A Constructive State-based Semantics and Interpreter for a Synchronous Data-flow Language with State Machines

Jean-Louis Colaço, Michael Mendler, Baptiste Pauget, Marc Pouzet

A Constructive State-based Semantics and Interpreter for a Synchronous Data-flow Language with State machines

Application to the Language Scade

JEAN-LOUIS COLAÇO, Ansys, France
MICHAEL MENDLER, The University of Bamberg, Germany
BAPTISTE PAUGET, Ansys, France and Inria, France
MARC POUZET, ENS/PSL, France and Inria, France

Scade is a domain-specific synchronous functional language used to implement safety-critical real-time software for more than twenty years. Two main approaches have been considered for its semantics: (i) an *indirect collapsing semantics* based on a source-to-source translation of high-level constructs into a data-flow core language whose semantics is precisely specified and is the entry for code generation; a *relational synchronous semantics*, akin to Esterel, that applies *directly* to the source. It defines what is a valid synchronous reaction but hides, on purpose, if a semantics exists, is unique and can be computed; hence, it is not executable.

This paper presents, for the first time, an *executable*, state-based semantics for a language that has the key constructs of Scade all together, in particular the arbitrary combination of data-flow equations and hierarchical state machines. It can apply *directly* to the source language before static checks and compilation steps. It is *constructive* in the sense that the language in which the semantics is defined is a statically typed functional language with call-by-value and strong normalization, e.g., it is expressible in a proof-assistant where all functions terminate. It leads to a reference, purely functional, interpreter. This semantics is modular and can account for possible errors, allowing to establish what property is ensured by each static verification performed by the compiler. It also clarifies how causality is treated in Scade compared with Esterel.

This semantics can serve as an oracle for compiler testing and validation; to prototype novel language constructs before they are implemented, to execute possibly unfinished models or that are correct but rejected by the compiler; to prove the correctness of compilation steps.

The semantics given in the paper is implemented as an interpreter in a purely functional style, in OCaml.

CCS Concepts: • **Software and its engineering → Semantics**; **Interpreters**; **Formal language definitions**; **Compilers**; • **Computer systems organization → Embedded software**.

Additional Key Words and Phrases: Programming language, dynamic semantics, synchronous programming, embedded software.

ACM Reference Format:
Jean-Louis Colaço, Michael Mendler, Baptiste Pauget, and Marc Pouzet. 2023. A Constructive State-based Semantics and Interpreter for a Synchronous Data-flow Language with State machines: Application to the Language Scade. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (August 2023), 34 pages. https://doi.org/10.1145/3609131

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Embedded Software (EMSOFT), 2023. **Notice: this text extends the TECS paper with an appendix (august 2023).**
Authors' addresses: Jean-Louis Colaço, SBU, Ansys, Park Avenue - Bâtiment Emeraude 1, 9 Rue Michel Labrousse, Toulouse, 31100, France, Jean-Louis.Colaco@ansys.com; Michael Mendler, Faculty of Information Systems and Applied Computer Sciences, The University of Bamberg, Gutenbergstrasse 13, Bamberg, D-96050, Germany, michael.mendler@uni-bamberg.de; Baptiste Pauget, SBU, Ansys, Toulouse, France, Baptiste.Pauget@ansys.com and Inria, 2 Rue Simone IFF, Paris, 75012, France; Marc Pouzet, DIENS, ENS/PSL, 45 rue d'Ulm, Paris, 75230, France, Marc.Pouzet@ens.fr and Inria, Paris, France.

1 INTRODUCTION

Scade [27] is a domain-specific, purely functional programming language used for more than twenty years to implement certified real-time software, in avionics, railways, nuclear plants and cars (e.g., fly-by-wire, braking system, electrical engine). The designer writes an ideal deterministic and synchronous (*zero-delay*) model [9] where signals are infinite streams and systems are represented by synchronous (*length-preserving*) stream functions. At its core, Scade stems from Lustre [31], inheriting its semantics, programming style and some of its language primitives. On top of this core, the language provides richer programming constructs, e.g., by-case definitions for streams, reset, hierarchical state machines. Those constructs preserve the data-flow style of the core language, with a single statically scoped definition per stream variable and per instant. The compiler generates C and Ada code and verifies safety properties on the source program: that it is well typed, deterministic, deadlock free and can be implemented in bounded time and memory. These verifications, expressed as dedicated type systems, are part of the language specification and are prescriptive, i.e., when they fail, no code is generated.

The compiler complies with the highest certification standards (e.g., DO178C, level A of avionics) for critical software so that the generated code can be used without any further verification that the semantics is preserved. It is a development process-based certification as required by industry safety standards and it is supported by independent testing, not to be confused with a certified proof of correctness developed within a proof assistant, e.g., the CompCert [39] compiler for C and the Vélus compiler [16, 17] for Lustre. [1] While the certification credit is unvaluable, it limits the evolution of the language and its compiler: any change, even the smallest, has to be done carefully because it must pass again the certification process. This raises the following questions: can we define a complete semantics for a language that has Scade's core programming constructs, considered all at once, and serving as an oracle for automatic testing and validation during compiler development? to prototype new language constructs before they are implemented in the compiler; to execute unfinished programs or programs rejected by the compiler because they are too conservative; to prove the correctness of compilation steps?

To answers those questions, this paper presents a *constructive state-based executable semantics* which leads to a reference, purely functional, interpreter and is independent of a compiler.

1.1 The Landscape of Synchronous Languages Semantics

It may seem odd to introduce an interpreter for a synchronous language like Scade after such a long time! Indeed, the semantics of synchronous languages [5], and more widely, domain specific languages for control software, have been studied extensively since their introduction in the mid eighties [6, 38]. For lack of space, we only describe the main achievements. In brief, two families of semantics have been defined: *relational* or *functional* semantics, the later being either only denotational (possibly non executable) or denotational and constructive; for each of them, some are *direct* — they apply directly to the source — or *indirect*, that is, high-level operations collapse into simple ones or are translated into a lower-level language for which a semantics is defined precisely.

A *relational or logical* semantics was originally introduced for Esterel [12], Lustre [21] and Signal [7]. It is defined by a collection of predicates that express what is a valid synchronous reaction. Synchronous parallel composition of two processes is expressed elegantly (and magically) as the conjunction of two relations. It purposely hides all the scheduling machinery necessary to compute a reaction, hence when a reaction exists and is unique. A relational semantics was

[1] Both aim at building a correct software, the first by imposing an obligation of means going from specification to verification, the second by imposing an obligation of result. Where both agree is on the need to have a precise specification of the semantics and code generation.

defined for a language that has the main features of Scade [24], in particular the mix of data-flow equations and hierarchical state machines. A relational semantics fits remarkably well in a proof assistant for proving compiler transformations. It is used extensively in the formally verified Vélus compiler [16, 17]. Nonetheless, a relational semantics does not define an effective, correct and complete, algorithm: it is not *computational* [29] or *constructive* [11]. For example, it cannot be used as an oracle for compiler testing. Moreover, some programs are causally correct — they do react at every instant with a reaction that is unique — but are unreasonable and counter intuitive [10, 11]. Or they are simply impossible to compile into executable code. Existing relational semantics do not characterize well the set of correct programs w.r.t programs accepted by a compiler, unless adding side conditions [54], making it more complex. Finally, with a relational semantics, errors are not explicit: they all transparently correspond to *the program does not react*, that is, the program's behavior is *under-determined*. Adding auxiliary predicates to model and propagate errors is possible but the semantics loses the conciseness and simplicity of the initial formulation.

Alternative and complementary approaches define the semantics *indirectly* as the result of a compilation process. The compilation of Esterel into Boolean equations [11, 13], the compilation of Quartz [51] and SCCharts [56] or the compilation of Argos [42] and mode automata [44] into guarded data-flow equations are examples of indirect semantics. In [25], the authors propose a *collapsing* or *translation semantics* for a language that has the main features of Scade — hierarchical automata, reset, by-case definition, local blocks and data-flow equations — and a source-to-source translation into a language core made of clocked data-flow equations. The semantics is also indirect but stays within the same language, with high-level constructs expressed modularily in term of simpler ones. The language subset has a precise semantics [17, 21, 26] and is the entry for sequential code generation [14, 17, 32]. This approach through collapsing was implemented in Lucid Synchrone [49] and the qualified compiler of Scade [27] for which it was retrospectively a sound decision, in terms of compiler design, to pass the certification process and generate good code. It is formally proven with respect to a relational semantics and implemented in the Vélus compiler [18]. Yet, an indirect semantics does not answer the need of a reference semantics that applies directly to the source, is constructive and gives an alternative way to run programs.

Several functional and denotational semantics were also defined, initially for Lustre [30] or as simple interpreters [43]. A Lustre node defines a continuous function on streams in the sense of Kahn process networks [37] with a restriction to characterize processes that execute synchronously [15, 20, 22, 26]. By encoding streams as lazy data-structures, it is possible to define an executable semantics as a shallow embedding in Haskell [22], hence an interpreter, an approach exploited extensively for FRP [34, 46]. Nonetheless, a stream-based semantics does not characterize well nor ensure important properties like execution in bounded time and space. Moreover, it is not constructive in the sense that the semantic is not defined as a total function which, given the sequence of inputs, computes the sequence of outputs or possibly stops with an error. E.g., it may diverge (that is, the interpreter enters into an infinite loop), when the source program is not causal. A constructive denotational stream semantics was given in Coq [48] for Kahn process networks and it can be applied to Lustre. Yet, it does not give an interpreter that can be used in practice because the computation of the n-th output of a stream always restarts from the very begining. The idea of a constructive semantics was introduced first for pure Esterel [10, 29] but with a different idea, and it was later extended to several languages reminiscent of Esterel, like Quartz [51] and SCCharts [56]. In this semantics, every synchronous reaction is the result of a fix-point computation of a monotone micro-step transition function on a Complete Partial Order (CPO) which is of bounded height, hence can be computed in statically known bounded time. This idea of a fix-point computation done at every reaction was exploited in [23] to define the semantics of a synchronous functional language on streams, with a shallow embedding in Haskell, so that the fix-point is computed lazily. In [28],

the authors show how the fix-point semantics can account for cyclic circuits that are constructively causal in a block-diagram, hence a data-flow, synchronous language. The proposal of the present paper directly builds on all these works.

Are we done? Not yet. The three language kernels for which a direct constructive semantics was given — pure Esterel and related languages, synchronous block-diagrams and a functional language of streams — are very different. Taken separately, none of them address a full language that have the main constructs of Scade, all together, in particular the free combination of data-flow equations and hierarchical state machines, the modular reset, nested local blocks and by-case definitions.

1.2 Contribution: A Constructive State-based Functional Semantics

We propose a new semantics for a synchronous data-flow language that deals with the main constructs of Scade. The main contributions are the following. First, the semantics applies *directly to the source program* before any static check and program transformation; it is independent of the compiler and does not need any hypothesis or properties on programs. This semantics is *state-based*, that is, a stream expression e running in an environment ρ defines a state machine, that is, a pair made of a transition function f and an initial state s [36]:

$$[\![e]\!]_\rho = (f, s) \text{ where } f : S \to V_\perp \times S \text{ and } s : S$$

The stream interpretation of e is obtained by iterating f from s. Compared to a relational semantics, this semantics is *constructive* because $[\![.]\!]_.$ is a total and pure function which can be expressed in a strongly normalizing typed lambda-calculus, e.g., the programming language of Coq where all computations terminate. A reaction may produce the value $\perp$ (the minimum element of the CPO) that represents a deadlock or *causality error*. For every reaction, in presence of mutually recursive equations on streams, $[\![e]\!]_\rho$ computes the least fix-point of a monotone function on a CPO of bounded height, hence always reached in bounded time, with no approximation. This semantics treats the *main language features of Scade all together*, that is, stream functions and applications, mutually recursive stream definitions, local definitions with default and memorized values, by-case definitions for streams, reset and hierarchical state machines. We go a bit further, with two features that are not in Scade, a simple pattern matching construct and state machines where states can be parameterized [24], two language features implemented in Lucid Synchrone and Zélus [19]. This semantics is able to deal with *static and run-time errors* (e.g., type error, initialization error, clock errors) with very small changes. Finally, it leads to a *reference functional interpreter*, that is, $[\![.]\!]_.$ is implemented in a purely functional subset of OCaml from which an implementation in Coq was produced automatically [2].

The software development, including the Ocaml interpreter and examples are available at https://zelus.di.ens.fr/zrun/emsoft2023.

Paper Organization. The language kernel and its semantics are defined in Section 2. Then, we consider an extended language with three important constructs of Scade in Section 3 and adapt the semantics accordingly. Causality is discussed in Section 4. We conclude in Section 5.

[2] https://github.com/formal-land/coq-of-ocaml

2 A SYNCHRONOUS DATA-FLOW KERNEL AND ITS SEMANTICS

We consider the following language kernel written in an abstract syntax close to that of [24, 25]. Types do not need to be declared since the semantics applies to untyped programs.

$$
\begin{array}{lcl}
d & ::= & \texttt{let}\ f = e \mid \texttt{let node}\ f\ p = e \mid d\ d \\
p & ::= & () \mid x \mid x, ..., x \\
e & ::= & c \mid x \mid f\ (e, ..., e) \mid (e, ..., e) \mid \texttt{pre}\ e \mid e\ \texttt{fby}\ e \mid \texttt{let}\ E\ \texttt{in}\ e \mid \texttt{let rec}\ E\ \texttt{in}\ e \\
 & & \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{when}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{reset}\ e\ \texttt{every}\ e \\
E & ::= & p = e \mid E\ \texttt{and}\ E
\end{array}
$$

A program is a set of declarations (d), each defining the value of a global identifier. A declaration can define the value of an identifier ($\texttt{let}\ f = e$); it can define a stream function with name f, a list of parameters p and body e ($\texttt{let node}\ f\ p = e$) [3]. p denotes the list, possible empty ($()$) of parameters of a function. e stands for an expression. An expression can be a constant (c), e.g., an integer, a Boolean value, the value $()$, etc; a variable identifier (x), the application of a function f to a (possibly empty) list of arguments ($f\ (e_1, ..., e_n)$ with $n \geq 0$), a tuple ($(e_1, ..., e_n)$ with $n \geq 2$). Other operators are the un-initialized unit delay ($\texttt{pre}\ e$), the initialized unit delay ($e_1\ \texttt{fby}\ e_2$). The expression ($\texttt{let}\ E\ \texttt{in}\ e$) uses variables defined by the equation (E) that are local to e. The equations can be defined recursively, that is, variables on the left-hand side of E bind those on the right-hand side ($\texttt{let rec}\ E\ \texttt{in}\ e$). The language provides three other constructs: the conditional or selector $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$; at every instant, the three branches are evaluated and it selects the value of e_2 if the current of e_1 is true; that of e_3 if the current value of e_1 is false. The activation $\texttt{when}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ executes e_2 only when e_1 is true and e_3 only when e_1 is false. It is possible to reset an expression e_1 when a Boolean condition e_2 is true ($e_1\ \texttt{reset every}\ e_2$). Finally, an equation either defines the current values of some variables ($p = e$) or is the parallel composition of two equations ($E\ \texttt{and}\ E$). As we shall see in Section 2.3, the equations $(x, y) = (e_1, e_2)$ and $x = e_1\ \texttt{and}\ y = e_2$ define the same streams. Some operators like the initialization $e_1\ \texttt{->}\ e_2$ are omitted because they can be defined from the kernel:

$$e_1\ \texttt{->}\ e_2 = \texttt{if}\ (\texttt{true}\ \texttt{fby}\ \texttt{false})\ \texttt{then}\ e_1\ \texttt{else}\ e_2$$

This kernel does not provide operations that involve clocks, like the `when` and `merge`. Instead, it provides a built-in activation structure ($\texttt{when}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$). Below are simple examples (forward and backward Euler, a PI controller and two mutually recursive stream equations). [4]

```
let node forward_euler(h, x0, xprime) returns (x) x = x0 fby (x +. h *. xprime)
let node backward_euler(h, x0, xprime) returns (x) x = x0 -> pre(x) +. h *. xprime
let node pi(p, i, u) = p *. u +. backward_euler(h, 0.0, i *. u)
let node sin_cos(h) returns (sin, cos)
  sin = forward_euler(h, 0.0, cos) and cos = backward_euler(h, 1.0, -. sin)
```

2.1 Stream Functions as Sequential Mealy Machines

Lustre and Scade are data-flow languages, i.e., a signal is an infinite stream and a system is a stream function. *Streams* can be represented as values of type $stream(T) =_{df} \mathbb{N} \rightarrow T$ where $\mathbb{N}$ stand for natural numbers. A stream can also be interpreted operationally as a co-iterative [36] *stream process* [47] or a *concrete stream* [23] $CoF(f, s)$ made of a transition function $f : S \rightarrow T \times S$ and an initial state $s : S$.

$$CoF : \forall S, T.(S \rightarrow T \times S) \times S \rightarrow coStr(T, S)$$

[3] We only give an excerpt of what is treated in the companion implementation. For simplicity, we do not consider here the definition of combinational functions.

