Clocks in Kahn Process Networks *

Marc Pouzet
ENS, Département d'informatique, 45 rue d'Ulm, 75230 Paris, France
Marc.Pouzet@ens.fr

October 12, 2021

Abstract

The language Lustre was introduced to design and implement real-time control software, modeling it as a *continuous function* over treams of datas. A set of equations written in Lustre defines a restricted class of Kahn process networks which can be executed synchronously: all computations can be dated according to a global time scale so that when a value is produced, it is immediately consumed. This restriction is obtained by associating to every stream a *clock* that defines when a value is present or not according to a global time scale. A dedicated type system — the clock calculus — computes a clock for every expression and checks that its actual clock equals its expected clock and thus that intermediate buffers are not needed.

In these course notes,[1] we present a static and dynamic semantics of synchronous Kahn networks. We consider a first-order functional language of streams reminiscent of Lustre and Lucid Synchrone to which we give several denotational semantics. We show that without imposing restrictions, we get two kinds of bad behavior: some networks may deadlock and some cannot execute without unbounded FIFOs. We introduce a clocked semantics and show that the clocking rules correspond to a type system with dependent types. We then extend the language kernel with an explicit `buffer` operator to model communication through a FIFO. The clock calculus is extended with a subtyping rule that is applied where the buffer is used and whose size is inferred. To reduce the complexity of the resolution, we present an abstraction of clocks.

1 Introduction

Synchronous languages [3] were introduced about thirty years ago by the concurrent work on three academic languages: Signal [5], Esterel [7] and Lustre [20]. These *domain specific languages* targeted real-time control software, allowing to write modular and mathematically precise system specifications, to simulate, test and verify them, and to automatically translate them into embedded executable code. The environment SCADE,[2] based on a synchronous language [15], is now used routinely to develop various critical control software: in planes (fly-by-wire, engine control, emergency braking), trains (on-board control, interlocking), etc.

All these languages are founded on the synchronous model of time [6] where a system is modeled ideally, with communications and computations assumed to be instantaneous, with formal checks of important safety properties like determinism, deadlock freedom, execution in bounded time and space, and with a posteriori verification that a given implementation in software or hardware executes quickly enough.

Lustre is a data-flow language: it manipulates infinite streams of data that represent the evolution of an input, an output or a local variable, streams are defined by writing mutually

*The title refers to the article [10] by Paul Caspi.

[1]The present notes are based on one of the lectures on synchronous programming given by the author at the Marktoberdorf summer school in August 2018. It includes previous works by Louis Mandel, Florence Plateau and the author.

[2]http://www.esterel-technologies.com/products/scade-suite

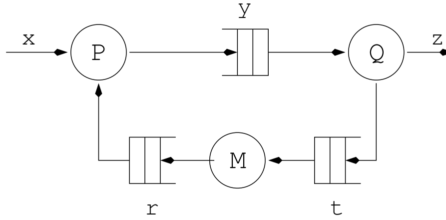

Figure 1: A Kahn Process Network with three processes

recursive equations over them, and a system is a function from streams to streams. Time is simply the index in a stream. After passing static checks, a stream function is compiled to sequential code (typically C). It can also serve as a functional model of a device or software for the purposes of formal verification ([19] summarises the different uses of Lustre).

A set of stream equations written in Lustre can be interpreted as a *Kahn Process Network* [21]: stream functions are the nodes, every stream defines a communication channel and a set of equations corresponds to a process network. Lustre is Kahnian because a stream function cannot dynamically test whether a signal is present or absent. The consequence is that all execution strategies for a network are guaranteed to compute the same set of streams. Nonetheless, as Lustre targets real-time applications, a function written in Lustre defines a particular subset for which the compiler ensures that it does not deadlock and can be compiled into statically scheduled code running in bounded time and space. This is achieved by imposing a set of static constraints to ensure that the network can be executed synchronously, that is, every computation in the network must be dated according to a global time scale so that when a value is produced, it can be immediately consumed. Hence, no intermediate buffers are needed. This synchronous interpretation is obtained by associating to every stream a *clock* that defines when a value is present or not according to a global time scale. Clocks may or may not be periodic and may depend on input values. A dedicated type system — the clock calculus — computes a clock for every expression and checks that it matches the expected clock.

