is supposed to define a counter increasing every ticks. But, if we assume property (1), then $cl(A)$ can be proved to be $\{0\}$. As a matter of fact, $0 \in cl(A)$ because $0 \in cl(1)$ and obviously the first tick in $cl(e)$ is also in $cl(e\,\texttt{fby}\,e')$. Furthermore, a delayed stream $\$e$ cannot have a defined value the first time e has a defined value. So, using property (1), it comes that $0 \notin cl(\$A)$. Furthermore, the value of $e\ \texttt{when}\ clock0$ is defined only when e has a defined value. So, again using property (1), we infer that

$$cl(A) = \top.Ok(cl(\$A + 1)) = \top.Ok(cl(\$A)) \tag{3}$$

where the predicate Ok tells if the clock has already ticked: $Ok(\bot.s) = \bot.Ok(s)$ and $Ok(\top.s) = True$ (the sequence $True$ is the solution of the equation $True = \top.True$). The clock of $\$A$ depends of the clock of A and more precisely, except for the first tick in $cl(A)$, we have $cl(\$A) = cl(A)$. So, for $t \neq 0$, equation (3) rewrites in:

$$t \neq 0, \quad cl(A)(t) = Ok(cl(A))(t) \tag{4}$$

Equation (4) is a recursive equation with solutions in $\textsc{Clock}$. This equation admits several solutions but the least solution, with respect to the structure of $\textsc{Clock}$, is $cl(A) = \top.False$ (where $False = \bot.False$). This is a problem because we expect the solution $True$.

The collapse of the clock is due to the confusion of two predicates : "having a definite value at tick t" and "changing possibly of value at tick t". Then, to develop a denotational semantics exhibiting a property similar to (1), our idea is to split the clock of a stream A in two sequences $\mathcal{D}(A)$ and $\mathcal{C}(A)$ with the following intuitive

interpretation: $\mathcal{D}(A)$ indicates when the first non *nil* value of A becomes available for further computations and $\mathcal{C}(A)$ indicates that some computations are necessary to maintain the relationship asserted by the stream definition.

3.2 Semantics of Expressions

We call environment a mapping from variables to CLOCK or VALUE. An element ρ of ENV is a mapping ID $\to$ CLOCK $\times$ CLOCK $\times$ VALUE. Such an element really represents three environments linking a variable to the two sequences representing its clock and the sequence representing its value.

The semantics of $\mathcal{L}$ expressions is defined by the three functions:

$$\mathcal{D}[\![\,]\!],\ \mathcal{C}[\![\,]\!],\ \mathcal{V}[\![\,]\!]\ : \text{EXP} \to \text{ENV} \to \text{VALUE}\ .$$

The reason of using an element of ENV instead of an environment, is the value of an expression involving variable may depend of the clocks $\mathcal{D}[\![\,]\!]$ and $\mathcal{C}[\![\,]\!]$ of this variable. By convention, if $\rho \in$ ENV, then ρ_d, ρ_c and ρ_v represents the components of ρ, that is: $\rho(x) = (\rho_d(x), \rho_c(x), \rho_v(x))$. In addition, we omit the necessary injections between the basic syntactic and semantic domains when they can be recovered from the context.

A constant c denotes the following three sequences:

$$\mathcal{D}[\![c]\!]\rho = True, \quad \mathcal{C}[\![c]\!]\rho = \top.False, \quad \mathcal{V}[\![c]\!]\rho = c^\infty,$$

where c^∞ denotes an infinite sequence of c's, i.e. $c^\infty = c.c^\infty$. The intuitive meaning is that the current values of a constant stream are available from the beginning of time, a computation being needed only at the first instant to build the initial value of the constant stream and the current values being all the same. Some other constants are needed if we want to have streams with more than singleton clocks. This is the purpose of the constant stream `Clock` n which has an unbounded clock:

$$\mathcal{D}[\![\texttt{Clock}\,n]\!]\rho = True, \quad \mathcal{C}[\![\texttt{Clock}\,n]\!]\rho = dev(n), \quad \mathcal{V}[\![\texttt{Clock}\,n]\!]\rho = True,$$

where $dev(n)$ is some device computing a boolean sequence depending on n, beginning by $\top$ and with an unbounded number of $\top$ values. Variables are looked up in the corresponding environment:

$$\mathcal{D}[\![x]\!]\rho = \rho_d(x), \quad \mathcal{C}[\![x]\!]\rho = \rho_c(x), \quad \mathcal{V}[\![x]\!]\rho = \rho_v(x)\ .$$

The predefined arithmetic and logical operators are all strict:

$$\begin{aligned}
\mathcal{D}[\![e_1\ bop\ e_2]\!]\rho &= \mathcal{D}[\![e_1]\!]\rho \mathbin{\underline{\wedge}} \mathcal{D}[\![e_2]\!]\rho \\
\mathcal{C}[\![e_1\ bop\ e_2]\!]\rho &= \mathcal{D}[\![e_1\ bop\ e_2]\!]\rho \mathbin{\underline{\wedge}} (\mathcal{C}[\![e_1]\!]\rho \mathbin{\underline{\vee}} \mathcal{C}[\![e_2]\!]\rho) \\
\mathcal{V}[\![e_1\ bop\ e_2]\!]\rho &= \mathcal{V}[\![e_1]\!]\rho\ bop\ \mathcal{V}[\![e_2]\!]\rho
\end{aligned}$$

that is, the value of $e_1\ bop\ e_2$ can be computed only when both e_1 *and* e_2 have a value. This value changes as soon as e_1 *or* e_2 changes its value, when both are defined. Notice that the definition of $\mathcal{C}[\![e]\!]\rho$ takes the form $\mathcal{D}[\![e]\!]\rho \mathbin{\underline{\wedge}} (\ldots)$ in order to ensure the property:

$$\forall t,\ \mathcal{C}[\![e]\!]\rho(t) \Rightarrow \mathcal{D}[\![e]\!]\rho(t) \tag{5}$$