[4] The declaration $\texttt{let node}\ f\ (p)\ \texttt{returns}\ (q)\ E$ is equivalent to $\texttt{let node}\ f\ (p) = \texttt{let rec}\ E\ \texttt{in}\ q$. Operations `+.` and `*.` stand for the addition and multiplication on Floatting point numbers, written in infix form.

$$
\begin{array}{lcll}
{}^{s}const(c) & = & \lambda n.c & {}^{c}const(c) = CoF(\lambda s.(c,s),()) \\
{}^{s}extend(f)(e) & = & \lambda n.(f\,n)\,(e\,n) & {}^{c}extend(CoF(f,s))(CoF(e,se)) = \\
{}^{s}pre\,e & = & if\ n=0\ then\ nil & \qquad CoF\,(((\lambda s.let\ v_f, s' = f\ s\ in\ let\ v_e, se' = e\ se\ in \\
 & & else\ e(n-1) & \qquad\qquad (v_f\ v_e), (s', se')) \\
{}^{s}fby\ f\ e & = & if\ n=0\ then\ f(0) & \qquad , (s, se))) \\
 & & else\ e(n-1) & {}^{c}(pre\ CoF(f,s)) = CoF((\lambda(m,s).(m, f\ s)), (nil, s))
\end{array}
$$

Fig. 1. Lustre primitives with their stream and sequential representations

$coStr(T,S)$ is the type of *concrete streams*. Given a state s, $f\,s$ returns the current output and a new state. The infinite iteration produces the infinite sequence of values. Hence, a value $CoF(f,s)$ of type $coStr(T,S)$ defines a stream v such that $v = nth(CoF(f,s))$ with $v(n)$ the n-th element. Conversely, $concrete(.)$ builds a concrete stream:

$$
\begin{array}{lcl}
nth(CoF(f,s))(0) & =_{df} & let\ v, s' = f\ s\ in\ v \\
nth(CoF(F,s))(n) & =_{df} & let\ \_, s' = f\ s\ in\ nth(CoF(f,s'))(n-1) \\
concrete(v) & =_{df} & CoF(\lambda n.(v(n),(n+1)), 0)
\end{array}
$$

with: $nth(.)(.) : \forall T,S.coStr(T,S) \to stream(T)$ and $concrete(.) : \forall T.stream(T) \to coStr(T,\mathbb{N})$.

Definition 1 (Equivalence). *Two concrete streams $CoF(f,s)$ and $CoF(f',s')$ are equivalent iff they produce the same stream:*

$$nth(CoF(f,s)) = nth(CoF(f',s'))$$

We write $CoF(f,s) \cong CoF(f',s')$ for equivalence of concrete streams.

Taking $stream(x)(n) =_{df} nth(x)(n)$, $concrete(stream(x)) \cong x$ and $stream(concrete(x)) = x$.

Proving that two concrete streams are equivalent amounts at finding an inductive relation R between states that is verified on initial states and preserved by the application of the two transition functions, that is, if $R(s,s')$ then $(fst\ o\ f)(s) = (fst\ o\ f')(s')$ and $R((snd\ o\ f)(s), (snd\ o\ f')(s'))$. Appendix available at https://zelus.di.ens.fr/zrun/emsoft2023 illustrates its use for proving the correctness of source-to-source transformations like the normalization and static scheduling.

2.1.1 Core Data-flow Primitives. A minimal subset of Lustre primitives can be defined with three operations: $const(.)$ builds a constant stream; $extend(.)(.)$ applies a stream of functions to a stream of inputs point-wise; $pre\,.$ is the un-initialized unit delay. At the initial instant, it returns the value nil which stands for an "undefined (non initialized) value" [21]. $fby\,..$ is the initialized delay such that $fby\,e_1\,e_2$ is a short-cut for $e_1 \to pre\,e_2$. Stream definitions are given on the left of Figure 1 (the tag ^{s} is for the "stream" semantics; the tag ^{c} for the "concrete" semantics). Any function with arity n can be lifted into a stream function using $const(.)$ and $extend(.)(.)$. E.g., if f is a binary function, we can simply write $f\,x\,y$ as a short-cut for:

$$lift_2\,f\,x\,y = extend(extend(const(f))(x))(y)$$

The conditional $if\,.\,then\,.\,else\,.$ of *Lustre* (that we write below $ifstrict\,...$) is:

$$ifstrict\,c\,x\,y = extend(extend(extend(const(\lambda c, y, z.if\ c\ then\ y\ else\ z))(c))(x))(y)$$

For example, the semantics of the edge front detector defined below is $\lambda x.x \to (not\,(pre\,x))\,\&\,x$. If $x : stream(T)$ is a stream, it returns $x(0)$ at instant 0; at instant n, it returns true if $x(n)$ is true and $x(n-1)$ is false.

```
1  let node edge(x) = x -> not (pre x) & x
```

Remark 1 (Why having two interpretations for streams?). *It is easy to check that the two interpretations for primitives — the stream-based interpretation versus the state-based interpretation — given on the left and right in Figure 1 are equivalent. In particular that* $nth(^c pre\, x)(0) = nil$ *and* $nth(^c pre\, x)(n) = nth(x)(n-1)$*. Both interpretations are useful. The fact that one can go modularily from an interpretation to the other (using* $concrete(.)$ *and* $stream(.)$*) highlights the data-flow nature of a language like Lustre. The stream interpretation is useful for reasonning because it abstracts how a stream is implemented. The coiterative interpretation is useful to establish resource properties like execution in bounded time and verify equivalence properties when an inductive relation is given.*

2.1.2 Length Preserving Stream Functions. A stream function, say f, maps streams to streams, hence, it should be a value of type:

$$f : stream(T) \rightarrow stream(T')$$

or, on concrete streams:

$$f : coStr(T, S) \rightarrow coStr(T', S')$$

In the general case, a stream function f may need the *entire recipe* of its input stream and use it in an arbitrary manner, zero, one or several times. Quoting Caspi and Pouzet [23], "In many cases, this is unnecessarily complex and counter intuitive if f is intended to model a reactive system where data are computed progressively. Most functions of interest like `pre` and + are *synchronous* in the sense that their evaluation at a given step does not require knowing their argument's transition function but only the value yielded by the argument at that step." Such a function can be represented concretely as $CoP(ft, fs)$ made of a step function $ft : S \rightarrow T \rightarrow T' \times S$ and an initial state $fs : S$.

$$CoP : \forall S, T, T'.(S \rightarrow T \rightarrow T' \times S) \times S \rightarrow snode(T, T', S)$$

$snode(T, T', S)$ is the type of a one-input/one-output sequential sequential machine [45] from T to T' and with state in S. If $sm = CoP(ft, fs)$, it defines the stream function $\lambda x.apply(sm)(x)$ with:

$$apply(CoP(ft, fs))(CoF(x, xs)) = \\ CoF((\lambda(m, s).let\, v, xs' = x\, xs\, in\, let\, v', m' = ft\, m\, v\, in\, (v', (m', xs'))), (fs, xs))$$

with

$$apply(.)(.) : \forall T, T', S, S'.snode(T, T', S') \rightarrow (coStr(T, S) \rightarrow coStr(T', S' \times S))$$

The type signature highlights that if sm has a state $s_f : S'$ and x has a state $s_x : S$, the application, $apply(sm)(x)$ has a state which is a pair $(s_f, s_x) : S' \times S$. The application $apply(.)(.)$ converts a Mealy machine into a stream function. In order to produce the current value v' of the output, it only reads the current value v of the input. f is said to be length preserving or *synchronous* [23] when it is the image of a sequential machine.

Remark 2 (Non length preserving functions). *Of course, many stream functions of interest are not length preserving, that is, they use their argument recipe several times. E.g., a function* `half` *that returns the subsequence of even index, that is, given* $(x_n)_{n\in\mathbb{N}}$*,* $half(x)$ *returns* $(x_{2n})_{n\in\mathbb{N}}$ *or, on concrete streams:*

$$half\ (CoF(f, s)) = CoF((\lambda s.let\ \_, s' = f\ s\ in\ let\ v, s = f\ s'\ in\ (v, s)), s)$$

In order to produce one output, half makes two steps on its input argument $CoF(f, s)$*. An other example is* $next(.)$*, with* $next(x)(n) = x(n+1)$ *(operator R. of [37] or the so-called "tail" function on lists) whereas the unit delay pre . is length preserving (see Figure 1).*

Other classical non length preserving functions are the sampling and combination operators `when` *and* `merge` *of Lucid Synchrone and Scade, the operators* `when` *and* `current` *of Lustre, the* `when` *and* `default` *of Signal [7]. Non length-preserving functions like half and next can be expressed using those*

functions. If $0(1)$ *stands for the boolean sequence which is false at the first instant and then always true,* $next(x) = x$ when $0(1)$. *If* (10) *is the alternating boolean sequence,* $half(x) = x$ when (10).

In a language like Lustre and Scade, sampling operators are treated as if they were length preserving functions. Instantaneous values are complemented with an explicit absent value [21, 23]. Expecting a present value which is actually absent (or the converse) produces an error that is statically detected by the clock checking [21, 26].

In the sequel, we shall consider that all stream functions are length preserving. We have now all the elements to define the functional semantics for the synchronous data-flow kernel, including function application ($f(e_1, ..., e_n)$) and definition (`let node` $f\ p = e$). What remains is the treatment of fix-points to define the solution of mutually recursive stream equations.

2.2 The Difficulty with Mutually Recursive Stream Equations or Feedback loops

Consider a stream function:

$$f : stream(T) \longrightarrow stream(T)$$

or, on concrete streams:

$$f : coStr(T, S) \longrightarrow coStr(T, S)$$

and the following feedback loop written in the kernel language:

$$\texttt{let rec}\ y = f(y)\ \texttt{in}\ y.$$

We would like to define a fix-point operation $fix\,(.)$ where $fix\,(.) : (stream(T) \longrightarrow stream(T)) \longrightarrow stream(T)$ such that $fix\,(f)$ is a fix-point of f, that is, $fix\,(f) = f(fix\,(f))$. Its existence and uniqueness depends on f. E.g., consider the three following definitions for f:

```
let node f(x) = x    let node f(x) = x + 1    let node f(x) = 0 fby (x + 1)
```

The definition on the left stands for the function $\lambda x.x$ on streams; hence, there is an infinite number of solutions for the equation $y = f(y)$. If $f = \lambda x.(\lambda n.x(n) + 1)$, then no solution exist. If $f = \lambda x.\lambda n.if\ n = 0\ then\ 0\ else\ x(n-1) + 1$, which denotes the stream interpretation for the function defined on the right, the solution for y is the stream $\lambda n.n$ which is unique.

We would like to define $fix\,(.)$ as a total function and constructively, that is, with an algorithm that given f always terminate and returns $fix\,(f)$. To study its existence and define $fix\,(.)$ constructively, we explicitly complete a set of values T with a special value $\bot$ that denotes a deadlock (a computation that is stuck). Given a set T, let $D = T_\bot = \bot + T$, with $\bot$ a minimal element and $\leq$ the flat order, i.e., $\forall x \in D.\ \bot \leq x$. T is a pre-domain (all elements are incomparable) and $(D, \bot, \leq)$ is the trivial "flat" CPO. Let $stream(T_\bot)$ be the set of streams completed with $\bot$. The bottom element for streams $\bot_{st} = \lambda n.\bot$ (written $\bot$ whenever possible). The *prefix order* on streams $\leq_p$ is $x \leq_p y =_{df} \forall n \in \mathbb{N}.\ (x(n) \neq y(n)) \Rightarrow \forall m \geq n.(x(m) = \bot)$ [6]. $(stream(T_\bot), \leq_p, \bot_{st})$ is also a CPO. By the Kleene theorem, if $f : D \longrightarrow D$ is continuous on a CPO (that is, $f(sup_i(x_i)) = sup_i(f(x_i))$ for any chain $\{x_i \mid x_i \leq x_{i+1}, i \in \mathbb{N}\}$), $fix\,(f) = \lim_{n\to\infty} f^n(\bot)$ is the minimal fix-point of f. The stream operations *pre*, *fby*, *extend*$(.)(.)$ and `const`$(.)$ are indeed continuous; abstraction and application preserve continuity.

Are we done? Not yet. The difficulty is that the function $fix\,(.)$ is not an effective algorithm which computes the fix-point in bounded time for any function f. In finite time, only a prefix $f^n(\bot)$ of the solution v of $v = f(v)$ can be computed. Without special hypothesis on f, the number of iterations to obtain the k-th element of v is undecidable. Moreover, the computation is not incremental. Either it uses more and more memory to store the successive elements or it takes more and more time. If $f^n(\bot)$ gives the first k elements that are stored in memory, the computation of $f^{n+m}(\bot)$ only makes m new steps. If no value is stored, computing the $k+1$ element recomputes the first k elements.

Instead of considering the general class of stream functions, consider instead the particular case of a length preserving function f that is, it exists $sm = CoP(ft, s_0)$ such that $f\ x = apply(sm)(x)$. The concrete stream y such that $v_n = nth(y)(n)$ for all $n \in \mathbb{N}$ and $v = f(v)$ should verify:

$$v_n, s_{n+1} = ft\ s_n\ v_n$$

Given an initial state $s : S$, we look for a value $feedback\,(ft)$ that is a solution of:

$$X(s) = let\ v, s' = X(s)\ in\ ft\ s\ v$$

A lazy functional language like Haskell [5] allows for writing such a recursively defined value. One can program in Haskell a function $feedback\,(.) : \forall S, V.(S \to V_\perp \to V_\perp \times S) \to (S \to V_\perp \times S)$:

$$feedback\,(ft) = \lambda s.let\ rec\ v, s' = ft\ s\ v\ in\ (v, s')$$

where v is defined recursively. $nth(CoF(feedback\,(ft), s))(.)$ is the stream solution of the equation $v = f(v)$ and is minimal. Quoting Caspi and Pouzet [23], "We have replaced a recursion on time, that is, a stream recursion, by a recursion on the instant." While a shallow embedding of the above primitives in a Haskell (lifting, application, abstraction, fix-point) gives an interpreter (and this is already a useful result), it not entirely satisfactory. By considering only length preserving functions, we have removed the problem of unbounded buffers, but the problem of causality remains. The function $feedback\,(.)$ may not terminate for some argument ft, and we do not know and for what reason it does not terminate. For example, $feedback\,(ft)$ is not defined (diverges) when $ft\ s\ x = (x, s)$, $ft\ s\ x = (x + 1, s)$ or $ft\ s\ (x, y) = ((y, x), s)$ which correspond respectively to:

`let rec` x `=` x `in` x `let rec` x `=` $x + 1$ `in` x `let rec` x `=` y `and` y `=` x `in` x, y

On the contrary, $feedback\,(ft)$ is defined when $ft\ s\ x = (1{+}s, (x{+}2))$ or $ft\ s\ (x, y) = (1{+}s, (y{+}2, x{+}3))$ which correspond to the following recursive equations: [6]

`let rec` x `=` $1 +$ `(`0 `fby` $(x + 2)$`)` `in` x `let rec` x `=` $1 +$ `(`0 `fby` $y + 2$`)` `and` y `=` $x + 3$ `in` x, y

Because the function $feedback\,(.)$ may not terminate, it cannot be defined as a function in the language of a proof assistant like Coq [7] where all functions must terminate, unless using a trick, like a *fuel* argument. By restricting ouselves to length preserving functions and their composition, the problem is simpler. The CPO of streams, which is of unbounded height, is replaced by a flat CPO of bounded height; and continuity is replaced by monotony.