In this text, we describe a static and dynamic semantics for Lustre from the perspective of Kahn process networks. We consider a simple first-order language of streams reminiscent of Lustre and Lucid Synchrone to which we give several denotational semantics. We show that the naive encoding of streams as lazy data structures gives rise to strange non-causal behaviors, highlighting the need for the prefix order introduced by Kahn. We then give a Kahn semantics to the language kernel. To account for the synchronous restriction, we introduce a clocked semantics and show that the clocking rules that a program must fulfill correspond to typing constraints in a type system with dependent types. We derive a simpler type system which reduces the equality of clocks to name equality. We then extend the language kernel with an explicit buffer operator to model communications via FIFOs. The clock calculus is extended with a subtyping rule that is applied where the buffer is used and whose size is inferred. To reduce the complexity of the resolution, we present an abstraction of clocks.

1.1 Kahn Process Networks

In the 1970s, Kahn studied the semantics of networks of deterministic parallel processes communicating asynchronously through FIFO channels. One may think, e.g., of a set of Unix processes communicating via pipes or of threads running asynchronously and synchronising through bounded FIFO queues. Kahn showed that in the case where elementary processes are deterministic, with blocking read on an empty channel and non-blocking writes, the overall network is deterministic — the result does not depend on the relative order in which nodes are activated — and delay insensitive — computation and communication times do not change the network semantics [21, 22]. In short, the model is one of the very few that conciliates parallelism and determinism. In a Kahn

network, a basic process can be programmed in a sequential language with two primitives: `push` to write a value to a channel and `pop` to read a value. Figure 1 depicts a network with three processes.[3] There is a single reader and writer per channel. A process may only read a single channel at a time and, once committed to reading, it must wait until a value is available. It may not test the channel for emptiness or impose a timeout; that is, it cannot test whether a value is present or absent. One cannot write, for instance:[4]

```
if is_empty a then ...    or    if not (is_empty a) or not (is_empty b) then ...
```

But, it is possible to conditionally read or write according to a value that has been read from a channel, e.g.:

```
let v = pop c in let w = if cond a then pop a else pop b in ...
```

or

```
let v = pop a in if cond v then push a (f v)
```

Figure 2 gives a few examples of elementary primitives: `lift2 f x y z` applies a function `f` pointwise to its two input channels `x` and `y`, and produces an output on channel `z`; the unit delay `fby x y z` concatenates the first element of its input channel `x` to the elements of its second input channel `y` and writes on channel `z`; `merge c x y z` conditionally reads an input channel `x` or `y` according to the value on channel `c` and writes on channel `z`; `split c y z` conditionally writes on channel `y` or `z` according to the value on channel `c`.

Kahn networks with bounded buffers can be implemented by adding a *back pressure* mechanism in order to avoid writes into a full buffer. Nonetheless, this may introduce artificial blocking if the size of buffers are underestimated. The size of buffers can be increased dynamically [28] but this solution cannot be used for real-time applications where overall memory use must be guaranteed at compile time.

Whether or not a Kahn network is deadlock free or can be executed in bounded memory is undecidable in general [9]. *Synchronous Data Flow* (or SDF) [24] and its variants (*Cyclo Static Data Flow* [27] among others) are restricted classes of networks where every node consumes and produces a fixed number of tokens at every step. The size of buffers can be computed at compile time and a periodic static schedule can be generated. This make SDF suitable for modeling and programming video intensive applications with periodic behavior [32].

To prove that determinism is preserved by composition, Kahn took an approach based on denotational semantics using the following interpretation of channels and processes. A communication channel that carries values of type T is interpreted as a (possibly infinite) sequence of values of type T that describe the *history* of values on the channel. Because a node has its own internal memory, it is interpreted as a function from the histories of its inputs to the histories of its outputs, that is, a stream function. We now recall a few basic properties of sequences, cpos and continuous functions.

1.1.1 Sequences and Continuous Functions over Sequences

Consider a set T of values. T^n denotes the set of sequences of length n made by concatenating elements from T. The sequence $v.s$ comprises head v and tail s. The empty sequence is written ϵ. The set of finite sequences is written $T^\star = \cup_{n=0}^{\infty} T^n$. The set of finite and infinite sequences of elements of T is written $T^\infty = T^* \cup T^\omega$. We write $\leq$ for the prefix order over sequences; $s \leq s'$ means that s is a prefix of s'. For any s, s', $\epsilon \leq s$ and if $s \leq s'$, then $v.s \leq v.s'$. A chain in T^∞ is any non-empty subset that is totally ordered by $\leq$. $(T^\infty, \leq, \epsilon)$ is a complete partial order (CPO): ϵ is its minimum element for the partial order $\leq$ and every chain has a least upper bound. In the case of boolean sequences, where $\mathtt{0}$ stands for *false* and $\mathtt{1}$ for *true*, $\epsilon \leq \mathtt{0} \leq \mathtt{0.1} \leq \mathtt{0.1.0} \leq \mathtt{0.1.0.0}$ but not $\mathtt{1.1} \leq \mathtt{1.0}$.