(Cf. section 3.3). For a delayed stream, the equations are:

$$\begin{aligned}
&\mathcal{D}[\![\$e]\!]\rho = delD(\mathcal{D}[\![e]\!]\rho), \qquad \mathcal{C}[\![\$e]\!]\rho = \mathcal{D}[\![\$e]\!]\rho \mathbin{\underline{\wedge}} \mathcal{C}[\![e]\!]\rho \\
&\mathcal{V}[\![\$e]\!]\rho = delV(nil, nil; \mathcal{V}[\![e]\!]\rho, \mathcal{C}[\![e]\!]\rho)
\end{aligned}$$

where $delD$ and $delV$ are auxiliary functions defined by (s, s'' are sequences and p, p' are scalar values $\neq nil$):

$$\begin{aligned}
delD(\bot.s) &= \bot.delD(s) \\
delD(\top.s) &= \bot.s \\
delV(nil, nil; v.s, \bot.s') &= nil.delV(nil, nil; s, s') \\
delV(nil, nil; v.s, \top.s') &= nil.delV(v, v; s, s') \\
delV(p, p'; v.s, \bot.s') &= p.delV(p, p'; s, s') \\
delV(p, p'; v.s, \top.s') &= p'.delV(p', v; s, s')
\end{aligned}$$

In other words, if t is the first tick for which A has a defined value, then the value of $\$A$ becomes available at $t + 1$. The computation needed for $\$A$ takes place at the same instants, as for A, except the first instant, and the values are shifted in time accordingly.
The sampling operator is specified by:

$$\begin{aligned}
\mathcal{D}[\![e_1 \texttt{ when } e_2]\!]\rho &= \mathcal{D}[\![e_1]\!]\rho \barwedge \mathcal{D}[\![e_2]\!]\rho \\
\mathcal{C}[\![e_1 \texttt{ when } e_2]\!]\rho &= \mathcal{D}[\![e_1 \texttt{ when } e_2]\!]\rho \barwedge (\mathcal{C}[\![e_2]\!]\rho \barwedge \mathcal{V}[\![e_2]\!]\rho) \\
\mathcal{V}[\![e_1 \texttt{ when } e_2]\!]\rho &= trigger(nil; \mathcal{V}[\![e_1]\!]\rho, \mathcal{C}[\![e_1 \texttt{ when } e_2]\!]\rho)
\end{aligned}$$

where $trigger$ is defined as:

$$\begin{aligned}
trigger(p; v.s, \bot.s') &= p.trigger(p; s, s') \\
trigger(p; v.s, \top.s') &= v.trigger(v; s, s')
\end{aligned}$$

The value of the sampling operator can be defined only when both operands are defined. The clock is defined by the (sub)clock of e_2 when e_2 takes the value $\top$.
Finally, the `fby` construct takes the first defined element in its first argument and then "switches" to its second argument:

$$\begin{aligned}
\mathcal{D}[\![e_1 \texttt{ fby } e_2]\!]\rho &= \mathcal{D}[\![e_1]\!]\rho \\
\mathcal{C}[\![e_1 \texttt{ fby } e_2]\!]\rho &= \mathcal{D}[\![e_1]\!]\rho \barwedge fbyC(\mathcal{C}[\![e_1]\!]\rho, \mathcal{C}[\![e_2]\!]\rho) \\
\mathcal{V}[\![e_1 \texttt{ fby } e_2]\!]\rho &= fbyV(\mathcal{C}[\![e_1]\!]\rho, \mathcal{V}[\![e_1]\!]\rho, \mathcal{V}[\![e_2]\!]\rho)
\end{aligned}$$

where:

$$\begin{aligned}
fbyC(\bot.s, b.s') &= \bot.fbyC(s, s') \\
fbyC(\top.s, b.s') &= \top.s' \\
fbyV(\bot.w, v.s, v'.s') &= v.fbyV(w, s, s') \\
fbyV(\top.w, v.s, v'.s') &= v.s'
\end{aligned}$$

3.3 Semantics of Programs

The semantics of a set of recursive equations $\{\ldots, x_i = e_i, \ldots\}$ is composed of an element $\rho \in \textsc{Env}$ assigning domain, clock and values to each stream variables x_i in the program. It can be computed as the least fixed point of the function

$$F(\rho) = [\ldots,\ x_i \mapsto (\mathcal{D}[\![e_i]\!]\rho,\ \mathcal{C}[\![e_i]\!]\rho,\ \mathcal{V}[\![e_i]\!]\rho),\ \ldots]$$

where $[\ldots, x \mapsto v, \ldots]$ stands for an environment which maps x to v. All auxiliary functions involved are monotone and continuous. Then, the fixed point can be calculated in the standard way as the least upper bound of a sequence of iterations F^n starting from the empty environments. We write $(\mathbf{D}(x), \mathbf{C}(x), \mathbf{V}(x))$ for the value associated to x in the meaning of a program.