Bounded Fix-point. If D is a CPO whose height is bounded by $n \in \mathbb{N}$ (chains shorter than n), and f is monotone ($\forall x, y \in D.\ x \leq y \Rightarrow f(x) \leq f(y)$), the fix-point can be replaced by a bounded one.

$$\begin{array}{lcl} feedback\,(0)\,(f)(s) & = & \perp, s \\ feedback\,(n)\,(f)(s) & = & let\ v, s' = feedback\,(n-1)\,(f)(s)\ in\ f\ s\ v \end{array}$$

with: $feedback\,(.) : \forall S, D.\mathbb{N} \to (S \to D \to D \times S) \to S \to D \times S$ or a tail-recursive form $feedback\,(n)\,(f)(=)(s)(\perp)$ that stops as soon as the fix-point is reached:

$$\begin{array}{l} feedback\,(0)\,(f)(=)(v)(s) = v, s \\ feedback\,(n)\,(f)(=)(v)(s) = let\ v', s' = f\ s\ v\ in\ if\ v = v'\ then\ v, s'\ else\ feedback\,(n-1)\,(f)(=)(v')(s) \end{array}$$

with: $feedback\,(.) : \forall S, D.\ \mathbb{N} \to (S \to D \to D \times S) \to (D \to D \to bool) \to D \to S \to D \times S$
Hence, if the concrete semantics of f is $CoP(ft, fs)$, the concrete semantics of `let rec` $y = f(y)$ `in` y is $CoF((\lambda s.feedback\,(n)\,(ft)(=)(\perp)(s)), fs)$. How many iterations are sufficient to get a fix-point? If $D = T_\perp$ with T a pre-domain (all elements are incomparable):

[5] https://www.haskell.org

[6] The latter defines the sequences $(x_n)_{n\in\mathbb{N}}$ and $(y_n)_{n\in\mathbb{N}}$ with $y_n = x_n + 3$ and $x_n = 1 + (if\ n = 0\ then\ 0\ else\ y_{n-1} + 2)$.

[7] https://coq.inria.fr

(1) Either the first element v' of the pair $ft\,v\,s$ depends on v, that is, $v' = \bot$ whenever $v = \bot$. The program contains a *causality loop*. In a lazy functional language like Haskell, this would correspond to an unbounded recursion when evaluating the recursive definition $let\ rec\ v, s' = ft\,s\,v\ in\ (v, s')$.
(2) or it does not, that is, $\bot < v'$.

At most 1 iteration is necessary to get the fix-point. If D is a richer domain, e.g., a Cartesian or smashed product, a function with a bounded domain, the number of iterations n is bounded by $\overline{D}$.

$$\overline{T_\bot} \quad = \quad 1 \qquad \overline{D_1 \times ... \times D_k} \quad = \quad \overline{D_1} + ... + \overline{D_k} \qquad \overline{D_1 \to D_2} \quad = \quad \overline{D_2}^{|D_1|}$$

where $|D|$ is the cardinality of D. In the case of a smashed product, the height is only $\overline{D_1} + ... + \overline{D_k} - k$. When the domain of a function is finite (e.g., Boolean), the fix-point is reached in bounded time. [8]

The above definition of *feedback* (.) can be used to compute the least fix-point of a set of mutually recursive equations E defining k variables. Intuitively, the current output defined by a set of mutually recursive stream definitions, e.g.,: `let rec` $x_1 = e_1$ `and` ... `and` $x_n = e_n$ `in` e is a function which associates the current value of e_i to x_i. Let $Def(E) = \{x_1, ..., x_k\}$ be the finite set of names defined by an equation E from the kernel language. Let $[v_1/x_1, ..., v_k/x_k] : env(D)$ be a valuation (a function from names to values in D).

$$env(D) = names \to D$$

The semantics for E is a function $f : S \to env(D) \to env(D) \times S$ which, given a state $s : S$ and environment $\rho : env(D)$, returns a new environment $\rho' : env(D)$ and new state $s' : S$. If the minimal elements for environments $[\bot/x_1, ..., \bot/x_k]$ (that we simply write $\bot$) and the equality $(=)$ is lifted to environments pairwise, and $\overline{D} = h$, then $feedback\,(k \times h)\,(f)(=)(s)(\bot)$ computes the least solution of E in at most $k \times h$ steps. In the particular case were $D = T_\bot$, that is, the value of a variable is either $\bot$ or known entirely, $h = 1$, that is, at most k iterations are necessary. This gives a simple, correct and complete, purely syntactic argument, to bound the number of iterations. Otherwise, the number of steps would need the type information, hence be done after static typing.

Remark 3. *The signature of feedback* (.) *is not of the form* $(D \to D) \to D$ *as one would expect for a fix-point. An alternative and equivalent definition can be obtained by splitting* $ft : S \to D \to D \times S$ *into an* output *function* $fo : S \to D \to D$ *and* update *function* $fu : S \to D \to S$ *such that:*

$$v_n = fo\,s_n\,v_n \qquad s_{n+1} = fu\,s_n\,v_n$$

that is $fo = fst \circ ft$, $fu = snd \circ ft$ *and* $v_n = fix\,(fo\,s_n)$ *where* fix (.) $: (D \to D) \to D$, *fst and snd are the left and right projections of a pair. What is presented in the sequel could easily be adapted accordingly. We keep the initial one because it is closer to the actual code produced by a synchronous compiler: the step function returns the current output and a new state. In the generated code, this state is modified in-place.*

2.3 Semantics

We are ready to define the semantics of the language kernel. [9] The semantics is untyped; values are defined inductively:

$$v^\star \quad ::= \quad a \mid nil \mid \bot \mid (v^\star, v^\star) \qquad\qquad a \quad ::= \quad () \mid b \mid i \mid (a, a) \mid op_n$$

[8] Having $D = D_1 \to D_2$ a functional domain would mean that equations define streams of functions; this is forbidden by static typing in Scade. Nonetheless, this could be allowed for finite maps (with finite domain) like arrays.
[9] In the sequel, to make the notation lighter, we will confuse sets and types and abusively write $T(A, B) =_{df} A + B$ both as a set and the definition of a type with two injective functions $inl : A \to T(A, B)$ and $inr : B \to T(A, B)$.

An extended value $v^\star$ can be a regular (atomic) value, *nil* or $\bot$ or a pair $(v_1^\star, v_2^\star)$. A value a can be void ($()$), a Boolean (b), an integer number (i), a pair (a_1, a_2) or an operator op_n with arity $n \in \mathbb{N}$. op_n denotes an element in $V^n \to V$ ($V^0 = unit$ and $V^n = V \times V^{n-1}$). We write $V^\star$ for the set of extended values and V for the set of values. The order between extended values is the flat order with $\bot$ the least element, all others being incomparable. For pairs, it is the pair-wise order. We write $(x_1, ..., x_n)$ as a short-cut for $x_1, (x_2, ..., x_n)$ and $V_1^\star \times \cdots \times V_n^\star$ as a short-cut for $V_1^\star \times (V_2^\star \times \cdots \times V_n^\star)$. A value $v^\star$ is atomic if it does not contain any bottom or nil value, i.e., it is an element ($a \in V$).

States are also defined inductively. We write S for the set of states.

$$s \quad ::= \quad () \mid v^\star \mid (s, s) \mid None \mid Some(v^\star) \mid \bot \mid nil$$

A global environment γ associates a value to a name.

$$genv\,(V, S) =_{df} names \to Global(V, S)$$

A global value gv can be either a value v, a combinational function or a process $CoP(p, s)$.

$$Global(V, S) =_{df} V + (V^\star \to V^\star) + snode(V^\star, V^\star, S)$$

and a local environment for streams $env\,(V) =_{df} names \to V^\star$. We use the following notation for environments. $[]$ denotes the empty environment, hence, $[](x) = \bot$. If $\rho = \rho' + [v/x]$, $\rho(x) = v$ and $\rho(y) = \rho'(y)$ if $x \neq y$. Finally, $\rho \backslash x = \rho'$.

Lifting $\bot$ and nil to environments and functions. We overload the notation $\bot$ and *nil* for environments. If $N = \{x_1, ..., x_n\}$ is a set of names:

$$\bot_N \quad = \quad [\bot/x_1, ..., \bot/x_n] \qquad nil_N \quad = \quad [nil/x_1, ..., nil/x_n]$$

If s is a state, $\bot_s$ (resp. nil_s) distributes $\bot$ (resp. *nil*) according to s. The same applies to values:

$$\bot(()) \quad = \quad \bot \qquad \bot((s_1, ..., s_n)) \quad = \quad \bot(s_1), ..., \bot(s_n) \qquad \bot(a) \quad = \quad \bot$$

Lifting: If $a : V$ is a constant, ${}^\star a : V^\star$. If $op_n : V^n \to V$, ${}^\star op_n$ lifts it. Definitions below are taken in order (left then right). ${}^\star op_n$ is strict w.r.t $\bot$ and *nil*. $\bot$ takes precedence over *nil*.

$$\begin{array}{lcllcl} {}^\star a & = & a & {}^\star op_n(..., nil, ...) & = & nil \\ {}^\star op_n(..., \bot, ...) & = & \bot & {}^\star op_n(v_1, ..., v_n) & = & op_n(v_1, ..., v_n) \end{array}$$

We also lift the conditional such that (definition taken in order, left then right):

$$\begin{array}{lcllcl} {}^\star if\ \bot\ then\ \_\ else\ \_ & = & \bot & {}^\star if\ true\ then\ x\ else\ \_ & = & x \\ {}^\star if\ nil\ then\ \_\ else\ \_ & = & nil & {}^\star if\ false\ then\ \_\ else\ x & = & x \\ & & & {}^\star if\ \_\ then\ \_\ else\ \_ & = & \bot \text{ otherwise} \end{array}$$

The Semantics of Expressions. The semantics of an expression e is defined in Figure 2. It is defined using two auxiliary functions. If $\rho : env\,(V)$ and $\gamma : genv\,(V, S)$ are local and global environments respectively, $[\![e]\!]^{Init}_{\rho,\gamma} : S$ denotes the initial state and $[\![e]\!]^{Step}_{\rho,\gamma} : S \to V^\star \times S$ the transition function.

$$[\![e]\!]_{\rho,\gamma} = CoF(f, s) \text{ where } f = [\![e]\!]^{Step}_{\rho,\gamma} \text{ and } s = [\![e]\!]^{Init}_{\rho,\gamma}$$

To make the notation lighter, γ is left implicit whenever possible.

Consider first the semantics of a unit delay c `fby` e initialized with a constant c given in Figure 2. The initial state is of the form (c, s) if s is the initial state for e. The transition function takes a state which is a pair (m, s); it returns m and a new state $[\![e]\!]^{Step}_{\rho}(s)$. The uninitialized delay `pre` e follows

$$
\begin{array}{lcl}
[\![c \texttt{ fby } e]\!]_\rho^{Init} & = & (c, [\![e]\!]_\rho^{Init}) \\
[\![c \texttt{ fby } e]\!]_\rho^{Step} & = & \lambda(m,s).(m, [\![e]\!]_\rho^{Step}(s)) \\
[\![\texttt{pre } e]\!]_\rho^{Init} & = & (nil, [\![e]\!]_\rho^{Init}) \\
[\![\texttt{pre } e]\!]_\rho^{Step} & = & \lambda(m,s).(m, [\![e]\!]_\rho^{Step}(s)) \\
[\![x]\!]_\rho^{Init} & = & () \\
[\![x]\!]_\rho^{Step} & = & \lambda s.(\rho(x), s) \\
[\![c]\!]_\rho^{Init} & = & () \\
[\![c]\!]_\rho^{Step} & = & \lambda s.(c, s) \\
[\![(e_1, e_2)]\!]_\rho^{Init} & = & ([\![e_1]\!]_\rho^{Init}, [\![e_2]\!]_\rho^{Init}) \\
[\![(e_1, e_2)]\!]_\rho^{Step} & = & \lambda(s_1, s_2).\mathit{let}\, v_1, s_1' = [\![e_1]\!]_\rho^{Step}(s_1)\ \mathit{in} \\
 & & \quad \mathit{let}\, v_2, s_2' = [\![e_2]\!]_\rho^{Step}(s_2)\ \mathit{in} \\
 & & \quad ((v_1, v_2), (s_1', s_2'))
\end{array}
$$

$$
\begin{array}{lcl}
[\![f\,(e)]\!]_\rho^{Init} & = & [\![e]\!]_\rho^{Init} \\
[\![f\,(e)]\!]_{\rho,\gamma}^{Step} & = & \lambda s.\mathit{let}\, v, s' = [\![e]\!]_\rho^{Step}(s)\ \mathit{in} \\
 & & \quad ((^\star\gamma(f)\, v), s') \\
[\![f\,(e)]\!]_\rho^{Init} & = & (fs, [\![e]\!]_\rho^{Init}) \\
 & & \text{where } \gamma(f) = CoP(ft, fs) \\
[\![f\,(e)]\!]_{\rho,\gamma}^{Step} & = & \lambda(m,s).\mathit{let}\, v, s' = [\![e]\!]_\rho^{Step}(s)\ \mathit{in} \\
 & & \quad \mathit{let}\, r, m' = ft\, m\, v\ \mathit{in} \\
 & & \quad (r, (m', s')) \\
 & & \text{where } \gamma(f) = CoP(ft, fs)
\end{array}
$$

$$
\begin{array}{lcl}
[\![\texttt{let node } f\ x = e]\!]_\rho & = & \rho + [CoP(ft, fs)/f] \\
 & & \text{where } fs = [\![e]\!]_{\rho+[\bot/x]}^{Init} \\
 & & \text{and } ft = \lambda s, v.[\![e]\!]_{\rho+[v/x]}^{Step}(s) \\
[\![\texttt{let } f = e]\!]_\rho & = & \rho + [v/f] \text{ where } v = [\![e]\!]_\rho
\end{array}
$$

Fig. 2. The Semantics of Expressions

the same principle but for the initial value of the state, it is set to *nil*. When the two arguments of a delay are streams, that is, e_1 `fby` e_2, the encoding can be:

$$
\begin{array}{lcl}
[\![e_1 \texttt{ fby } e_2]\!]_\rho^{Init} & = & (None, [\![e_1]\!]_\rho^{Init}, [\![e_2]\!]_\rho^{Init}) \\
[\![e_1 \texttt{ fby } e_2]\!]_\rho^{Step} & = & \lambda(m, s_1, s_2).\mathit{let}\, v_1, s_1' = [\![e_1]\!]_\rho^{Step}(s_1)\ \mathit{in} \\
 & & \quad \mathit{let}\, v_2, s_2' = [\![e_2]\!]_\rho^{Step}(s_2)\ \mathit{in}\, (m\ \mathit{init}\ v_1, (Some(v_2), s_1', s_2'))
\end{array}
$$

with: $m\ \mathit{init}\ v \quad = \quad \mathit{match}\ m\ \mathit{with}\ None \rightarrow v \mid Some(v) \rightarrow v$

Note that in the definition above, $\mathit{let}\, v_1, s_1' = [\![e_1]\!]_\rho^{Step} s_1\ \mathit{in} \ldots$ is a notation at the meta (semantics) level. It defines a value for the pair (v_1, s_1') which equals, by definition, that of $[\![e_1]\!]_\rho^{Step} s_1$.

The initial state $[\![x]\!]_\rho^{Init}$ for a variable x is `()` while the step function is $\lambda s.(\rho(x), s)$. The initial state for a constant c is also `()` and the step function is simply $\lambda s.(c, s)$. For a pair (e_1, e_2), the initial state is the pair of initial states for e_1 and e_2; the step function steps into e_1 and e_2. The semantics for an application $f(e)$ has two cases. When f is a combinational function, we simply apply the value of f to the current input. When f is a sequential machine $CoP(ft, fs)$, we follow the definition of the synchronous application $apply(.)(.)$: the initial state is made of the initial state of f and the initial state of the argument. The step function applies the step function ft to its current state and input. Finally, the semantics of a node definition is a value $CoP(ft, fs)$.

Activation Conditions and Reset. The semantics of the two forms of conditionals of the kernel language are given in Figure 3. The `if . then . else .` executes both branches whereas `when . then . else .` executes one branch according to the value of the condition. The later one is called the "activation condition" in Scade [27]. The two constructs differ when their body contain stateful computations. E.g.:

```
let node from () returns (nat)
  nat = 0 fby (nat + 1)

let node f() =
  let rec half = true fby not half in
  if half then (from 0) else 0
```

```
let node g() =
  let rec half = true fby not half in
  when half then (from 0) else 0
```

While f produces the sequence 0 0 2 0 4 0 6..., g produces the sequence 0 0 1 0 2 0 3 0 4....