[3] Figure 16 in Appendix A gives an implementation with threads.
[4] The concrete syntax is that of OCaml.

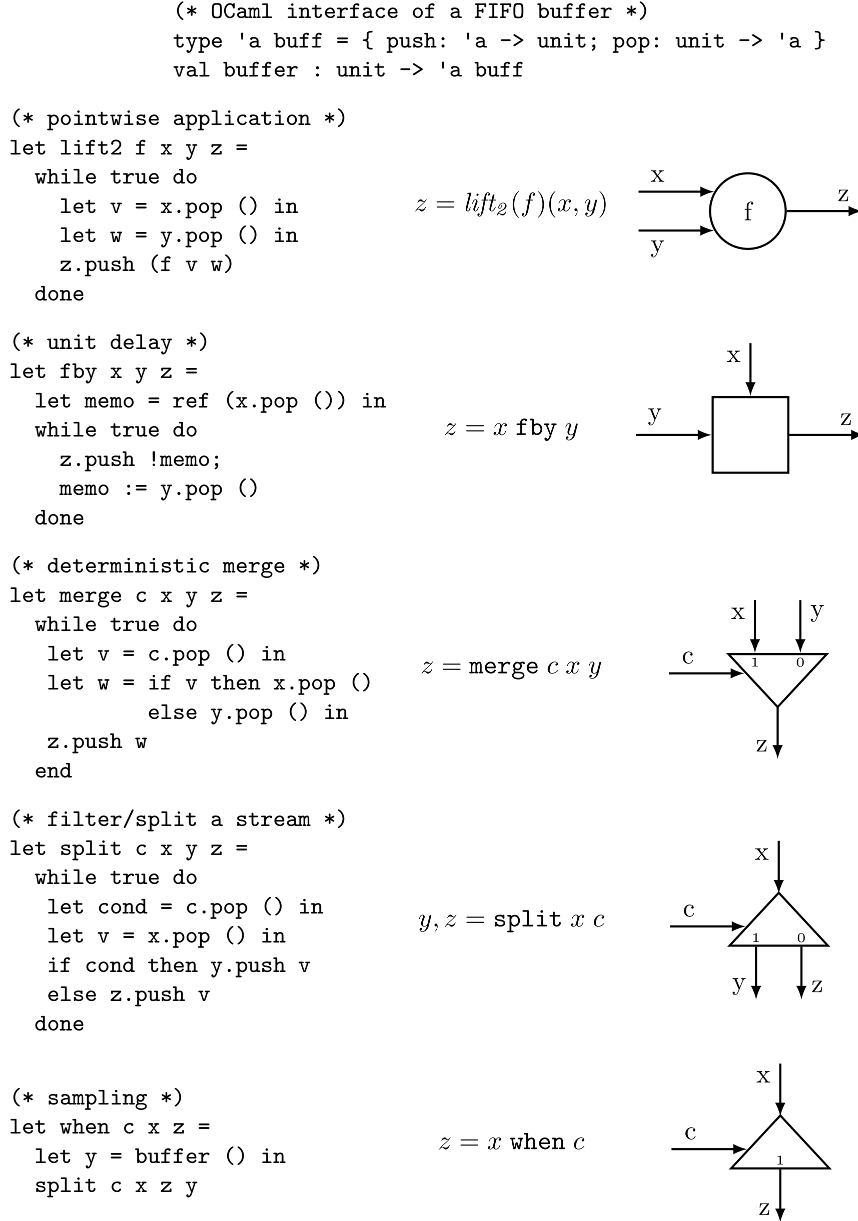

Figure 2: A set of data-flow primitives

x	x_0	x_1	x_2	x_3	x_4	x_5
y	y_0	y_1	y_2	y_3	y_4	y_5
$x + y$	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	$x_5 + y_5$
x fby y	x_0	y_0	y_1	y_2	y_3	y_4
h	1	0	1	0	1	0
$x' = x$ when h	x_0		x_2		x_4	
z		z_0		z_1		z_2
merge $h\ x'\ z$	x_0	z_0	x_2	z_1	x_4	z_2

Figure 3: A set of primitives interpreted as stream functions

In the sequel, we shall sometimes write a sequence in a more traditional way. A sequence $u = (u_i)_{i \in I}$, finite or not, is a set indexed by an initial segment I of $\mathbb{N}$. $I \subseteq \mathbb{N}$ is an initial segment when $\forall n, m \in \mathbb{N}. (n \in I) \wedge (m \leq n) \Rightarrow (m \in I)$.