The simple form of the semantics may accommodate several variations to specify other stream algebra. The affirmation (5) holds for any environment ρ, and then it holds also for the fixpoint:

$$\forall t,\ \mathbf{C}(e)(t) \Rightarrow \mathbf{D}(e)(t) \tag{6}$$

A proof by induction on the structure of an expression shows that a property similar to (1) holds between $\mathcal{C}[\![e]\!]$ and $\mathcal{V}[\![e]\!]$ for any expression e in a program: $\forall t,\ \mathbf{C}(e)(t) \Rightarrow \mathbf{V}(e)(t) \neq nil$. Another result will be extremely useful for the implementation. Once defined, the current value of a stream may change on tick t only if the clock of the stream takes the value $\top$ at t:

$$\forall t,\ \mathbf{D}(e)(t-1) \wedge \mathbf{V}(e)(t-1) \neq \mathbf{V}(e)(t) \Rightarrow \mathbf{C}(e)(t) \tag{7}$$

the proof is by induction on terms in $\mathcal{L}$.

Example of a counter. As an example, we consider the semantics of the clock of the program (2). We assume that $dev(0) = True$. The semantics of the counter A is defined by the following equations:

$$\begin{aligned}
\mathbf{D}(A) &= \mathbf{D}(1\ \texttt{fby}\ (\$A + 1)\ \texttt{when Clock}\ 0)) = \mathbf{D}(1) = True \\
\mathbf{C}(A) &= \mathbf{C}(1\ \texttt{fby}\ (\$A + 1)\ \texttt{when Clock}\ 0)) \\
&= fbyC(\mathbf{C}(1), \mathbf{C}((\$A + 1)\ \texttt{when Clock}\ 0)) \\
&= fbyC(\top.False, \mathbf{D}((\$A + 1\ \texttt{when Clock}\ 0) \mathbin{\underline{\wedge}} (\mathbf{C}(\texttt{Clock}\ 0) \wedge True))\ .
\end{aligned}$$

We have $\mathbf{D}((\$A+1)\ \texttt{when Clock}\ 0) = \mathbf{D}(\$A+1) \mathbin{\underline{\wedge}} \mathbf{D}(\texttt{Clock}\ 0) = \mathbf{D}(\$A+1) = \mathbf{D}(\$A) \mathbin{\underline{\wedge}} True = \mathbf{D}(\$A) = \bot.True$ because $\mathbf{D}(A) = True$. So, as expected:

$$\begin{aligned}
\mathbf{C}(A) &= True \mathbin{\underline{\wedge}} fbyC(\top.False, \bot.True \mathbin{\underline{\wedge}} (\mathbf{C}(\texttt{Clock}\ 0) \wedge True)) \\
&= fbyC(\top.False, \bot.True) = True\ .
\end{aligned}$$

4 Compiling Recursive Streams into a Loop

We implement a sequence s as *the successive values of one memory location* associated with (the current value of) s. We emphasize that successive means here successive *in time*. The idea is to translate a set of equations $\{\ldots, x = e, \ldots\}$ into the imperative program (in a `C` like syntax):

```
for(;;)  { ...; xd = ed; xc = ec; xv = ev; ...; }
```

where $\texttt{x}_\texttt{d}$ is associated to the current value of $\mathbf{D}(x)$, etc. This implementation is far from the representation needed for Lucid (or for the lazy lists of Haskell) where several elements of a sequence can be present at the same time in the memory so that a garbage collector is involved to remove useless elements from the memory.

With the denotational semantics defined above, this representation implies the update of the three memory locations at each tick (i.e. for each element in the sequence). However, property (6) implies that it is sufficient to update the memory location representing $\mathbf{C}(e)$ only when $\mathbf{D}(e)$(t) evaluates to true. And property (7) implies that is

is sufficient to evaluate $\mathbf{V}(e)(\mathrm{t})$ when $\mathbf{C}(e)(\mathrm{t})$ evaluates to true. These two conditions are sufficient but not necessary (e.g. `Clock`0 has an unbounded clock but its current value is always $\top$). So, a $\mathcal{L}$ program can be translated into the following `C` skeleton:

`for(;;) { ...; if(`$\mathtt{x_d}$ `=` $\mathtt{e_d}$`) { if(`$\mathtt{x_c}$ `=` $\mathtt{e_c}$`) { `$\mathtt{x_v}$ `=` $\mathtt{e_v}$`; }} ...; }`

However, translating a set of definitions into imperative assignments is not straightforward because of the recursive definitions: how to evaluate fixed points of sequences expressions without 1) handling explicitly infinite sequences and 2) iterations. In the rest of the section, we will build the tools that are necessary for this translation.

4.1 LR(1) Functions

We say that a function $f : \textsc{ScValue} \times \textsc{Value}^n \to \textsc{Value}$ is LR(1) if:

$$f(m; v_1.s_1, \ldots, v_n.s_n) \;=\; f'(m, v_1, \ldots, v_n) \,.\, f(f''(m, v_1, \ldots, v_n); s_1, \ldots, s_n)$$

where f' and f'' are functions from scalar values to scalar values: $f', f'' : \textsc{ScValue}^{n+1} \to \textsc{ScValue}$. Being LR(1) means that computing f on sequences can be a left to right process involving only computation on scalars, with only one memory location, assuming that the arguments are also provided from left to right.