The reset operator `reset` e_1 `every` e_2 executes the body e_1 at every step but restarts from the initial state whenever e_2 is true [23].

$$
\begin{array}{lcl}
[\![\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2]\!]_\rho^{Init} & = & ([\![e]\!]_\rho^{Init}, [\![e_1]\!]_\rho^{Init}, [\![e_2]\!]_\rho^{Init}) \\
[\![\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2]\!]_\rho^{Step} & = & \lambda(s, s_1, s_2).\, let\ v, s = [\![e]\!]_\rho^{Step}(s)\ in\ let\ v_1, s_1' = [\![e_1]\!]_\rho^{Step}(s_1)\ in \\
 & & \qquad let\ v_2, s_2' = [\![e_2]\!]_\rho^{Step}(s_2)\ in \\
 & & \qquad ((^\star if\ v\ then\ v_1\ else\ v_2), (s, s_1', s_2')) \\
[\![\texttt{when } e \texttt{ then } e_1 \texttt{ else } e_2]\!]_\rho^{Init} & = & ([\![e]\!]_\rho^{Init}, [\![e_1]\!]_\rho^{Init}, [\![e_2]\!]_\rho^{Init}) \\
[\![\texttt{when } e \texttt{ then } e_1 \texttt{ else } e_2]\!]_\rho^{Step} & = & \lambda(s, s_1, s_2).\, let\ v, s = [\![e]\!]_\rho^{Step}(s)\ in \\
 & & \qquad ^\star if\ v\ then\ let\ v_1, s_1' = [\![e_1]\!]_\rho^{Step}(s_1)\ in\ (v_1, (s, s_1', s_2)) \\
 & & \qquad\ else\ let\ v_2, s_2' = [\![e_2]\!]_\rho^{Step}(s_2)\ in\ (v_2, (s, s_1, s_2')) \\
[\![\texttt{reset } e_1 \texttt{ every } e_2]\!]_\rho^{Init} & = & ([\![e_1]\!]_\rho^{Init}, [\![e_2]\!]_\rho^{Init}) \\
[\![\texttt{reset } e_1 \texttt{ every } e_2]\!]_\rho^{Step} & = & \lambda(s_1, s_2).\, let\ v_2, s_2' = [\![e_2]\!]_\rho^{Step}(s_2)\ in \\
 & & \qquad let\ s_1' = ^\star if\ v_2\ then\ [\![e_1]\!]_\rho^{Init}\ else\ s_1\ in \\
 & & \qquad let\ v_1, s_1 = [\![e_1]\!]_\rho^{Step}(s_1')\ in\ (v_1, (s_1, s_2'))
\end{array}
$$

Fig. 3. Activation Conditions and the Reset

Equations and Local Definitions. The final two constructs to consider are the local definitions `let` E `in` e and `let rec` E `in` e where E is a set of equations. For that, we need auxiliary definitions. If E is an equation, ρ is an environment, $[\![E]\!]_\rho^{Init}$ is the initial state and $[\![E]\!]_\rho^{Step}$ is the step function. The semantics of an equation eq is:

$$[\![E]\!]_\rho = [\![E]\!]_\rho^{Init}, [\![E]\!]_\rho^{Step}$$

If p is a pattern and v is a value, $[v|p]$ builds the environment by matching v by p such that:

$$
\begin{array}{lcl}
[v|x] & = & [v/x] \\
[(v_1, v_2)|(p_1, p_2)] & = & [v_1|p_1] + [v_2|p_2] \\
[v|p] & = & \bot \text{ otherwise}
\end{array}
$$

$+$ is the union of two environments provided their domains do not intersect (otherwise, it returns $\bot$). [10] We also need two auxiliary functions. $Def(E)$ returns the set of names defined by eq. $\|E\|$ is the number of elements of $Def(E)$.

$$
\begin{array}{lcllcl}
Def(p = e) & = & Vars(p) & Vars(()) & = & \emptyset \\
Def(E_1 \texttt{ and } E_2) & = & Def(E_1) \cup Def(E_2) & Vars((x_1, ..., x_n)) & = & \{x_1, ..., x_n\}
\end{array}
$$

The semantics of equations and local definitions is given in Figure 4. The initial state of an equation $p = e$ is that of e. The step function returns an environment which associates a value to every variable from pattern p. The initial state of a parallel composition E_1 `and` E_2 is the pair of initial states of E_1 and E_2. The step function returns an environment which compose that of E_1 and that of eq_2. For a recursive definition `let rec` E `in` e, a bounded fix-point is computed. The number of iteration is at most $n + 1$ if n is the number of variables defined by E. The semantics for local (possibly recursive) declarations consists in computing at every step, the current values defined by eq then the expression e.

As an example, `count(z, r)` counts the number of instants `z` is true between two occurrences of `r`. `periodic()` returns a periodic Boolean signal with period 42.

[10] In the actual implementation, it is encoded as an error.

$$
\begin{array}{lcl}
[\![p = e]\!]^{Init}_{\rho} & = & [\![e]\!]^{Init}_{\rho} \\
[\![p = e]\!]^{Step}_{\rho} & = & \lambda s.let\ v, s' = [\![e]\!]^{Step}_{\rho}(s)\ in\ ([v|p], s') \\
[\![E_1\ \texttt{and}\ E_2]\!]^{Init}_{\rho} & = & ([\![E_1]\!]^{Init}_{\rho}, [\![E_2]\!]^{Init}_{\rho}) \\
[\![E_1\ \texttt{and}\ E_2]\!]^{Step}_{\rho} & = & \lambda(s_1, s_2).let\ \rho_1, s_1' = [\![E_1]\!]^{Step}_{\rho}(s_1)\ in\ let\ \rho_2, s_2' = [\![E_2]\!]^{Step}_{\rho}(s_2)\ in\ (\rho_1 + \rho_2, (s_1', s_2')) \\
[\![E]\!]^{rec}_{\rho} & = & \lambda s.feedback\ (\|E\| + 1)\ (\lambda s, \rho'.[\![E]\!]^{Step}_{\rho+\rho'}(s))(s) \\
[\![\texttt{let}\ E\ \texttt{in}\ e]\!]^{Init}_{\rho} & = & ([\![E]\!]^{Init}_{\rho}, [\![e]\!]^{Init}_{\rho+[\bot_N/N]}) \text{ with } N = Def(E) \\
[\![\texttt{let}\ E\ \texttt{in}\ e]\!]^{Step}_{\rho} & = & \lambda(s_1, s_2).let\ \rho', s_1' = [\![E]\!]^{Step}_{\rho}(s_1)\ in\ let\ v', s_2' = [\![e']\!]^{Step}_{\rho+\rho'}(s_2)\ in\ (v', (s_1', s_2')) \\
[\![\texttt{let rec}\ E\ \texttt{in}\ e]\!]^{Init}_{\rho} & = & ([\![E]\!]^{Init}_{\rho}, [\![e]\!]^{Init}_{\rho+[\bot_N/N]}) \text{ with } N = Def(E) \\
[\![\texttt{let rec}\ E\ \texttt{in}\ e]\!]^{Step}_{\rho} & = & \lambda(s_1, s_2).let\ \rho', s_1' = [\![E]\!]^{rec}_{\rho}(s_1)\ in\ let\ v', s_2 = [\![e]\!]^{Step}_{\rho+\rho'}(s_2)\ in\ (v', (s_1', s_2'))
\end{array}
$$

Fig. 4. Equations and Local Definitions

```
let node count(z, r) =
  reset
    let rec o = if z then 1 else 0
    and cpt = o -> pre cpt + o in o
  every r
```

```
let node periodic() returns (ok)
    v = count(true, false fby (v = 41))
  and ok = (v = 1)
let node deadlock() returns ok
  v = count(true, v = 41) and ok = (v = 1)
```

If the initialized delay . fby . in the definition of `periodic` is removed, we get a deadlock, that is, the value for `v` and `ok` is bottom. Indeed, starting with the initial environment $[\bot/v, \bot/ok]$, the fix-point is reached in one iteration because the step function of `count(z, r)` returns $\bot$.

2.4 Dealing with Errors

The type of concrete streams $coStr(V^\star, S)$ is sufficient to define the semantics of an expression. All errors, type or run-time errors are all represented by the value $\bot$ (cf. [50], page 235). Yet, this solution is not satisfactory because $\bot$ is intended to model an expression that deadlocks. A better one is to distinguish the different errors with different values. Several solutions are possible. The set of extended values $V^\star$ can be complemented with two values, *tyerr* and *err* [50]; the first being returned when the actual entry of a function is not in its domain; the second is any kind of run-time error (e.g., division by zero). Taking v^+ instead of $v^\star$ forces all errors to propagate to the top.

$$v^\star \quad ::= \quad \ldots \mid tyerr \mid err \qquad\qquad v^+ \quad ::= \quad v^\star \mid tyerr \mid err$$

The set of values $V^\star$ is extended with $TyErr$ and Err with $tyerr \in TyErr$ and $err \in Err$. Errors and values are incomparable with each others and $\bot$ stay the minimal element. The lifting function * is extended to propagate those errors (cf. [50], page 235). The definition for concrete streams and length preserving function stay the same. With this representation, it is possible to have a stream process that produces a value containing an error at some instant $n \in \mathbb{N}$ but this error is discarded because it is not used to compute the result. E.g., `let` $x = 1/$`true` `in` 0.

An alternative that is followed in the implementation associated to the present material is to force the execution to stop whenever an error occurs. If $Result(V, E) = V + E$ and $E = TyErr + Err$, a concrete stream becomes:

$$CoF : \forall S, V, E.(S \longrightarrow Result(V^\star \times S, E)) \times S \longrightarrow coStrEr(V^\star, S, E)$$

and a sequential machine:

$$CoP : \forall S, V, E.(S \longrightarrow V^\star \longrightarrow Result(V^\star \times S, E)) \times S \longrightarrow sNodeError(V^\star, V^\star, S, E)$$

When an error e occurs, the transition function does not returns a value and a new state but stops with an error value. In order to propagate errors, we use a simple result monad (https://v2.ocaml.org/api/Result.html) with a data-type defined with the two constructors $Ok(.) : V \longrightarrow Result(V, E)$ and $Error(.) : E \longrightarrow Result(V, E)$:

$$let\ bind\ e\ f = match\ e\ with\ Ok(v) \longrightarrow f\ v \mid Error(\_) \longrightarrow e \qquad \begin{array}{l} let\ return\ v = Ok(v) \\ let\ error\ v = Error(v) \end{array}$$

Defining $let^\star\ x = e_1\ in\ e_2$ as a shortcut for $bind(e_1)(\lambda x.e_2)$, the definitions of the semantics function $[\![.]\!]$: needs only small changes. Returning a result (v, s) is now written $return(v, s)$. A local definition $let\ v, s' = [\![e]\!]_\rho^{Step}(s)\ in \ldots$ is now written $let^\star\ v, s' = [\![e]\!]_\rho^{Step}(s)\ in \ldots$ [11]

3 LANGUAGE EXTENSIONS

We now consider two main extensions of the language kernel. The first extends the set of equations with local declarations and a by-case definition for streams with an implicit rule for missing equations; the second extends them with hierarchical automata, a major increase in expressiveness based on [24, 25].

3.1 Local Blocks with Last/Default Values and By-case Definitions of Streams

Equations are extended with two constructs. Local declarations of a variable and a pattern-matching construction which allows for defining streams by case.

$$\begin{array}{lcl} E & ::= & \ldots \mid () \mid \texttt{local}\ loc\ \texttt{in}\ E \mid \texttt{match}\ e\ \texttt{with}\ pat \longrightarrow E \mid \ldots \mid pat \longrightarrow E\ \texttt{end} \\ pat & ::= & C^n(x, ..., x) \mid C^0 \\ loc & ::= & x \mid x\ \texttt{init}\ e \mid x\ \texttt{default}\ e \end{array}$$

Expressions are extended with a construct to access the last value of a stream:

$$e \quad ::= \quad \ldots \mid \texttt{last}\ x$$

() is the empty equation; it defines no variable. The construct $\texttt{local}\ loc\ \texttt{in}\ E$ declares a variable to be local in E. loc can be a simple variable (x); it can have a default value ($x\ \texttt{default}\ e$); its previous value $\texttt{last}\ x$ can be initialized ($x\ \texttt{init}\ e$). pat define a pattern. It is either a constructor C^0 (arity is zero) or of the form $C^n(x_1, ..., x_n)$ (arity is n). An equation can be a pattern matching construct $\texttt{match}\ e\ \texttt{with}\ pat_1 \longrightarrow E_1 \ldots pat_n \longrightarrow E_n\ \texttt{end}$. The active set of equations is E_i if e_i evaluates to a value v and pat_i is the first pattern that matches v.

The set of values v is extended with constructors. The definition of environments is adapted to associate a default or initial value to a variable.

$$a ::= \ldots \mid C^0 \mid C^n(a, ..., a) \qquad \rho ::= \rho + [v^\star/x] \mid \rho + [v^\star/default\ x] \mid [v^\star/last\ x] \mid []$$

To deal with incomplete definitions of a stream, ρ by ρ' is the completion of ρ with default or initial values from ρ'. The function will be used such that defined names in ρ are not defined in ρ'.

$$\begin{array}{lcllcl} \rho\ \text{by}\ [] & = & \rho & \rho\ \text{by}\ (\rho' + [v/last\ x]) & = & (\rho + [v/x])\ \text{by}\ \rho' \\ \rho\ \text{by}\ (\rho' + [v/default\ x]) & = & (\rho + [v/x])\ \text{by}\ \rho' & \rho\ \text{by}\ (\rho' + [v/x]) & = & \rho\ \text{by}\ \rho' \end{array}$$

[11] The $let^\star$ notation is borrowed from OCaml (https://v2.ocaml.org/manual/bindingops.html). It corresponds to the "do" notation of Haskell and Coq.

The Last Computed Value. The language of expressions is extended with the construction `last` x. If x is a local variable, `last` x contains the previous defined value of x in its scope:

$$[\![\texttt{last}\, x]\!]_\rho^{Init} = () \qquad [\![\texttt{last}\, x]\!]_\rho^{Step} = \lambda s.(\rho(last\, x), s)$$

The initial state associated to `last` x is empty (that is, `()`). The step function returns the current value of `last` x in ρ at every instant. To illustrate the by-case definition of streams, consider the following example on the left:

```
let node count(u) returns (i, d)
   li = 0 fby i and ld = 0 fby d
  and match x with
  | Incr -> do i = li + 1 and d = ld done
  | Zero -> do i = 0 and d = ld done
  | Decr -> do d = ld + 1 and i = li done
 end
```

```
let node count(u)
   returns (i init 0, d init 0)
  match x with
  | Incr -> do i = last i + 1 done
  | Zero -> do i = 0 done
  | Decr -> do d = last d + 1 done
  end
```

Given input `u`, `count(u)` produces two output streams `i` and `d`. In the version on the left, a definition for all stream is given in each branch. This makes the code overly verbose. It is equivalent to the program on the right: when a definition is missing in a branch of a conditional for a variables that is declared as a state variable (e.g., `i` in branch `Incr`), its value is implicitly kept (that is, `i = last i`). The declaration `i init 0` states that `last i` is initialized with value `0`. Blocks with last and default values allow for writting less verbose code. For example, they allow to define pure signals of Esterel: a pure signal is a Boolean stream whose default value is *false*. Taking the convention that the conditional over equations `if` e `then` E_1 `else` E_2 is a shortcut for `match` e `with true` $\rightarrow E_1$ `| false` $\rightarrow E_2$ `end` and `if` e `then` E, a shortcut for `match` e `with true` $\rightarrow E$ `| false` $\rightarrow$ () `end`, the P13 example of Esterel is written on the right.

```
(* Esterel primer v5.91, Berry *)
module P13:
  input I;
  output O1, O2;
  present I then
    present O2 then emit O1 end
  else
    present O1 then emit O2 end
  end present
end module
```

```
let node p13(i) returns
  (o1 default false, o2 default false)
  if i then
    if o2 then o1 = true
  else
    if o1 then o2 = true

let node p13_verbose(i) returns (o1, o2)
  if i then
    do if o2 then o1 = true else o1 = false
    and o2 = false done
  else
    do if o1 then o2 = true else o2 = false
    and o1 = false done
```

The data-flow version for `p13` (and its verbose version `p13_verbose`) is a perfectly valid Scade definition. In particular, it passes all the static checks, including the type-based causality implemented by the compiler.

The semantics for equations with local variables and control structures is given in Figure 5. For the declaration `local` x `in` E of a local variable x in an equation E, the semantics is that of E but removing the entry x from the resulting environment ρ. For the declaration `local` x `default` e `in` E of a local variable with a default value, the semantics is that of a local declaration except that the environment is extended with a default value for x. For the declaration `local` x `init` e `in` E

$$
\begin{array}{lcl}
[\![()]\!]_\rho^{Init} & = & () \\
[\![()]\!]_\rho^{Step} & = & \lambda s.([], s) \\
[\![\texttt{local}\ x\ \texttt{in}\ E]\!]_\rho^{Init} & = & [\![E]\!]_\rho^{Init} \\
[\![\texttt{local}\ x\ \texttt{in}\ E]\!]_\rho^{Step} & = & \lambda s.let\ \rho', s' = [\![E]\!]_\rho^{rec}(s)\ in\ (\rho'\backslash x, s') \\
\\
[\![\texttt{local}\ x\ \texttt{default}\ e\ \texttt{in}\ E]\!]_\rho^{Init} & = & ([\![e]\!]_\rho^{Init}, [\![E]\!]_\rho^{Init}) \\
[\![\texttt{local}\ x\ \texttt{default}\ e\ \texttt{in}\ E]\!]_\rho^{Step} & = & \lambda(s_1, s_2).let\ v, s_1' = [\![e]\!]_\rho^{Step}(s_1)\ in \\
 & & \quad let\ \rho', s_2' = [\![E]\!]_{\rho+[v/default\ x]}^{rec}(s_2)\ in\ (\rho'\backslash x, (s_1', s_2')) \\
\\
[\![\texttt{local}\ x\ \texttt{init}\ e\ \texttt{in}\ E]\!]_\rho^{Init} & = & (None, [\![e]\!]_\rho^{Init}, [\![E]\!]_\rho^{Init}) \\
[\![\texttt{local}\ x\ \texttt{init}\ e\ \texttt{in}\ E]\!]_\rho^{Step} & = & \lambda(m, s_1, s_2).let\ v, s_1' = [\![e]\!]_\rho^{Step}(s_1)\ in \\
 & & \quad let\ \rho', s_2' = [\![E]\!]_{\rho+[m\ init\ v/last\ x]}^{rec}(s_2)\ in \\
 & & \quad (\rho'\backslash x, (\rho'(x), s_1', s_2'))
\end{array}
$$

$$
[\![\texttt{match}\ e\ \texttt{with}\ (pat_i \rightarrow E_i)_{i\in[1..n]}\ \texttt{end}]\!]_\rho^{Init} = ([\![e]\!]_\rho^{Init}, [\![E_1]\!]_\rho^{Init}, ..., [\![E_n]\!]_\rho^{Init})
$$

$$
\begin{array}{l}
[\![\texttt{match}\ e\ \texttt{with}\ (pat_i \rightarrow E_i)_{i\in[1..n]}\ \texttt{end}]\!]_\rho^{Step} = \\
\quad \lambda(s, s_1, ..., s_n).let\ v, s' = [\![e]\!]_\rho^{Step}(s)\ in\ ^\star match\ v\ with \\
\qquad\qquad \left(\begin{array}{ll} pat_i \rightarrow & let\ \rho_i, s_i' = [\![E_i]\!]_{\rho+[v|pat_i]}^{Step}(s_i)\ in \\ & \rho_i\ \text{by}\ (\rho[N\backslash N_i]), (s', (s_1, ..., s_i', ..., s_n)) \end{array} \right)_{i\in[1..n]}
\end{array}
$$

where $N = \cup_{i\in[1..n]}(N_i)$ and $N_i = Def(E_i)$

Fig. 5. Semantics of Equations, Blocks and By-case Definitions

of a state variable x initialized with some value e, it is necessary to store the current value of x. The value of *last* x is the value stored in the state. When e has never been computed, the state is initialized with *None*, the same way it is done for the construction `. fby .`. Indeed, a block declaration `local` x `init` e `in` E has the meaning of `local` x `in` $E[lx/\texttt{last}\ x]$ `and` $lx = e$ `fby` x where lx is a fresh name, a feature introduced in [25].

The semantics for a conditional `match . with . end` must consider the case where a branch defines a value for a variable `x` in one branch but not in the other branch (e.g., example P13). We take the following convention. When a definition for x is missing in a branch, x takes the default value defined in the current environment in case x have a definition in one other branch. Hence, the environment produced by the branch is in fact ρ_i completed with default values that are defined in ρ. Only the names in N, that is, the union of names defined in the conditional minus those of the current branch are taken, that is, ρ_i by $(\rho[N\backslash N_i])$. When the variable is defined with a default value (x `default` e), x takes the value of `default` x in the environment. Otherwise, $x =$ `last` x. In case x is declared with a last value (x `init` e), it 2 takes the value of `last` x. In case, no initial value is given, the default value is initialized with *nil*.

In the `count` example given previously, when `x = Zero`, `d` implicitly keep its previous value, i.e., `d = last d`. Here, $[0/i]$ by $[v_1/last\ i, v_2/last\ d] = [0/i, v_2/d]$.

3.2 Hierarchical State Machines with Parameters

The language of equations is now extended with hierarchical state machines. The ones we consider (both the notation, semantics and compilation) were introduced in [24, 25]. A simple example given in Figure 6 is the relay controller (the first state `Low` is the initial state). The second is a controller with two modes `nominal` and `failsafe` that compute an output `o`. When `observe(i, o)` is true, the second mode is entered with an initial value. Appendix available at https://zelus.di.ens.fr/zrun/emsoft2023

```
let node relay(low, high, u)
   returns (o)
  automaton
  | Low -> do o = false
     unless (u <= low) then High
  | High -> do o = true
     unless (u >= high) then Low
  end
```

```
let node controller(i, u) returns (o)
  automaton
  | Nominal -> local err in
     do o = nominal(i, u)
     and err = observe(i, o)
     until err then Failsafe(i, o)
  | Failsafe(i, v) -> do o = fail(i, v, u) done
  end
```

Fig. 6. Automata with Parameters

show the Esterel clock watch. Scade automata build on a synchronous interpretation [3] of the Statecharts [33]. An automaton is defined by a finite set of states, a sequence of transitions between states that can be by "reset" or by "history"; "weak" or "strong". Compared to [25], we consider a small generalisation of automata where states can be parameterized [24], a language feature implemented in Lucid Synchrone [49] and Zélus [19]. It is a useful feature to transmit informations between states of an automaton which, otherwise, would need shared variables, a less elegant solution. We consider it because it semantics is not longer nor more complex than the simpler situation where all states are non parameterized. The syntax is extended in the following way.

$$
\begin{array}{lcl}
E & ::= & \ldots \mid \texttt{automaton}\ (A(p) \rightarrow u\ tr)^+\ \texttt{init}\ se \\
u & ::= & \texttt{local}\ loc\ \texttt{in}\ u \mid \texttt{do}\ E \\
tr & ::= & \texttt{until}\ t^* \mid \texttt{unless}\ t^* \\
t & ::= & e\ \texttt{then}\ A(e, ..., e) \mid e\ \texttt{continue}\ A(e, ..., e) \\
se & ::= & A(e, \ldots, e) \mid \texttt{if}\ e\ \texttt{then}\ se\ \texttt{else}\ se
\end{array}
$$

t^* stands for the repetition of a transition t (ϵ denotes the empty repetition). Transitions can be weak ($\texttt{until}\ t^*$) or strong ($\texttt{unless}\ t^*$). Each transition can reset the target state ($e\ \texttt{then}\ A(e, ..., e)$) or enter by history ($e\ \texttt{continue}\ A(e, ..., e)$). When the list of parameters (or arguments) of a state A is empty, we write A instead of $A()$. The initialization state expression se does not need to be given (the notation is simply $\texttt{automaton}\ (A_i(p_i) \rightarrow u_i\ tr_i)_{i \in [0..n]}$). In that case, $se = A_0()$ where A_0 is the first state name in the list.

The semantics of hierarchical automata is defined in Figures 7, 8 and 9. Figure 7 defines the initial state for all the elements of an automaton and the step function. The step function of an automaton takes a memory state (s, v, r, su, sw) and returns a new one. The intuition is that the initial state for the transition function of an automaton is of the form $(s, None, false, (su_1, \ldots, su_n), (sw_1, \ldots, sw_n))$ where s is the state used to compute the current value of se. *None* means that the initial state of the automaton is unknown (exactly as for a `fby`). *false* means that the current active state of the automaton does not need to be reset. $(su_1, ..., su_n)$ is the initial state associated to the bodies u_i; $(sw_1, ..., sw_n)$ is the initial memory for the transitions t_i (sw stands for the *state* of *weak* transitions; ss will stands for the *state* of *strong* transitions).

The step function of an automaton takes a state (s, v, r, su, sw) and returns a new one. It uses an auxiliary function:

$$[\![(A_i(p_i) \rightarrow u_i\ tr_i)_{i \in [1..n]}]\!]_\rho^{v,r}(su, sw)$$

Given v the current active state of the automaton (of the form $A_i(p_i)$), the reset bit r, the memory state su for the body of the automaton $(u_i)_{i \in [1..n]}$ and sw for transitions $(tr_i)_{i \in [1..n]}$, it returns an environment ρ, a new state name v', a new reset name r', and a new memory state (su', sw').

$$
\begin{array}{l}
[\![\texttt{automaton}\ (A_i(p_i) \longrightarrow u_i\ tr_i)_{i\in[1..n]}\ \texttt{init}\ se]\!]^{Init}_{\rho} = \\
\quad let\ s' = [\![se]\!]^{Init}_{\rho}\ in \\
\quad let\ (su'_i = [\![u_i]\!]^{Init}_{\rho})_{i\in[1..n]}\ in\ let\ (sw'_i = [\![tr_i]\!]^{Init}_{\rho})_{i\in[1..n]}\ in\ (s', None, false, (su'_1, \ldots, su'_n), (sw'_1, \ldots, sw'_n)) \\
\\
[\![\texttt{automaton}\ (A_i(p_i) \longrightarrow u_i\ tr_i)_{i\in[1..n]}\ \texttt{init}\ se]\!]^{Step}_{\rho}(s, v, r, su, sw) = \\
\quad let\ w, s' = [\![se]\!]^{Step}_{\rho}(s)\ in \\
\quad let\ v' = v\ init\ w\ in\ let\ (\rho, v, r'), (su', sw') = [\![(A_i(p_i) \longrightarrow u_i\ tr_i)_{i\in[1..n]}]\!]^{v',r}_{\rho}(su, sw)\ in \\
\qquad\qquad (\rho, (s', Some(v), r', su', sw'))
\end{array}
$$

Fig. 7. The Semantics of Hierarchical Automata (I) — the initial state

$$
\begin{array}{l}
[\![(A_i(p_i) \longrightarrow u_i\ \texttt{until}\ t^*_i)_{i\in[1..n]}]\!]^{v,r}_{\rho}((su_1, \ldots, su_n), (sw_1, \ldots, sw_n)) = \\
\quad \star match\ v\ with \\
\quad \left(\begin{array}{l} A_i(p_i) \longrightarrow let\ \rho', su'_i = [\![u_i]\!]^{r}_{\rho+[A_i(p_i)|v]}(su_i)\ in \\ \qquad\qquad\qquad let\ (v', r'), sw'_i = [\![t^*_i]\!]^{v,r}_{\rho+[A_i(p_i)|v]+\rho'}(sw_i)\ in \\ \qquad\qquad\qquad (\rho, (v', r', (su_1, \ldots, su'_i, \ldots, su_n), (sw_1, \ldots, sw'_i, \ldots, sw_n))) \end{array} \right)_{i\in[1..n]} \\
\\
[\![(A_i(p_i) \longrightarrow u_i\ \texttt{unless}\ t^*_i)_{i\in[1..n]}]\!]^{v,r}_{\rho}((su_1, \ldots, su_n), (ss_1, \ldots, ss_n)) = \\
let\ (v', r', (ss'_1, \ldots, ss'_n)) = \\
\star match\ v\ with \left(A_i(p_i) \longrightarrow let\ (v', r'), ss'_i = [\![t^*_i]\!]^{v,r}_{\rho+[A_i(p_i)|v]}(ss_i)\ in\ (v', r', (ss_1, \ldots, ss'_i, \ldots, ss_n)) \right)_{i\in[1..n]} \\
in\ \star match\ v'\ with \\
\quad \left(\begin{array}{l} A_i(p_i) \longrightarrow let\ \rho', su'_i = [\![u_i]\!]^{r'}_{\rho+[A_i(p_i)|v']}(su_i)\ in \\ \qquad\qquad\qquad (\rho', (v', r', (su_1, \ldots, su'_i, \ldots, su_n), (ss'_1, \ldots, ss'_n))) \end{array} \right)_{i\in[1..n]}
\end{array}
$$

Fig. 8. The Semantics of Hierarchical Automata (II) — the step function

To define this function, we need an auxiliary function. Given a body u, a reset condition r, an environment ρ and state s, $[\![u]\!]^r_\rho(s)$ resets u when r is true, that is:

$$[\![u]\!]^r_\rho(s) \quad = \quad [\![u]\!]^{Step}_\rho(if\ r\ then\ [\![u]\!]^{Init}_\rho\ else\ s)$$

Given a transition t, a state value v of an automaton, a reset condition r:

$$[\![t]\!]^{v,r}_\rho(s) = v', r', s'$$

returns a new state automaton v', a new reset r' and new state s'. We shall instantiate if for the two cases: t can be a weak transition (wt) or a strong transition (st). The semantics of those functions is defined in Figure 8 and 9. We explain them below.

For an automaton with weak transitions (that is, where they are all of the form `until` .), if the current active state when entering the automaton is $v = A_k(v_k)$, the first state A_k in order that matches v is selected. The body u_i is executed in the current state su_i. The execution defines an environment ρ and a new state su'_i. Then weak transitions wt_i are executed under the current memory state sw_i. This execution returns a new value for v (next state of the automaton), a reset condition r and a new memory state sw'_i. This is the first definition in Figure 8. For an automaton with strong transition (that is, where they are all of the form `unless` .), the execution is done in two steps: (1) strong transitions are first evaluated. They determine the actual current state $v' = A_k(v'_k)$ of the automaton and the reset condition r'; (2) according to the value v', the corresponding body of equations is executed.

$$
\begin{array}{llll}
[\![\texttt{until}\ t^*]\!]_\rho^{Init} & = & [\![t^*]\!]_\rho^{Init} & [\![\epsilon]\!]_\rho^{Init} = () \\
[\![\texttt{unless}\ t^*]\!]_\rho^{Init} & = & [\![t^*]\!]_\rho^{Init} & [\![e\ \texttt{then}\ se\ t^*]\!]_\rho^{Init} = (([\![e]\!]_\rho^{Init}, [\![se]\!]_\rho^{Init}), [\![t^*]\!]_\rho^{Init}) \\
[\![\texttt{until}\ t^*]\!]_\rho^{v,r}(s) & = & [\![t^*]\!]_\rho^{v,r}(s) & [\![e\ \texttt{continue}\ se\ t^*]\!]_\rho^{Init} = (([\![e]\!]_\rho^{Init}, [\![se]\!]_\rho^{Init}), [\![t^*]\!]_\rho^{Init}) \\
[\![\texttt{unless}\ t^*]\!]_\rho^{v,r}(s) & = & [\![t^*]\!]_\rho^{v,r}(s) & [\![\epsilon]\!]_\rho^{v,r}(s) = ((v,r),s)
\end{array}
$$

$$
\begin{array}{l}
[\![e_1\ \texttt{then}\ se_2\ t^*]\!]_\rho^{v,r}((s_1,s_2),s_3) = \mathit{let}\ (s_1,s_2,s_3) = \mathit{if}\ r\ \mathit{then}\ ([\![e_1]\!]_\rho^{Init}, [\![se_2]\!]_\rho^{Init}, [\![t^*]\!]_\rho^{Init})\ \mathit{else}\ (s_1,s_2,s_3)\ \mathit{in} \\
\qquad \mathit{let}\ (v_1,s_1 = [\![e_1]\!]_\rho^{Step}(s_1))\ \mathit{and}\ (v_2,s_2 = [\![se_2]\!]_\rho^{Step}(s_2))\ \mathit{in} \\
\qquad \mathit{let}\ (v_3,r_3),s_3 = [\![t^*]\!]_\rho^{v,r}(s_3)\ \mathit{in} \\
\qquad {}^\star\mathit{if}\ v_1\ \mathit{then}\ ((v_2,\mathit{true}),((s_1,s_2),s_3))\ \mathit{else}\ ((v_3,r_3),((s_1,s_2),s_3))
\end{array}
$$

$$
\begin{array}{l}
[\![e_1\ \texttt{continue}\ se_2\ t^*]\!]_\rho^{v,r}((s_1,s_2),s_3) = \mathit{let}\ (s_1,s_2,s_3) = \mathit{if}\ r\ \mathit{then}\ ([\![e]\!]_\rho^{Init}, [\![se]\!]_\rho^{Init}, [\![t^*]\!]_\rho^{Init})\ \mathit{else}\ (s_1,s_2,s_3)\ \mathit{in} \\
\qquad \mathit{let}\ (v_1,s_1 = [\![e_1]\!]_\rho^{Step}(s_1))\ \mathit{and}\ (v_2,s_2 = [\![se_2]\!]_\rho^{Step}(s_2))\ \mathit{in} \\
\qquad \mathit{let}\ (v_3,r_3),s_3 = [\![t^*]\!]_\rho^{v,r}(s_3)\ \mathit{in} \\
\qquad {}^\star\mathit{if}\ v_1\ \mathit{then}\ ((v_2,\mathit{false}),((s_1,s_2),s_3))\ \mathit{else}\ ((v_3,r_3),((s_1,s_2),s_3))
\end{array}
$$

$$
\begin{array}{l}
[\![A(e_1,...,e_n)]\!]_\rho^{Init} = [\![e_1]\!]_\rho^{Init}, ..., [\![e_n]\!]_\rho^{Init} \\
[\![A(e_1,...,e_n)]\!]_\rho^{Step} = \mathit{let}\ (v_i,s_i' = [\![e_i]\!]_\rho^{Step}(s_i))_{i\in[1..n]}\ \mathit{in}\ A(v_1,...,v_n),(s_1',...,s_n')
\end{array}
$$

Fig. 9. The Semantics of Hierarchical Automata (III) — transitions

For the semantics of a transitions given in Figure 9, they are evaluated in order. When a reset condition r is true, this means that the whole memory of the target automaton state must be reset. Moreover, in the state of an automaton, all the conditions are evaluated.