Suppose F is LR(1); to solve the equation $v.s = F(m; v.s)$ on sequences (v and s are unknown, m is a parameter) it is then sufficient to solve the equation

$$v = F'(m; v) \tag{8}$$

on scalars and then to proceed with the resolution of $s = F(F''(m, v); s)$. Thus we have to consider the two sequences:

$$v_i = F'(m_i; v_i), \qquad m_i = F''(m_{i-1}, v_{i-1}), \quad i \geq 1$$

obtained by enumerating the successive solutions of (8) starting from an initial value m_0. The sequence of v_i's is obviously a solution of $s = F(s)$ and moreover, it is the least solution for $\preceq$. The equation (8) is called the I-equation associated with the equation $s = F(s)$ (I stands for "instantaneous"). It is easy to show that all the functions involved in the semantics of an expression given in section 3 are LR(1). This provides the basis for the implementation of declarative sequential streams into an imperative code.

4.2 Guarded LR(1) Semantic Equations

It is easy to rephrase the semantic definition of each $\mathcal{L}$ construct given in section 3 to make explicit properties (6), (7) and LR(1). The semantic equations are rephrased in Fig. 2 but due to the lack of place, we omit to rephrase some auxilliary functions. We have explicitly stated the values for a tick t in order to give directly the expressions $\mathtt{e_d}$, $\mathtt{e_c}$ and $\mathtt{e_v}$ corresponding to `C` skeleton. The notation $s(t)$ refers to the tth element in sequence s, where element numbering starts 0. Semantics of systems remains the same. We will omit the tedious but straightforward proof by induction on terms to check that the two semantic definitions compute the same thing.

for commodity, let $\mathcal{D}[\![e]\!]\rho(-1) = \bot$, $\mathcal{C}[\![e]\!]\rho(-1) = \bot$, $\mathcal{V}[\![e]\!]\rho(-1) = nil$
for any expression e and environment ρ, and assume $t > -1$ below:

$\mathcal{D}[\![c]\!]\rho(t) = \top \qquad \mathcal{C}[\![c]\!]\rho(t) = (t == 0) \qquad \mathcal{V}[\![c]\!]\rho(t) = c$

$\mathcal{D}[\![\texttt{Clock}\, n]\!]\rho(t) = \top \qquad \mathcal{C}[\![\texttt{Clock}\, n]\!]\rho(t) = dev(n,t) \qquad \mathcal{V}[\![\texttt{Clock}\, n]\!]\rho(t) = \top$

$\mathcal{D}[\![x]\!]\rho(t) = \rho(x)(t) \qquad \mathcal{C}[\![x]\!]\rho(t) = \rho(x)(t) \qquad \mathcal{V}[\![x]\!]\rho(t) = \rho(x)(t)$

$\mathcal{D}[\![e_1\ bop\ e_2]\!]\rho(t) = \mathcal{D}[\![e_1]\!]\rho(t) \mathbin{\underline{\wedge}} \mathcal{D}[\![e_2]\!]\rho(t)$
$\mathcal{C}[\![e_1\ bop\ e_2]\!]\rho(t) = \mathit{if}\ \ \mathcal{D}[\![e_1\ bop\ e_2]\!]\rho(t)\ \ \mathit{then}\ \ \mathcal{C}[\![e_1]\!]\rho(t) \mathbin{\underline{\vee}} \mathcal{C}[\![e_2]\!]\rho(t)\ \ \mathit{else}\ \ \bot$
$\mathcal{V}[\![e_1\ bop\ e_2]\!]\rho(t) = \mathit{if}\ \ \mathcal{C}[\![e_1\ bop\ e_2]\!]\rho(t)\ \ \mathit{then}\ \ \mathcal{V}[\![e_1]\!]\rho(t)\ bop\ \mathcal{V}[\![e_2]\!]\rho(t)$
$\qquad\qquad \mathit{else}\ \ \mathcal{V}[\![e_1\ bop\ e_2]\!]\rho(t-1)$

$\mathcal{D}[\![\$e]\!]\rho(t) = \mathcal{D}[\![e]\!]\rho(t-1)$
$\mathcal{C}[\![\$e]\!]\rho(t) = \mathit{if}\ \ \mathcal{D}[\![\$e]\!]\rho(t)\ \ \mathit{then}\ \ \mathcal{C}[\![e]\!]\rho\ \ \mathit{else}\ \ \mathcal{C}[\![\$e]\!]\rho(t-1)$
$\mathcal{V}[\![\$e]\!]\rho(t) = \mathit{if}\ \ \mathcal{C}[\![\$e]\!]\rho(t)\ \ \mathit{then}\ \ delV(nil, nil; \mathcal{V}[\![e]\!]\rho, \mathcal{C}[\![e]\!]\rho)(t)\ \ \mathit{else}\ \ \mathcal{V}[\![\$e]\!]\rho(t-1)$