The state-based semantics extends naturally to hierarchical automata and is only one page long. It was a surprise that a computational semantics would not be longer nor more complex than a relational synchronous semantics like the one in [24].

Remark 4. *Mix of weak and Strong Transitions Contrary to [25], we have considered a small variant where weak and strong transitions cannot be mixed in the very same automaton. Transitions in an automaton are all weak or all strong. Automata also come with a way to compute the initial state of the automaton. The general case would be easily obtained by composing functions given in Figure 8.*

This simplification was introduced in Zélus [19] because programs are simpler to understand as well as the semantics, and the generated code is more efficient. In practice, we never found real examples with mixed (weak/strong) transitions that cannot be expressed in this simpler form.

4 CAUSALITY

With a relational semantics, some programs are non deterministic because several valuations for variables are possible; some have no solution, i.e, they deadlock. For example, the node `non_deterministic` below can output `true` or `false` whereas it outputs $\bot$ with a functional semantics. The node `deadlock` has a deadlock because there is no value such that $a = a + 1$. It outputs $\bot$ with a functional semantics. The equation for `tobe` in node `hamlet`[12] has a unique boolean solution `tobe` = *true* whereas the least fix-point is $\bot$ with a functional semantics.

```
1 let node non_deterministic() returns (o)
2   local a, b in a = b or b and b = a and o = a
3 let node deadlock() returns a = a + 1
4 let node hamlet() = tobe where rec tobe = tobe or not(tobe)
5 let node hamlet() = tobe where rec tobe = if tobe then true else not(tobe)
```

[12]The example and name is by G. Berry.

The functional constructive semantics is modular, by construction: no information is lost by node abstraction. E.g., the two functions `main1()` and `main2()` below produce the same streams. If the function call `delay` is inlined, the semantics is unchanged:

```
let node delay(x) = 0 fby x
let node main1() = o where rec o = 0 fby (o + 1)
let node main2() = o where rec o = delay(o + 1)
```

Adapting the definition of [11] to a declarative language, an expression is said to be *causally correct* if, provided all of its free variables do not produce bottom, its value does not produce bottom. This property is dynamic. In the accompanying implementation, it is implemented as a dynamic test that may return a specific error.

The causality analysis performed by a synchronous language compiler computes an over-approximation, e.g., based on data-dependences: *all static dependency cycles must be broken by a unit delay*. This is called "syntactic causality" in Lustre [21] or "ASC-criterion" in SCCharts [56]. The analysis can be done after the static inlining of function calls (as the Lustre and SCCharts compiler are doing) or made modularly by building a type signature for every function definition that represent input/output dependences. This is what the Scade [27] and Zélus [4] compilers do.

Still, the constructive semantics is useful as a reference to characterize the set of programs that are causally correct no matter what a compile-time static analysis is able to accept. For instance, the following program `cyclic(c,x)` encodes a cyclic combinational circuit [41] for which the Scade compiler complains by returning: `Causality error: the definition of flow x2 depends on flow y1; the definition of flow y1 depends on flow x1; the definition of flow x1 depends on flow y2; the definition of flow y2 depends on flow x2;` No code is generated.

```
let node mux(c, x, y) = o where rec o = if c then x else y
let node cyclic(c, x) = y where
  rec x1 = mux(c, x, y2) and x2 = mux(c, y1, x)
  and y1 = f(x1) and y2 = g(x2) and y = mux(c, y2, y1)
let node acyclic(c, x) = mux(c, g(f(x)), f(g(x)))
let node check(c, x) = (cyclic(c, x) = acyclic(c, x))
```

While this program is statically rejected by the compiler, it is nonetheless causally correct at run-time: when `c` is true, `y` equals `g(f(x))`; when `c` is false, `y` equals `f(g(x))`. It is (dynamically) causal because the conditional `if . then . else .` in the `mux` nodes is interpreted by $^\star if\ .\ then\ .\ else\ .$ which is only strict in its first argument. Expressed with such a ("lazy") conditional, boolean operators are strict in their first arguments:

$$or(x,y) = \texttt{if}\ x\ \texttt{then true else}\ y \qquad and(x,y) = \texttt{if}\ x\ \texttt{then}\ y\ \texttt{else false}$$

In particular, $or(\bot, true) = \bot$ and $or(true, \bot) = true$. In the theory of causality analysis [52], these definitions are different ternary extensions of the boolean operators *or* and *and*. They are "smaller" (tighter, more strict) than the extensions used by constructive Esterel, but "larger" (looser, more lazy) than the strict operators of Lustre and Scade. Indeed, if we now interpret the conditional as being strict in all of its arguments, that is:

$$^\star if\ \bot\ then\ \_\ else\ \_ \quad =_{df} \quad ^\star if\ \_\ then\ \bot\ else\ \_ \quad =_{df} \quad ^\star if\ \_\ then\ \_\ else\ \bot \quad =_{df} \quad \bot$$
$$^\star if\ true\ then\ x\ else\ \_ \quad =_{df} \quad ^\star if\ false\ then\ \_\ else\ x \quad =_{df} \quad x$$

the function `cyclic(c, x)` is not causally correct. Variables `x1`, `x2`, `y1`, `y2` and `y` all evaluate to $\bot$. The maximal ternary extension of a boolean function in the sense of [52] will be causal in a maximal number of contexts, but requires more compilation effort and run-time overhead. The strict lifting is most simply implemented, but may deadlock. The compiler needs to find the right trade-off.

```
1  let node sequential_or_gate(x,y) returns (z) if x then z = true else z = y

3  let node arbiter(i, request, pass_in, tok_in) returns (grant, pass_out, tok_out)
4    local o in
5    do grant = and_gate(request, o) and pass_out = and_gate(not request, o)
6    and o = sequential_or_gate (tok_in, pass_in) and tok_out = i fby tok_in done

8  let node arbiter_three(req1, req2, req3) returns (grant1, grant2, grant3)
9    local pass_out1, pass_out2, pass_out3, tok_out1, tok_out2, tok_out3 in
10   do grant1, pass_out1, tok_out1 = arbiter(true, req1, pass_out3, tok_out3)
11   and grant2, pass_out2, tok_out2 = arbiter(false, req2, pass_out1, tok_out1)
12   and grant3, pass_out3, tok_out3 = arbiter(false, req3, pass_out2, tok_out2) done
```

Fig. 10. The Bus Arbiter

The bus arbiter of R. de Simone described in [28] and given in Figure 10 is an even more interesting example because causality depends on values and on a finer interpretation of boolean operations. If boolean operators are interpreted strictly, that is, their output is bottom as soon as one of their input is bottom (as any other operators lifted with *.), the node `arbiter_three` is not causally correct: the variables output `grant1`, `grand2` and `grant3` take the value $\bot$. If, instead of interpreting *or* and *and* strictly, the operators are interpreted as maximal ternary extensions in three-value logic, the program is causal:

$$\begin{array}{llllllllll} {}^\sharp or(true,\_) & =_{df} & {}^\sharp or(\_,true) & =_{df} & true & {}^\sharp and(false,\_) & =_{df} & {}^\sharp and(\_,false) & =_{df} & false \\ {}^\sharp or(false,x) & =_{df} & {}^\sharp or(x,false) & =_{df} & x & {}^\sharp and(true,x) & =_{df} & {}^\sharp and(x,true) & =_{df} & x \end{array}$$

In particular, ${}^\sharp or(true,\bot) = {}^\sharp or(\bot,true) = true$. An even simpler way is to keep the boolean operators strict but use a ternary extension of the conditional

$${}^\sharp if\ \bot\ then\, v_1\ else\, v_2 \ =_{df}\ {}^\star if\ {}^\star{=}(v_1,v_2)\ then\, v_1\ else\, \bot \qquad {}^\sharp if\ x\ then\, v_1\ else\, v_2 \ =_{df}\ {}^\star if\ x\ then\, v_1\ else\, v_2$$

with the following definition for equality:

$$\begin{array}{llll} {}^\star{=}(\bot,\_) & =_{df} & \bot & {}^\star{=}(v_1,v_2) \quad =_{df} \quad (v_1 = v_2) \text{ if } atomic(v_1) \wedge atomic(v_2) \\ {}^\star{=}(\_,\bot) & =_{df} & \bot & {}^\star{=}(v_1,v_2) \quad =_{df} \quad \bot \text{ otherwise} \end{array}$$

It the two branches of a conditional produce the same value v, the conditional can output v, whatever be the value of the condition. This definition is possible because all imported operations are total (they terminate at every instant), that is, equality is decidable. Hence, it is possible to obtain an Esterel-like interpretation for Scade programs simply by changing the definition of the conditional. This experiment is implemented in the companion implementation.

While testing the semantics on the example given in [28], we observed that the bus arbiter does not need three-valued logic. Indeed, it is causally correct when the two *and* gates (line 5) are strict and the *or* gate on line 6 is simply sequential, that is, encoded with a lazy conditional. This is because `token_in` is an output of a unit delay, hence its value is known. If we flip the order of arguments and write `o = sequential_or_gate (pass_in, token_in)` instead, the program is not causal anymore (that is, `o` = $\bot$): we do need the full expressiveness of Esterel. As noted by R. de Simone, if none of the three unit delays is initialized with the value `true`, the program is not causal anymore, even under Esterel's dynamic ternary semantics.

Synchronous compilers have to deal with causality, to decide whether equations have a solution or not and if the compiler is able to produce code. This question is not limited to synchronous languages but exists in any language able to express systems with feedback loops. Yet, causality is

```
let node comp(c1, c2, y) = (x, z, t, r)
 where rec
  if c1 then do x = y + 1 and z = t + 1 done
        else do x = 1 and z = 2 done      and
  if c2 then do t = x + 1 and r = z + 2 done
        else do t = 1 and r = 2 done
```

```
let node comp(c1, c2, y) =
  (x, z, t, r) where rec
     if c1 then x = y + 1 else x = 1
 and if c2 then t = x + 1 else t = 1
 and if c1 then z = t + 1 else z = 2
 and if c2 then r = z + 2 else r = 2
```

Fig. 11. An example where control duplication is necessary

not an absolute notion. It depends on the expressiveness that is expected for the language, the target of the compiler (e.g., hardware or software, concurrent or sequential), the possible effect on the generated code (size, efficiency, etc.). On one side, Lustre took the most restrictive approach, forcing that all equations be statically schedulable. All operations except the unit delay are considered strict in all their arguments. Languages like Scade and Zélus are also based on a static over-approximation of data-dependences, rejecting programs with cyclic dependences. Their analysis is a bit more expressive nonetheless, treating specially variables defined by-case in two different branches of a conditional that are, by syntax, exclusive. This covers many useful situations like the program P13 (Section 3.1). The P13 example is correct because in each mode, it is never the case that `o1` depends on `o2` and conversely. Another common example is a system which defines two variables, a position and a speed, with the position that is computed from the speed in one mode, and the speed computed from the position in the other. This approach to static causality works well and is reasonable when the target is software. On the other side, Esterel does a more powerful static causality analysis which allow data-dependent cycles provided they can be computed constructively using the three valued interpretation of boolean operations. It mimics statically what the compilation into circuits does. This solution is reasonable when generating hardware, if cyclic circuits are allowed. It is more debatable when targeting software because of unavoidable code duplication, a more expensive static analysis that is difficult to do modularily (function per function with signatures stored in interfaces) and a more complex code generation scheme.

The constructive semantics we have defined can illustrate those possibilities on a very same program, by changing solely the interpretation of the conditionals and leaving boolean operations unchanged.

The impact of causality on static code generation. The choice of causality may have an impact on the efficiency of the code, even for a strict causality which only accepts equations with acyclic dependences like Scade. Consider the program on the left of Figure 11.

This program is causal: if inputs `c1`, `c2` and `y` are non bottom, all outputs are non bottom. E.g., taking true for `c1` and `c2`, starting with $x_0 = \bot$, $z_0 = \bot$, $t_0 = \bot$ and $r_0 = \bot$, the fixpoint is the limit of the sequence: $x_n = y + 1 \wedge\ z_n = t_{n-1} + 1 \wedge\ t_n = x_{n-1} + 1 \wedge\ r_n = z_{n-1} + 2$ and is obtained after 4 iterations. Nonetheless, if we want to generate statically scheduled sequential code, the control structure must be duplicated, that is, a test `c1` to compute `x`; a test `c2` to compute `t`; a test `c1` (again) to compute `z`; a test `c2` (again) to compute `r`. Accepting programs with inter-wined dependencies impacts code size and efficiency. It would be possible to over-constrain the causality analysis so that control structures are considered to be atomic, that is, in every branch, all outputs are supposed to depend on all inputs but this would reduce expressiveness and modularity. A compromise is to add extra annotations, like `atomic` E, so that all outputs in E are considered to depend on all inputs. Semantically, `atomic` E returns a bottom environment as soon as one free variable is bottom. This construct was introduced in Zélus. Its semantics is given in the companion implementation.

5 CONCLUSION

This work has presented a constructive semantics for a language that has the main programming constructs of Scade and leads to a reference interpreter that is independent of a compiler. By interpreting streams as state transformers and stream functions to be length preserving, a fixpoint on streams is replaced by a fixpoint on values which can be computed exactly and in bounded time. This old idea, introduced first by Gonthier for the semantics of Esterel, works surprisingly well for treating the programming constructs of an expressive language like Scade, in particular the mix of data-flow equations and control structures like hierarchical state machines. The formalization is short and results in an effective interpreter; this is a key contribution. The semantics is defined by a function that is total and constructive in the sense that it can be expressed in a typed functional language where all computations terminate, e.g., the language of a proof assistant. The semantics also accounts for errors by adopting the classical monadic solution. We are not aware of a similarly encompassing approach for a synchronous language.

The presented material is supported by an implementation in OCaml, in purely functional style and a version in Coq, produced automatically that we used to prove some of the source-to-source transformations done by the compiler. The OCaml implementation has been instrumental to define, test and validate the semantics. It could be developed incrementally by enriching the language gradually. With few changes, we were able to prototype new programming constructs, in particular static arguments and array operations. In the future, we would like to make the semantics even more generic so as to define a set-based or symbolic interpreter or to cover other language features like the mix of discrete and continuous-time that exist in languages such as Zélus [19], the reaction to absence as in Esterel, for which a fixpoint semantics exist [2], or adding imperative features for shared memory as introduced in [1, 56].

Finally, we hope that this work can also clarify important, and sometimes misunderstood, differences between Esterel and Scade in terms of expressiveness, particularly regarding how they deal with *causality*. We explain them by different interpretations of the conditional that are justified by different application domain: synchronous circuits for Esterel; software code for Scade.

REFERENCES

[1] Joaquín Aguado, Michael Mendler, Marc Pouzet, Partha S. Roop, and Reinhard von Hanxleden. 2018. Deterministic Concurrency: A Clock-Synchronised Shared Memory Approach. In *ESOP*. Thessaloniki, Greece, 86–113.

[2] J. Aguado, M. Mendler, R. von Hanxleden, and I. Fuhrmann. 2015. Denotational fixed-point semantics for constructive scheduling of synchronous concurrency. *Acta Informatica* 52, 4 (2015), 393–442.

[3] Ch. André. 1996. Representation and Analysis of Reactive Behaviors: A Synchronous Approach. In *CESA*. IEEE-SMC, Lille.

[4] A. Benveniste, T. Bourke, B. Caillaud, B. Pagano, and M. Pouzet. 2014. A Type-based Analysis of Causality Loops in Hybrid Systems Modelers. In *HSCC*. ACM, Berlin, Germany.

[5] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* 91, 1 (Jan. 2003).

[6] A. Benveniste, P. Caspi, R. Lublinerman, and S. Tripakis. 2008. *Actors without Directors: a Kahnian View of Heterogeneous Systems.* Technical Report. Verimag, Centre Équation, 38610 Gières.

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

[8] Gérard Berry. 1989. *Programming a Digital Watch in Esterel V3.* Technical Report 1032. INRIA.

[9] G. Berry. 1989. Real time programming: Special purpose or general purpose languages. *Information Processing* (1989).