$\mathcal{D}[\![e_1\ \texttt{when}\ e_2]\!]\rho(t) = \mathcal{D}[\![e_1]\!]\rho(t) \mathbin{\underline{\wedge}} \mathcal{D}[\![e_2]\!]\rho(t)$
$\mathcal{C}[\![e_1\ \texttt{when}\ e_2]\!]\rho(t) = \mathit{if}\ \ \mathcal{D}[\![e_1\ \texttt{when}\ e_2]\!]\rho(t)\ \ \mathit{then}\ \ \mathcal{C}[\![e_2]\!]\rho(t) \mathbin{\underline{\wedge}} \mathcal{V}[\![e_2]\!]\rho(t)\ \ \mathit{else}\ \ \bot$
$\mathcal{V}[\![e_1\ \texttt{when}\ e_2]\!]\rho(t) = \mathit{if}\ \ \mathcal{C}[\![e_1\ \texttt{when}\ e_2]\!]\rho(t)\ \ \mathit{then}\ \ \mathcal{V}[\![e_1]\!]\rho(t)\ \ \mathit{else}\ \ \mathcal{V}[\![e_1\ \texttt{when}\ e_2]\!]\rho(t-1)$

$\mathcal{D}[\![e_1\ \texttt{fby}\ e_2]\!]\rho(t) = \mathcal{D}[\![e_1]\!]\rho(t)$
$\mathcal{C}[\![e_1\ \texttt{fby}\ e_2]\!]\rho(t) = \mathit{if}\ \ \mathcal{D}[\![e_1\ \texttt{fby}\ e_2]\!]\rho(t)\ \ \mathit{then}\ \ fbyC(\mathcal{C}[\![e_1]\!]\rho, \mathcal{C}[\![e_2]\!]\rho)(t)\ \ \mathit{else}\ \ \bot$
$\mathcal{V}[\![e_1\ \texttt{fby}\ e_2]\!]\rho(t) = \mathit{if}\ \ \mathcal{C}[\![e_1\ \texttt{fby}\ e_2]\!]\rho(t)\ \ \mathit{then}\ \ fbyV(\mathcal{C}[\![e_1]\!]\rho, \mathcal{V}[\![e_1]\!]\rho, \mathcal{V}[\![e_2]\!]\rho)(t)$
$\qquad\qquad \mathit{else}\ \ \mathcal{V}[\![e_1\ \texttt{fby}\ e_2]\!]\rho(t-1)$

Fig. 2. Semantics of $\mathcal{L}$ in an explicit LR(1) form.

4.3 *I*-system Associated with a Program

Each equation $x = F(x)$ in a $\mathcal{L}$ program is directly interpreted through the semantics of an expression, as three equations defining $\mathcal{D}[\![x]\!]$, $\mathcal{C}[\![x]\!]$ and $\mathcal{V}[\![x]\!]$, the images of x by the program meaning. Each right hand-side, written respectively $F_d[x]$, $F_c[x]$ and $F_v[x]$, corresponds to a LR(1) function and therefore can be decomposed into the F' and F'' forms. In order to implement the various environments simply as a set of memory locations, we write x_d, x_c and x_v for the current value of $\mathcal{D}[\![x]\!]$, $\mathcal{C}[\![x]\!]$ and $\mathcal{V}[\![x]\!]$ and x_{md}, x_{mc} and x_{mv} for the first argument in F'. The three I-equations associated with $x = F(\ldots)$ can then be rephrased as:

$$x_d = F'_d[x](x_{md}; \ldots), \quad x_c = F'_c[x](x_{mc}; \ldots), \quad x_v = F'_v[x](x_{mv}; \ldots)\ .$$

For each variable in the program there is one equation defining x_d, one for x_c and one for x_v. The expression defining x_c has the form: *if* x_d *then* $\ldots$ *else* x_{mc} and the expression defining x_v follow the pattern *if* x_c *then* $\ldots$ *else* x_{mv}, except for the constants. The variables x_{mc} and x_{mv} are in charge to record the value x_c or x_v at the previous tick (or equivalently, they denote the one-tick shifted sequence that appears in the right hand side of the semantic equations). The expressions "$\ldots$" that appear in the *if* *then* *else* expression are also LR(1) functions of the sequences x_{md}, x_{mc}, x_{mv}, x_d, x_c and x_v. Thus they may require some additional scalar variables x'_m.

The set of I-equations associated with a program is called the I-system associated with the program. Suppose we can solve an I-system, then a sketch of the code implementing the computation of a $\mathcal{L}$ program is given in Fig. 3.

```
data declarations corresponding to the x_d, x_c, x_v 's
data declarations corresponding to the x_mc, x_mv 's
for(;;) {
    solve the I-system and update the x_d, x_c, x_v 's
    update the x_md, x_mc, x_mv 's according to the function F''_...[x]
}
```

Fig. 3. Sketch of the code implementing the computation of a $\mathcal{L}$ program.

4.4 Solving Efficiently an *I*-system

The problem of computing the least fixed point of a set of equations on sequences has now be turned into the simpler problem of computing the least solution of the I-system, a set of equations between scalar values. A straightforward solution is to compute it by fixed point iterations. If l is the number of expressions in the program, the iterations must become stationary after at most l steps, because the scalar domains are all flat. The problem is that this method may require l steps (l can be large) and that each step may require the computation of all the l expressions in the program.

Consider the dependence graph of an I-system: vertices correspond to variables and an edge from x to y corresponds to the use of x in the definition of y. This graph may be cyclic if the given definitions are recursive. For instance in $a@0 = b, a = \$a$ `or` b which defines a signal a always *true* after the first true value in b, $\mathcal{C}[\![a]\!]$ depends of $\mathcal{C}[\![a]\!]$ (and also of $\mathcal{C}[\![b]\!]$ which imposes its clock).