[10] G. Berry. 1993. The Semantics of Pure Esterel. *Series F: Computer and System Sciences* 118 (01 1993).

[11] G. Berry. 2002. The Constructive Semantics of Pure Esterel, Draft Version 3. (2002).

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

[13] G. Berry and L. Rieg. 2019. Towards Coq-verified Esterel Semantics and Compiling. *CoRR* abs/1909.12582 (2019).

[14] D. Biernacki, JL. Colaco, G. Hamon, and Marc Pouzet. 2008. Clock-directed Modular Code Generation of Synchronous Data-flow Languages. In *ACM LCTES*. Tucson, Arizona.

[15] S. Boulmé and G. Hamon. 2001. Certifying Synchrony for Free. In *LPAR*, Vol. 2250. La Havana, Cuba.

[16] T. Bourke, L. Brun, P.-É. Dagand, X. Leroy, M. Pouzet, and L. Rieg. 2017. A formally verified compiler for Lustre. In *PLDI*.

[17] T. Bourke, L. Brun, and M. Pouzet. 2020. Mechanized semantics and verified compilation for a dataflow synchronous language with reset. In *POPL*. ACM.

[18] T. Bourke, B. Pesin, and M. Pouzet. 2023. Verified Compilation of Synchronous Dataflow with State Machines. In *EMSOFT*.

[19] T. Bourke and M. Pouzet. 2013. Zélus, a Synchronous Language with ODEs. In *HSCC*. ACM, Philadelphia, USA.

[20] P. Caspi. 1992. Clocks in dataflow languages. *Theoretical Computer Science* 94 (1992), 125–140.

[21] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. 1987. Lustre: A Declarative Language for Programming Synchronous Systems. In *POPL*. ACM.

[22] P. Caspi and M. Pouzet. 1996. Synchronous Kahn Networks. In *ACM ICFP*. Philadelphia, Pensylvania.

[23] P. Caspi and M. Pouzet. 1998. A Co-iterative Characterization of Synchronous Stream Functions. In *CMCS'98*.

[24] JL. Colaço, G. Hamon, and M. Pouzet. 2006. Mixing Signals and Modes in Synchronous Data-flow Systems. In *ACM EMSOFT*. Seoul, South Korea.

[25] JL. Colaço, B. Pagano, and M. Pouzet. 2005. A Conservative Extension of Synchronous Data-flow with State Machines. In *ACM EMSOFT*. Jersey city, New Jersey, USA.

[26] JL. Colaço and M. Pouzet. 2003. Clocks as First Class Abstract Types. In *ACM EMSOFT*. Philadelphia, USA.

[27] JL. Colaco, B. Pagano, and M. Pouzet. 2017. Scade 6: A Formal Language for Embedded Critical Software Development. In *Symposium on Theoretical Aspect of Software Engineering (TASE)*. Sophia Antipolis, France.

[28] S. A. Edwards and E. A. Lee. 2003. The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming* 48 (2003), 21–42.

[29] G. Gonthier. 1988. *Sémantiques et modèles d'exécution des langages réactifs synchrones*. Ph. D. Dissertation. Université d'Orsay.

[30] N. Halbwachs. 1984. *Modélisation et analyse du comportement des systèmes informatiques temporisés*. Ph. D. Dissertation. Institut National Polytechnique de Grenoble - INPG; Université Joseph - Fourier - Grenoble I.

[31] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The Synchronous Dataflow Programming Language Lustre. *Proc. IEEE* 79, 9 (September 1991), 1305–1320.

[32] N. Halbwachs, P. Raymond, and C. Ratel. 1991. Generating Efficient Code From Data-Flow Programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*. Passau (Germany).

[33] D. Harel. 1987. StateCharts: a Visual Approach to Complex Systems. *Science of Computer Programming* 8-3 (1987), 231–275.

[34] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. 2003. *Arrows, Robots, and Functional Reactive Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 159–187.

[35] John Hughes. 2000. Generalising Monads to Arrows. *Science of Computer Programming* 37 (2000), 67–111.

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

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

[38] E. A. Lee and A. Sangiovanni-Vincentelli. 1998. A Framework for comparing models of computation. *IEEE Transactions on CAD* 17, 12 (December 1998).

[39] Xavier Leroy. 2021. The Compcert verified compiler. http://compcert.inria.fr/doc/index.html.

[40] Hai Liu, Eric Cheng, and Paul Hudak. 2011. Causal commutative arrows. *J. Funct. Program.* 21, 4-5 (2011), 467–496.

[41] S. Malik. 1994. Analysis of cyclic combinational circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems* 13, 7 (1994).

[42] F. Maraninchi. 1991. The Argos Language: Graphical Representation of Automata and Description of Reactive Systems. In *IEEE Workshop on Visual Languages*. Kobe, Japan.

[43] F. Maraninchi and F. Gaucher. 2000. Step-wise + Algorithmic debugging for Reactive Programs: LuDiC, a debugger for Lustre. In *AADEBUG'2000 – Fourth International Workshop on Automated Debugging*. Munich.

[44] F. Maraninchi and Y. Rémond. 2003. Mode-Automata: a new Domain-Specific Construct for the Development of Safe Critical Systems. *Science of Computer Programming* 46 (2003), 219–254.

[45] G. H. Mealy. 1955. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal* 34, 5 (1955), 1045–1079.

[46] R. Paterson. 2001. A New Notation for Arrows. In *ICFP* (Firenze, Italy). ACM Press, 229–240.

[47] C. Paulin-Mohring. 1995. Circuits as Streams in Coq: Verification of a Sequential Multiplier. In *TYPES*. Springer.

[48] Ch. Paulin-Mohring. 2009. A constructive denotational semantics for Kahn networks in Coq. In *From Semantics to Computer Science*, Y. Bertot, G. Huet, JJ. Lévy, and G. Plotkin (Eds.). Cambridge University Press, 383–413.

[49] M. Pouzet. 2006. *Lucid Synchrone, version 3. Tutorial and reference manual.* Université Paris-Sud, LRI.
[50] John C. Reynolds. 1998. *Theories of Programming Languages.* Cambridge University Press.
[51] Klaus Schneider and Jens Brandt. 2016. *Handbook of Hardware/Software Codesign.* S. Ha and J. Teich (Editor); Springer Science+Business Media Dordrecht, Chapter Quartz: A Synchronous Language for Model-Based Design of Reactive Embedded Systems.
[52] Klaus Schneider, Jens Brandt, Tobias Schüle, and Thomas Türk. 2005. Maximal Causality Analysis. In *Conference on Application of Concurrency to System Design (ACSD'05).* St. Malo, France, 106–115.
[53] Enno Scholz. 1998. Imperative streams—a monadic combinator library for synchronous programming. In *International Conference on Functional Programming (ICFP 1998).* ACM, 261–272.
[54] O. Tardieu. 2004. A Deterministic Logical Semantics for Esterel. In *SOS Workshop.* London, United Kingdom.
[55] Tarmo Uustalu and Varmo Vene. 2005. Signals and Comonads. *J. Univers. Comput. Sci.* 11, 7 (2005), 1310–1326. https://doi.org/10.3217/jucs-011-07-1311
[56] R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. O'Brien. 2014. SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications. In *PLDI'14.*

6 APPENDIX

6.1 Example: the Esterel stopwatch

The stopwatch [8] is an old example of a synchronous program written in Esterel. We give below a data-flow version in the syntax of the core language used in this paper. It is available with the source code distribution and can be tested with the interpreter.

```
(* ------------------------- Watch Interface------------------------
-- stst : start/stop button
-- rst : reset button
-- set : set time button
-- md : mode selection button
-- a1, a2, a3 : time data display
-- l_ : is displaying lap time
-- s_ : is in setting time mode
-- sh_ : is in setting hour mode
-- s_ and not sh_ : is in setting minutes mode
------------------------------------------------------------------- *)

let node root (stst,rst,set,md) returns (a1, a2, a3, l_, s_, sh_ )
local
  isStart default false, (* -- is the chrono started? *)
  is_w default false, (* -- is watch in clock mode? *)
  sigS default false,
  sigSh default false,
  sigL default false,
  m init 0, s init 0, d init 0, (* -- chrono timers *)
  last wh, last wm, last ws, w (* -- clock timers *) in
do l_ =  sigL
and s_ =  sigS
and sh_ =  sigSh
and automaton (* -- Chrono ----------------------*)
    | Stop ->
        do m, s, d = (0, 0, 0) -> (last m, last s, last d)
        unless (stst && not is_w) continue Start
        unless (rst && not (false -> pre l_) && not is_w) then Stop
```

```
    | Start ->
        do d = (last d + 1) mod 100
        and s = (if (d < last d) then last s + 1 else last s) mod 60
        and m = if (s < last s) then last m + 1 else last m
        and isStart = true
        unless (stst && not is_w) continue Stop
    end
and automaton (* -- Watch ------------------*)
    | Count ->
        do wm = 0 -> (if (ws < last ws)
                      then last wm + 1 else last wm) mod 60
        and wh = 0 -> (if (wm < last wm)
                       then last wh + 1 else last wh) mod 24
        until (set && is_w) then Set
  | Set -> (* -- Set time *)
      local synchro default false in
      do sigS = true
      and automaton (* -- set Watch -----------*)
          | Set_hr -> (* -- set hour first *)
              do sigSh = true
              and wh = (if stst then last wh + 1
                        else if rst then last wh +23
                        else last wh) mod 24
              until set then Set_mn
          | Set_mn -> (* -- then set minutes *)
              do wm = (if stst then last wm + 1
                        else if rst then last wm +59
                        else last wm) mod 60
              until set then Set_end
          | Set_end -> do synchro = true done
          end
     until synchro continue Count
  end
and w = 0 -> (pre w + 1) mod 100
and ws = 0 -> (if (w < pre w) then pre ws + 1 else pre ws) mod 60
and automaton (* -- Display ----------------*)
    | DispClk -> (* -- display watch *)
        do is_w = true
        and a1, a2, a3 = (wh, wm, ws)
        unless (md && not s_) continue DispChr
    | DispChr ->(* -- display chrono *)
        local
          lm init 0, ls init 0, ld init 0 in
          (* -- chrono display (to deal with lap time) *)
        do a1, a2, a3 = (lm, ls, ld)
        and automaton (* -- deal with lap time and current time ---*)
            | DispTime ->
               do lm, ls, ld = (m, s, d)
```

$$
\begin{array}{lcl}
y^0.step & = & \lambda s.(\bot, s) \\
y^0.init & = & \bot \\
y^1.step & = & \lambda s.let\ v, s' = y^0.step\ s\ in\ ft\ s\ v \\
 & = & \lambda s.ft\ s\ \bot \\
 & = & \lambda s.s, \bot \\
y^1.init & = & 0
\end{array}
$$

$$
\begin{array}{lcl}
y^2.step & = & \lambda s.let\ v, s' = y^1.step\ s\ in\ ft\ s\ v \\
 & = & \lambda s.let\ v = s\ in\ ft\ s\ v \\
 & = & \lambda s.let\ v = s\ in\ s, v + 1 \\
 & = & \lambda s.s, s + 1 \\
y^2.init & = & 0
\end{array}
$$

$$
\begin{array}{lcl}
y^3.step & = & \lambda s.let\ v, s' = y^2.step\ s\ in\ f_s\ s\ v \\
 & = & \lambda s.let\ v = s\ in\ ft\ s\ v \\
 & = & \lambda s.let\ v = s\ in\ s, v + 1 \\
 & = & \lambda s.s, s + 1 \\
y^3.init & = & 0
\end{array}
$$

Fig. 12. Unfolding the semantics

```
78              unless (rst && isStart) then DispLap
79            | DispLap ->
80              do sigL = true
81              unless (rst) then DispTime
82            end
83      unless md continue DispClk
84  end
85 done
```

6.2 Program Transformations and Static Scheduling

We shall see now conditions under which recursions can be removed inside the step function. Consider the stream equation:

```
1 let rec nat = 0 fby (nat + 1) in nat
```

Can we get rid of the recursion in this definition? Surely we can, since it can be compiled into the non recursive step function $nat = Co(\lambda s.(s, s + 1), 0)$. This code can be obtained by a simple unfolding of the semantics. `nat` is the solution of the fix-point equation `y = f(y)` where:

```
1   let node f x = 0 fby (x+1)
```

Let us write $CoP(ft, s_0) = CoP(\lambda s, x.s, (x + 1), 0)$, the Mealy machine that implements f. ft and s_0 can be obtained by unfolding the definitions of $[\![.]\!]^{Init}_{.}$ and $[\![.]\!]^{Step}_{.}$ on the definition of f. Indeed, $[\![0\ \texttt{fby}\ x + 1]\!]^{Init}_{[\bot/x]} = (0, (((), ())))$. The transition function is $\lambda(m, s), v.[\![0\ \texttt{fby}\ x + 1]\!]^{Step}_{[v/x]}(m, s)$, that is, $\lambda(m, s), v.m, [\![x + 1]\!]^{Step}_{[v/x]}(s)$, that is, $\lambda(m, s), v.m, (x + 1, s)$. That is:

$$ft = \lambda(m, s), v.m, (x + 1, s) \quad \text{and} \quad s_0 = (0, (((), ())))$$

This could further be simplified into:

$$ft = \lambda m, v.m, (x + 1, s) \quad \text{and} \quad s_0 = 0$$

Now, solving $y = f(y)$ amount at finding a transition function Y such that:

$$Y(s) = let\ v, s' = Y(s)\ in\ ft\ s\ v$$

Let $y = CoF(g_y, s_y)$. We write $y.step$ for g_y and $y.init$ for s_y. The code for equation $y = f(y)$ can be obtained iteratively, by a simple unfolding of semantics definitions. As the recursion is on a single variable, two iterations are enough.

If we execute the semantics symbolically and propagate $\bot$, as shown in Figure 12, we obtain:

$$CoF(\lambda s.(s+1, s+1), 0)$$

This example gives an insight of a different use of the semantics. Instead of interpreting operations as functions from concrete values to concrete values, operations can be interpreted symbolically as functions from terms to terms like compilation operators. Terms can be from the same language, e.g., a subset language or an other one. The resulting language can be a simply typed functional language with call-by-value and a bounded recursion operation or one with a recursion operation (e.g., Haskell or OCaml with a combination of `lazy` and `force` with call-by-need so that the fix-point is computed lazilly at every reaction. The later is how the very first interpreter of Lucid Synchrone was done, following [23].

Recursion on a Single Variable. Suppose that f is a length preserving function with value value $CoP(ft, s)$ where ft is the transition function and s is its initial state. The semantics of an expression `let rec` y `=` $f(y)$ `in` y is:

$$\begin{aligned}
[\![\texttt{let rec } y = f(y) \texttt{ in } y]\!]_\rho^{Init} &= s \\
[\![\texttt{let rec } y = f(y) \texttt{ in } y]\!]_\rho^{Step} &= \lambda s.let\ rec\ v, s' = ft\ s\ v\ in\ v, s'
\end{aligned}$$

We purposely keep the notation of a recursion on value for v to highlight the fact that v should verify the equation $v, s' = ft\ s\ v$ (even if we know that it is actually a bounded iteration). Two cases can happen:

- Either the first element of the pair $ft\ s\ v$, that is v, s' depends on v and we will get bottom values;
- or it does not and the evaluation succeeds.

When the program does not contain any causality loop, it means that indeed the recursive evaluation of the pair v, s' can be split into two non recursive ones. This case appears, for example, when every stream recursion appears on the right of a unit delay `pre` or `fby`. A synchronous compiler takes advantage of this in order to produce non recursive code like the co-iterative *nat* expression given above. Consider a variant of the above equation but where y appears on the right of a unit delay:

$$\begin{aligned}
[\![\texttt{let rec } y = f(v \texttt{ fby } y) \texttt{ in } y]\!]_\rho^{Init} &= (v, s_t) \\
[\![\texttt{let rec } y = f(v \texttt{ fby } y) \texttt{ in } y]\!]_\rho^{Step} &= \lambda(m, s).let\ rec\ v, s' = ft\ s\ m\ in\ v, (v, s')
\end{aligned}$$

This time, the recursion is no more necessary, that is:

$$[\![\texttt{let rec } y = f(v \texttt{ fby } y) \texttt{ in } y]\!]_\rho^{Step} = \lambda(m, s).let\ v, s' = ft\ s\ m\ in\ v, (v, s')$$

So, when a variable y in an equation $y = e$ appears on the right of a unit delay, the step function is a simple non recursive computation that first computes the value of y and then update some state variables which may depend on y.