Without recursive equations, solving the I-system is easily done by simple substitutions: a topological sort can be used to order the equations at compile time. Non strict operators, like the conditional expression `if...then...else...`, can rise a problem because they induce a dependence graph depending on the value of the argument, value which is known only at evaluation time. Most of the time, it is possible to consider the union of the dependence graphs without introducing a cycle (which enables a static ordering of the equations). For the remaining rare cases, more sophisticated techniques, like conditional dependence graphs [24], can be used to infer a static scheduling. Solving the sorted system reduces to compute, in the order given by the topological sort, each right hand side and update the variables in the left hand side. In addition, the environment is implicitly implemented in the $x_d, x_c, x_v, \ldots$ variables.

For cyclic dependence graphs, the vertices can be grouped by *maximal strongly connected components*. The maximal strongly components form an acyclic graph corresponding to a partition of the initial I-system into several sub-systems. We call this graph the c-graph of the system (c stands for "component"). A *root* in the c-graph is a minimal element, that is, a node without predecessor (because c-graphs are acyclic, at least a root must exist). Each root of the c-graph represents a valid sub-system of the I-system, that is, a system where all variables present are defined (this is because roots are minimal elements). The solution of the entire I-system can be obtained by solving the sub-systems corresponding to the roots, propagating the results and then iterating the process. The processing order of the components can be determined by a topological sort on the c-graph.

So, we have turned the problem of solving an I-system into the problem of solving a root, that is: solving a subsystem of the initial system that corresponds to a maximal strongly connected component without a predecessor. In a root, we make a distinction

between two kinds of nodes: the V-nodes corresponding to expressions computing the current value of some stream and the B-nodes generated by the computation of the current boolean value for the clock of some stream. It can be seen that if there is a cycle between V-nodes, there is also a corresponding cycle involving only B-nodes (because the computation of $\mathcal{D}[\![e]\!]$ and $\mathcal{C}[\![e]\!]$ involves the same arguments as the computation of $\mathcal{V}[\![e]\!]$ for any expression e).

First, we turn our attention on cycles involving only B-nodes: they correspond to $\lambda x.x$, $\wedge$, $\vee$ and *if then else* operations between SCCLOCK. We assume that the root is reduced, that is, each argument of a B-node is an output of another B-node in the root (e.g., expressions like $\top \wedge x$ are reduced to x before consideration). Then, the output of any node in the root reduces to $\bot$. This is because a B-node *op* is strict (i.e. $op(\ldots, \bot, \ldots) = \bot$). Consequently, the fixed point is reached after one iteration.

Now, we turn our attention on cycles involving only V-nodes. Circular equations between values result also in circular equations between domains and clocks. The associated clock then evaluates to $false$ so there is no need to compute the associated value (which therefore remains *nil*).

A cycle involving both V-nodes and B-nodes is not possible inside a reduced root because there is no operator that promotes a clock into a value (clocks are hidden objects to the programmer, appearing at the semantical and implementation levels only).

5 Evaluation

The approach described in this paper has been fully implemented in the experimental environment of the $8_{1/2}$ language [25–27] (available at `ftp://ftp.lri.fr/LRI/soft/archi/Softwares/8,5`). The current compiler is written in `C` and in `CAML`. It generates either a target code for a virtual machine implemented on a UNIX workstation or directly a straight `C` code (no run-time memory management is necessary).

To evaluate the efficiency of our compilation scheme, we have performed some tests. We have chosen to compare the sequential generated `C` code from the $8_{1/2}$ equations with the hand-coded corresponding `C` implementation (because the application domain of $8_{1/2}$ is the simulation of dynamical systems, tests include a standard example of the numerical resolution of a partial differential equation through an explicit scheme and an implementation of the Turing's equations of diffusion-reaction). We details the results of the benchmark for the numerical resolution of a parabolic partial differential equation governing the heat diffusion in a thin uniform rod (Cf. Tab. 2).

The mean execution time corresponding to the compiler generated code without optimization is about 2.9 times slower than the hand-written one. The slight variation of the ratio with the number of iterations (which is the tick at which the program stops) are explained by a cache effect [28].

Four optimizations can be made on the generated `C` code to improve the performances. The first two concern the management of arrays (array shifting instead of gather/scatter and array sharing instead of copying for the concatenation) and does not interfere with the stream compilation scheme.

The last two optimizations have to do with the management of streams. For each delay $\$F$ appearing in the $8_{1/2}$ code, a copy has to be performed. The current value of a stream F is copied as many times as F is referenced by the delay operator. So, the sharing of delay expressions removes the useless copies. Moreover, the copy of expressions referenced by a delay operator (x_d into x_{md}, etc.) can be time-consuming,

Table 2. The heat diffusion resolution. Each element represents the ratio of the generated code execution time by the hand-written one. They both have been compiled using the GNU `C` compiler with the optimization option set `-O`. The evaluation has been performed on a *HP 9000/705 Series* under the *HP-UX 9.01* operating system. The first number represents the ratio without high-level optimizations, the second with the four optimizations sketched. The ratio does not depend of the number of iterations, i.e. the number of stream elements that are computed, which shows the strict temporal nature of the stream evaluation scheme.

Number of iterations → / Size of the rod ↓	100	500	1000	5000	10000
10	5.66	5.13	4.87	4.96	4.93
	3.89	3.59	3.65	3.70	3.66
100	2.27	2.17	2.17	2.15	2.15
	1.34	1.26	1.26	1.25	1.25
1000	2.80	2.76	2.76	2.76	2.76
	1.10	1.09	1.08	1.08	1.08
10000	2.62	2.60	2.61	2.60	2.61
	1.01	1.01	1.01	1.00	1.01

especially when large arrays are manipulated. However, the copy of the value of a stream F is not required, under some conditions (a similar optimization is described in Lustre [29]). If these conditions are not met, it is however possible to discard the delay copy. But it is necessary to have a temporary variable associated with the stream $\$F$. This kind of delay optimization consists in the definition of a single variable for each of the streams F and $\$F$ and to alternatively let it play the role of F or $\$F$ (a similar optimization is proposed in Sisal [30]).

The second number in Tab. 2 underlines the impact of these improvements: the mean ratio decreases to 1.5. Actually, it goes as far as 1.1 if we do not take into account the tests for which the rod has less than 100 elements, that is a size such that control structures are not negligible. However, it must be noted that there is a large room for further optimizations. More benchmarks can be found in [28].

6 Conclusion

Denotational semantics of recursive streams goes back to [17]. Equivalence between the denotational semantics and the operational behavior of a dataflow networks is studied in the work of [31]. Denotational semantics of timed flow begins in the framework of Lustre with [32, 18]. A very general framework has been formulated in [33] but its sophistication makes its use uneasy. The work of Jensen [19] formalizes clock analysis in terms of abstract interpretation and extends the works of Plaice and Bergerand. We should mention the work of Caspi and Pouzet [34]: the clock calculus there is different than most other in not using fixpoints. Our proposal fills a gap left open in these approaches by providing a denotational semantics of clock tightly coupled with the denotational semantics of values. Notice that there is a great difference between our handling of time and the synchronous concept of time in reactive systems: our clocks indicates when the value of a stream has to be recalculated as a result of other

changes in the system, while clocks in reactive systems tells when the value of a signal is present.

If $\mathbf{D}(x)$ or $\mathbf{C}(x)$ reduces to $False$, there is no value produced in the sequence $\mathbf{V}(x)$. This situation is a kind of deadlock. Deadlocks detection in declarative stream definitions are studied in [35, 36] and for lazy lists in [37]. Thanks to the ROBDD [38] representation of clocks, it is possible to detect at compile-time some cases of such definitions. Clock reducing to $True$ can also be detected and their implementation optimized. Signal has developed a sophisticated clock calculus to solve clock equations (dynamical system over $Z/3Z$ and Grobner bases). This approach is powerful but computation consuming. Its extension to our own stream algebra is not obvious and must be carefully studied.

The transformation of stream expressions into loops is extensively studied in [3]. The expressions considered do not allow recursive definitions of streams. Our proposition handles this important extension as well as "off-line cycle" expressions and is based upon the formal semantics of the expressions. We share the preorder restriction, i.e.: the succession of stream elements must be processed in time ascending order (this is not the case in Lucid). We focus also on unbounded streams and therefore we do not consider operations like concatenation of bounded streams. The work in [39] considers the static scheduling of a class of dataflow graphs used in digital signal processing. The translation of a (recursive) stream definition into a (cyclic) dataflow graph is straightforward. Their propositions apply but are limited to the subset of "on-line" programs [40]. This restriction excludes the sampling operator and requires the presence of, at least, one delay on each cycle of the dataflow graph.

The benchmarks performed validate the approach used in the compilation of the clock expressions although all the needed optimizations are not currently implemented. When made by hand, the ratio between the `C` version and the $8_{1/2}$ version lies between 1.1 and 2.3 (in favor of `C`) for the benchmark programs. As an indication, the hand-written `C` program for the Turing example of diffusion-reaction has 60 lines of code whereas the $8_{1/2}$ program is only 15 lines long (which are the straight transcription of the mathematical equations governing the process). Thus the price to pay for high expressivity (declarative definition of high-level objects) is not always synonym of low efficiency provided that some carefully tuned optimization techniques are used. Nevertheless, the cost of the control structures cannot be neglected and several optimizations must be performed [27].

Acknowledgments. The authors wish to thank Jean-Paul Sansonnet, the members of the *Parallel Architectures* team in LRI and the anonymous reviewers for their constructive comments.

References

1. G. L. Tesler and H. J. Enea. A language design for concurrent processes. In *AFIPS Conference Proceedings*, volume 32, pages 403–408, 1968.
2. W. W. Wadge and E. A. Ashcroft. *Lucid, the Data flow programming language.* Academic Press U. K., 1985.
3. R. C. Waters. Automatic transformation of series expressions into loops. *ACM Trans. on Prog. Languages and Systems*, 13(1):52–98, January 1991.
4. W. W. Wadge and E. A. Ashcroft. Lucid - A formal system for writing and proving programs. *SIAM Journal on Computing*, 3:336–354, September 1976.