Mutually Recursive Equations. Consider the following mutually recursive equation (see the examples given in the beginning of Section 2).

```
let rec sin = 0.0 fby (sin +. h *. cos)
and cos = 1.0 -> (pre cos) -. h *. sin in
sin, cos
```

These two equations can be rewriting into:

```
let rec sin = 0.0 fby sin_next  and pre_cos = pre cos
    and sin_next = sin +. h *. cos
    and cos = if i then 1.0 else pre_cos +. h *. sin and i = true fby false
in sin, cos
```

All the delays, that is,-> , pre and fby have been un-nested and their result given a name. This is the so-called "normalization" step of a synchronous compiler [14]). Then, equations are statically scheduled so that an equation that reads x is scheduled after the equation that computes x. This steps corresponds to the following step of static scheduling [14]. If we compute the transition function corresponding to the set of equations, we obtain:

$$\begin{array}{l}\lambda(m_1, m_2, m_3).let\ sin = m_1\ in\\ \qquad\qquad let\ i = m_3\ in\\ \qquad\qquad let\ pre\_cos = m_2\ in\\ \qquad\qquad let\ sin\_next = sin + .h * .cos\ in\\ \qquad\qquad let\ cos = \texttt{if}\ i\ \texttt{then}\ 1.0\ \texttt{else}\ pre\_cos + .h * .sin\ in\\ \qquad\qquad (sin, cos), (sin\_next, cos, false)\end{array}$$

and the initial state is:

$$(0.0, 0.0, true)$$

No recursion is needed. That is, in case a set of mutually recursive equations can be put in the normal form:

$$\begin{array}{ll}\texttt{let rec} & x_1 = v_1\ \texttt{fby}\ nx_1\\ & \texttt{and}\ ...\\ & x_n = v_n\ \texttt{fby}\ nx_n\\ & \texttt{and}\ p_1 = e_1\\ & \texttt{and}\ ...\\ & \texttt{and}\ p_k = e_k\\ \texttt{in}\ e & \end{array}$$

where

$$\forall i, j.(i < j) \Rightarrow Var(e_i) \cap Var(p_j) = \emptyset$$

where $Var(p)$ and $Var(e)$ are the set of variable names appearing in p and e. Then the semantics is $CoF(ft, s_t)$ where ft is:

$$\begin{array}{l}\lambda(x_1, ..., x_n, s_1, ..., s_k, s).let\ p_1, s_1 = [\![e_1]\!]_\rho^{Step}(s_1)\ in\\ \qquad\qquad let\ ...\ in\\ \qquad\qquad let\ p_k, s_k = [\![e_k]\!]_\rho^{Step}(s_k)\ in\\ \qquad\qquad let\ r, s = [\![e]\!]_\rho^{Step}(s)\ in\\ \qquad\qquad r, (nx_1, ..., nx_n, s_1, ..., s_k, s)\end{array}$$

and its initial state s_t is:

$$(v_1, ..., v_n, s_1, ..., s_k, s)$$

if $[\![e_i]\!]_\rho^{Init} = s_i$ and $[\![e]\!]_\rho^{Init} = s$. There is no more fixpoint computation. The proof that the semantics is preserved is immediate and obtained by a simple unfolding of the semantics.

The sequence of source-to-source transformations illustrated here results in an expression where there is no more recursion. This is the basic principle of the generation of statically scheduled code done by a synchronous language compiler. Equations are first normalized with all delays unested and the set of equations is scheduled. This static scheduling is possible, that is, the recursion is removed, only when the dependence graph between variables is acyclic. This may call for inlining some function calls and applying distribution rules.

The proof that those transformations preserve the semantics can be done by unfolding semantics definitions. This is where we can illustrate the equivalence relation $\cong$ given in Definition 1. Let $e \cong e'$ when e and e' produce equivalent concrete streams. Let $E \cong_N E'$ when both equations produce equivalent environments of concrete streams for all names in N. Equivalence proofs can be done by exhibiting an inductive relation between states associated to every expression and equations by executing the semantics. The equivalence relations below are proven by exhibiting an inductive relation R that we give for every case:

(1) E_1 and $E_2 \cong_N E_2$ and E_1. Take the relation between states R such that $\forall s1, s2.R((s_1, s_2), (s_2, s_1))$.
(2) $p = \texttt{let}\ E\ \texttt{in}\ e \cong_N E$ and $p = e$ if $Def(E) \cap Def(p) = \emptyset$. Take $R((s_1, s_2), (s_1, s_2))$.
(3) $p = \texttt{let rec}\ E\ \texttt{in}\ e \cong_N E$ and $p = e$ if $Def(E) \cap Def(p) = \emptyset$. Take $R((s_1, s_2), (s_1, s_2))$.
(4) $f(e_1, ..., \texttt{let}\ E\ \texttt{in}\ e, ..., e_n) \cong \texttt{let}\ E\ \texttt{in}\ f(e_1, ..., e, ..., e_n)$ if $Def(E) \cap Def(e_1, ..., e_n) = \emptyset$. Take $R((s_1, ..., (s_E, s_e), ..., s_n), (s_E, ..., s_e, ..., s_n))$.
(5) $f(e_1, ..., \texttt{let rec}\ E\ \texttt{in}\ e, ..., e_n) \cong \texttt{let rec}\ E\ \texttt{in}\ f(e_1, ..., e, ..., e_n)$ if $Def(E) \cap Def(e_1, ..., e_n) = \emptyset$. Take $R((s_1, ..., (s_E, s_e), ..., s_n), (s_E, ..., s_e, ..., s_n))$.
(6) $\texttt{pre}\ e \cong \texttt{let}\ x = \texttt{pre}\ e\ \texttt{in}\ x$ if $x \notin FV(e)$. Take $R((m, s), ((m, s), ()))$.
(7) $e_1\ \texttt{fby}\ e_2 \cong \texttt{let}\ x = e_1\ \texttt{fby}\ e_2\ \texttt{in}\ x$ if $x \notin FV(e_1) \cup FV(e_2)$. Take $R((m, s_1, s_2), ((m, s_1, s_2), ()))$.
(8) $(x_1, ..., x_n) = (e_1, ..., e_n) \cong_N x_1 = e_1$ and ... and $x_n = e_n$. Take $R((s_1, ..., s_n), (s_1, ..., s_n))$.

6.3 Implementation

The semantics is implemented in OCaml. The implementation takes the form of an interpreter which, given a node and a sequence of inputs, produces a sequence of outputs or stops with an error. It is implemented in the purely functional subset of OCaml; imperative features are only used for parsing, the printing of values, error messages, and automatic random testing.

This reference specification and implementation directly builds on the presented material. It provides important extensions that we discuss below.

Dealing with Errors. The presented material considers a formulation where all errors (e.g., static and dynamic errors) collapse to $\perp$. This simplification made possible the definition of the semantics that is rather short. Nonetheless, error must be added for two reasons: if the interpreter is used for testing the compiler, e.g., the front-end before all the static checks are done (typing, causality analysis, initialization analysis), it is important to signal what error is raised. The second is to formulate the correctness of compile-type verifications (e.g., "when the program type-checks, it cannot produce a type error" [27]).

One purpose of the semantics is to be used for testing during compiler development. E.g., let Tr be a source-to-source transformation such that $Tr(e)$ returns an expression e'. Let $v_e = CoF(f, s)$ where $f = [\![e]\!]^{Step}_{[]}$ and $s = [\![e]\!]^{Init}_{[]}$; and $v'_e = CoF(f', s')$ where $f' = [\![e']\!]^{Step}_{[]}$ and $s' = [\![e]\!]^{Init}_{[]}$. e and e' agree for n steps if $\forall i \leq n, nth(v_e)(i) = nth(v'_e)(i)$. When e and e' have an input (free) variable x, we can check that e and e' return two identical finite sequences for identical values for x. Different values for x can be generated randomly (or replaced by an expression which itself depend on a variable whose sequence of values is choosen randomly). This testing technique does not prove that Tr is correct but is independent of it.

Compiler testing justifies a semantics that applies directly to the source before compilation starts, and that distinguishes all the possible errors, e.g., type mismatch, unbounded identifier, causality errors, initialisation errors, clock errors, etc. We developed and tested the semantics incrementally, adding more and more constructs to the language. We started a very first version using $\perp$ to represent all errors. Then, as explained in 2.4, we used the simplest error monad (also

called *maybe* or *option* monad). [13] This solution distinguihes errors from $\bot$ but not different errors. The number of lines were almost unchanged. When the semantics was done entirely, we switched to the *result* monad to distinguish all errors [14].

It has been a surprise that all the semantics definitions were kept almost unchanged, adding in total only fifty lines of extra code. The constructive semantics is not way longer than a relational semantics. The common belief (that we though and that was also noted by Tardieu [54]) is that a constructive synchronous semantics is not the most suited for compiler proofs w.r.t a relational (logical) semantics. The constructive semantics of hierarchical automata, for example, is not longer than the relational one, e.g. [24]. It is too early to say whether proofs using the constructive semantics are shorter or better (in any kind) w.r.t proofs based on a relational semantics. For the moment, all the compiler correctness proofs of the Vélus compiler, for example, use a relational semantics.

Prototyping new language constructs. The implemented semantics treats an input language with programming constructs that are not in the paper. Our objective was to deal with the features of a real language all together, in particular those of Scade and to be able to prototype new programming constructs independently from a compiler. Two features were experimented: (1) static parameters (function arguments that are constant and can be computed during instantiation time, i.e., the computation of $[\![.]\!]_{\cdot}^{Init}$). (2) several forms of *for* loops and arrays. The state-based semantics and its OCaml implementation appeared to be well adapted to an incremental and modular development. It was also a surprise for us.

An automatically generated implementation in Coq. Several papers have addressed the formal verification of a compiler downto C code for a synchronous language reminiscent of Lustre [16, 17]. They are based on a relational semantics, not a functional one.

An implementation in Coq of the semantics presented in this paper have been developed. Instead of reprogramming it, with the risk of a mismatch w.r.t the OCaml implementation), we used `coq_of_ocaml` [15] which produces valid Coq definitions automatically. We are currently using it to prove the correctness of source-to-source transformations, e.g., in particular the ones in 6.2, and a few meta properties of the semantics, in particular monotony. It is too early to conclude.

6.4 Other Related Works

A relational semantics for a synchronous language can be described in the following manner. Given an environment R for free variables in an expression e, the predicate $R \vdash e \xrightarrow{v} e'$ states that e reacts by producing the value v and rewrites to e'. A stream-based semantics is obtained by iterating the state-based relation: if s is a sequence of values and v a value, $v.s$ is a sequence whose head is v and tail is s, with ϵ the empty sequence. An history $R.H$ is a sequence whose head is R and tail is H such that:

$$R \vdash e \xrightarrow{v} e' \qquad \frac{R \vdash e \xrightarrow{v} e' \qquad H \vdash e' : s}{R.H \vdash e : v.s} \qquad \epsilon \vdash e : \epsilon$$

The strength of a relational synchronous semantics is the elegant (and a bit magical) way synchronous composition is expressed: if E_1 and E_2 are two equations that run in parallel synchronously,

[13] That is, a concrete stream is a pair: $S \times (S \to 1 + (V \times S))$ meaning that, given a state $s : S$, the transition function either stops (when an error occurs) or returns a value $v : V$ and a new state $s' : S'$.
[14] The type *option* (https://v2.ocaml.org/api/Option.html) is replaced by the type *result* (https://v2.ocaml.org/api/Result.html).
[15] https://github.com/formal-land/coq-of-ocaml

the reaction and execution predicate are simply:

$$\frac{R(x) = v \qquad R \vdash e \xrightarrow{v} e'}{R \vdash x = e \xrightarrow{[v/x]} x = e'} \qquad \frac{R \vdash E_1 \xrightarrow{R_1} E_1' \qquad R \vdash E_2 \xrightarrow{R_2} E_2'}{R \vdash E_1 \text{ and } E_2 \xrightarrow{R_1+R_2} E_1' \text{ and } E_2'} \qquad \frac{R \vdash E \xrightarrow{R'} E' \qquad H \vdash E : H'}{R.H \vdash E : R'.H'}$$

$$\epsilon \vdash E : \epsilon$$

with the invariant that $R \vdash E \xrightarrow{R'} E'$ with $R' \subseteq R$, meaning that E instantaneously sees what is emitted with no added delay. Hence, if E_1 and E_2 react by producing R_1 and R_2 respectively, their parallel composition reacts by producing the composition $R_1 + R_2$. The implicit side condition is that the domains R_1 and R_2 do not overlap and thus $R_1 \subseteq R$ and $R_2 \subseteq R$ implies $R_1 + R_2 \subseteq R$. Thus, E_1 instantaneously sees what is defined by E_2. A relational semantics is not an effective (correct and complete) algorithm that is constructive. Nothing says when a reaction exists, if it is unique and how to construct it. Moreover, some programs do have a semantics that is unique but they are unreasonable and counter intuitive. For example, the node `non_deterministic`, if given a relational semantics, can either returns `true` or `false` (whereas its output is $\bot$ with a constructive semantics). The node `deadlock` has a deadlock because there is no value such that $a = a + 1$. The `hamlet` node that uses an `or` gate has a unique solution (`tobe = true`). With the constructive fix-point semantics, it deadlocks, that is, `tobe` = $\bot$, if `or` is interpreted strictly. It also deadlocks if `or` is interpreted in three valued logic, like in Esterel. Finally, if `or` is encoded with a conditional and we take the interpretation $^\sharp if\,.\,then\,.\,else\,.$, it also deadlocks: starting with `tobe` = $\bot$, $not(tobe) = \bot$ hence $^\sharp if\ \bot\ then^\star{=}(\bot, \bot)\ else\ \bot = \bot$. On the contrary, the cyclic circuit presented by Malik in [41] has no deadlock; it is causally correct.

```
let node non_deterministic() returns (o)
  local a, b do a = b or b and b = a and o = a done

let node hamlet() returns (tobe)
  do tobe = tobe or not (tobe) done

let node hamlet() returns (tobe)
  do tobe = if tobe then true else not (tobe)

(* The constructive cyclic synchronous circuit of Malik *)
(* computes [y = if c then g(f(x)) else f(g(x))] *)
let node malik(c, x) returns (y)
  local x1, x2, y1, y2 in
  do x1 =  if c then x else y2
  and x2 =  if c then y1 else x
  and y1 =  f(x1) and y2 =  g(x2)
  and y =  if c then y2 else y1 done
```

6.4.1 Relations to Arrows and FRP. The algebraic properties of synchronous (length-preserving) stream functions have been studied by [35] who identifies them as an instance of so-called *arrows*. The latter form a categorical structure, more general than *monads*, with operations for stream composition >>>, pairing &&&, product ***, bypass `first` and `second`, extensible by multiplexing |||, choice +++ and feedback `loop`. This combinatory (i.e., variable-free) syntax is not meant for practical programming. Like for monads, there is an equivalent syntax for arrows introduced

by [46] which comes closer to the equational definitions used here. The arrow abstraction is used in the functional reactive language Yampa [34] implemented as a domain-specific library in Haskell. However, Yampa's combinator syntax is more restrictive compared to the syntax available in a synchronous language. Moreover, it cannot guarantee the same safety properties — bounded memory and absence of deadlock — because it gives the programmer unlimited access to the ambient higher-order lazy language Haskell.

In [40], authors observe that the composition of length preserving functions can be simplified into a normal form where all delays gathered. The resulting body can then be expressed in a state-space representation:

$$o = f(s, i, o) \qquad s = s_0 \texttt{ fby } g(s, i, o)$$

where i is the current input, o the current output and s the internal state initialized with value s_0. The transformation made by a synchronous compiler [14] does the same transformation but with the supplementary constraint that o must not depend on o and so $o = f(s, i)$ only. In the target code is Haskell, this extra constraint can be removed. The output of the transition function f can be computed lazilly, as noted in [23].

$$\mathit{feedback}\,(\mathit{ft}) = \lambda s.\mathit{let\ rec}\, v, s' = \mathit{ft}\, s\, v\ \mathit{in}\,(v, s')$$

No static scheduling is necessary nor an interative computation for v. This approach is useful for defining an interpreter. [16] Without a static analysis done a priori, the interpreter may deadlock. Because a synchronous language targets real-time applications, a compiler generates statically scheduled code, hence that does not need lazy evaluation.

fix f s = let rec vs' = lazy (f s Some work explain the essence of data-flow languages and in particular synchronous languages via Co-Kleisli functions for co-monads [55] rather than as Kleisli functions for monads. A monadic model of synchronous *imperative streams* inside Haskell is proposed in [53]. However, these monadic "streams" represent behavior on finite time episodes and permit arbitrary side effects in the IO monad. This makes imperative streams even less useful for safety-critical applications than Yampa. Our executable semantics hedges the risk and safely resides within the limits of simply-typed lambda-calulus.

[16] In the definition above, we consider a meta language with call-by-need evaluation.