5. P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 178–188, Munich, West Germany, January 21–23, 1987. ACM SIGACT-SIGPLAN, ACM Press.
6. P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. Signal, a dataflow oriented language for signal processing. *IEEE-ASSSP*, 34(2):362–374, 1986.
7. M. C. Chen. A parallel language and its compilation to multiprocessor machines or VLSI. In *Principles of Programming Languages*, pages 131–139, Florida, 1986.
8. N. Schmitz and J. Greiner. Software aids in PAL circuit design, simulation and verification. *Electronic Design*, 32(11), May 1984.
9. S. D. Johnson. *Synthesis of Digital Designs from Recursion Equations.* ACM Distinguished Dissertations. ACM Press, 1983.
10. J.-L. Giavitto. A synchronous data-flow language for massively parallel computer. In D. J. Evans, G. R. Joubert, and H. Liddell, editors, *Proc. of Int. Conf. on Parallel Computing (ParCo'91)*, pages 391–397, London, 3-6 September 1991. North-Holland.
11. K. Chandy and J. Misra. *Parallel Program Design - a Foundation.* Addison Wesley, 1989.
12. J. A. Plaice, R. Khédri, and R. Lalement. From abstract time to real time. In *ISLIP'93: Proc. of the 6th Int. Symp. on Lucid and Intensional programming*, 1993.
13. A. Benveniste and G. Berry. Special section: Another look at real-time programming. *Proc. of the IEEE*, 79(9):1268–1336, September 1991.
14. N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer Academic publishers, 1993.
15. Paul Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94:125–140, 1992.
16. O. Michel, J.-L. Giavitto, and J.-P. Sansonnet. A data-parallel declarative language for the simulation of large dynamical systems and its compilation. In *SMS-TPE'94: Software for Multiprocessors and Supercomputers*, Moscow, 21-23 September, 1994. Office of Naval Research USA & Russian Basic Research Foundation.
17. Gilles Kahn. The semantics of a simple language for parallel programming. In *proceedings of IFIP Congress'74*, pages 471–475. North-Holland, 1974.
18. J. A. Plaice. *Sémantique et compilation de LUSTRE un langage déclaratif synchrone.* PhD thesis, Institut national polytechnique de Grenoble, 1988.
19. T. P. Jensen. Clock analysis of synchronous dataflow programs. In *Proc. of ACM Symposium on Partial Evaluation and Semantics-Based Program Evaluation*, San Diego CA, June 1995.
20. H. R. Andersen and M. Mendler. An asynchronous process algebra with multiple clocks. In D. Sannella, editor, *Programming languages and systems - ESOP'94*, volume 788 of *Lecture Notes in Computer Sciences*, pages 58–73, Edinburgh, U.K., April 1994. Springer-Verlag.
21. P.-A. Nguyen. Représentation et construction d'un temps asynchrone pour le langage 81/2, Avril-Juin 1994. Rapport d'option de l'Ecole Polytechnique.
22. Patti Maes. A bottom-up mechanism for behavior selection in an artificial creature. In Bradford Book, editor, *proceedings of the first international conference on simulation of adaptative behavior.* MIT Press, 1991.
23. P. D. Mosses. *Handbook of Theoretical Computer Science*, volume 2, chapter Denotational Semantics, pages 575–631. Elsevier Science, 1990.

24. A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.
25. O. Michel. Design and implementation of 81/2, a declarative data-parallel language. *Computer Languages*, 22(2/3):165–179, 1996. special issue on Parallel Logic Programming.
26. O. Michel, D. De Vito, and J.-P. Sansonnet. 81/2 : data-parallelism and data-flow. In E. Ashcroft, editor, *Intensional Programming II:Proc. of the 9th Int. Symp. on Lucid and Intensional Programming*. World Scientific, May 1996.
27. D. De Vito and O. Michel. Effective SIMD code generation for the high-level declarative data-parallel language 81/2. In *EuroMicro'96*, pages 114–119. IEEE Computer Society, 2–5September 1996.
28. D. De Vito. Semantics and compilation of sequential streams into a static SIMD code for the declarative data-parallel language 81/2. Technical Report 1044, Laboratoire de Recherche en Informatique, May 1996. 34 pages.
29. N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In Springer Verlag, editor, *3rd international symposium, PLILP'91, Passau, Germany*, volume 528 of *Lecture Notes in Computer Sciences*, pages 207–218, August 1991.
30. D. C. Cann and P. Evripidou. Advanced array optimizations for high performance functional languages. *IEEE Trans. on Parallel and Distributed Systems*, 6(3):229–239, March 1995.
31. A. A. Faustini. An operational semantics of pure dataflow. In M. Nielsen and E. M. Schmidt, editors, *Automata, languages and programing: ninth colloquium*, volume 120 of *Lecture Notes in Computer Sciences*, pages 212–224. Springer-Verlag, 1982. equivalence sem. op et denotationelle.
32. J.-L. Bergerand. *LUSTRE: un langage déclaratif pour le temps réel*. PhD thesis, Institut national polytechnique de Grenoble, 1986.
33. A. Benveniste, P. Le Guernic, Y. Sorel, and M. Sorine. A denotational theory of synchronous reactive systems. *Information and Computation*, 99(2):1992–230, 1992.
34. Paul Caspi and Marc Pouzet. Synchronous Kahn networks. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 226–238, Philadelphia, Pennsylvania, 24–26 May 1996.
35. W. W. Wadge. An extensional treatment of dataflow deadlock. *Theoretical Computer Science*, 13(1):3–15, 1981.
36. E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proc. of the IEEE*, 75(9), September 1987.
37. B. A. Sijtsma. On the productivity of recursive list definitions. *ACM Transactions on Programming Languages and Systems*, 11(4):633–649, October 1989.
38. R. E. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, August 1986.
39. K. K. Parhi and D. G. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding,. *IEEE Trans. on Computers*, 40(2), February 1991.
40. A. Aho, J. Hopcroft, and J. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

—oOo—