For any subset A of $\mathbb{N}$, there exists a strictly increasing, one-to-one function ϕ_A between an initial segment I_A of $\mathbb{N}$ and A. An operation that builds a sub-sequence from a sequence by picking a subset of indices or merges two sequences to build another one corresponds to defining particular ϕ functions. This picking does not have to be periodic, as in $(u_{2i})_{i \in \mathbb{N}}$ that is made by taking one element of u every two. It can depend on the value of streams. We shall see concrete examples in the next section.

General properties of a CPO If $D_1 = (A_1, \leq_1, \perp_1)$ and $D_2 = (A_2, \leq_2, \perp_2)$ are two cpos, with respective minimum elements $\perp_1$ and $\perp_2$, a function $f : D_1 \rightarrow D_2$ is monotonic if and only if for any $x, x' \in D_1$, $x \leq_1 x' \Rightarrow f(x) \leq_2 f(x')$. It is continuous if and only if for any chain C in D_1, $f(sup(C)) = sup(\{f(d), d \in C\})$. Any continuous function $f : D \rightarrow D$ on a CPO $D = (A, \leq, \perp)$ has a least fix point $\text{fix}\,(f) = lim_{n \rightarrow \infty}(f^n(\perp))$, with $f^0(x) = x$ and $f^{n+1}(x) = f(f^n(x))$ (Kleene theorem).

If A_1 and A_2 are CPOs, then $(D_1 \times D_2, \leq', \perp')$ is also a CPO, with $D_1 \times D_2$ being the set of pairs (x_1, x_2) comprising an element x_1 from D_1 and an element x_2 from D_2, taking $\perp' = (\perp_1, \perp_2)$ as the minimum element and $\leq'$ such that $(x_1, x_2) \leq' (y_1, y_2) \Leftrightarrow (x_1 \leq_1 y_1) \wedge (x_2 \leq_2 y_2)$. The set $D = (D_1 \mapsto D_2, \leq', \perp')$ where $D_1 \mapsto D_2$ is the set of total continuous functions from D_1 to D_2, with $f \leq' g \Leftrightarrow \forall s \in D_1. f(s) \leq_2 g(s)$ and $\perp' = (\lambda s. \perp_2)$ is the minimum element, is also a CPO.

1.1.2 Application to Kahn Process Networks

Following the formulation in [21], a network is represented by a set of equations built according to the two following rules:

- If $x_1, ..., x_k$ are the input channels of the network fed with the sequences $i_1, ..., i_k$, add the equations
$$\{x_1 = i_1, \ldots, x_n = i_n\}$$
- Interpret every node f with n input channels $x_1, ..., x_n$ and p output channels $x'_1, ..., x'_p$ as p continuous functions over sequences and add the equations
$$\left\{y'_1 = f_1(x_1, ..., x_n), \ldots, y'_p = f_p(x_1, ..., x_n)\right\}$$

The example in Figure 1 is represented by the following set of equations, if p is the stream function associated to process P; $\langle q_1, q_2 \rangle$ is associated to Q; m to M:

$$\{y = p(x, r), z = q_1(y), t = q_2(y), r = m(t)\}$$

Elementary nodes in the network are interpreted as *continuous functions* over sequences. Monotonicity corresponds to the intuition that as a process reads more inputs, it can only produce more outputs: it cannot contradict what has already been produced.

Since every node in a Kahn process network is a continuous function, a set of equations:

$$\{x_1 = f_1(x_1, ..., x_n), ..., x_n = f_n(x_1, ..., x_n)\}$$

has a minimal solution which is $x_1^\infty, ..., x_n^\infty = lim_{j \rightarrow \infty}(x_1^j, ..., x_n^j)$ where for all $1 \leq i \leq n$, $x_i^0 = \epsilon$ and $x_i^{j+1} = f_i(x_1^j, ..., x_n^j)$.

The primitives given in Figure 2 [5] can be interpreted as stream functions as illustrated in Figure 3. An important consequence of the interpretation of elementary nodes as continuous functions is that any composition, where some variable may be made local, still defines a continuous function. For the network in Figure 1, if the channels y, r and t are considered to be local, the network can be interpreted as a continuous function f of the input x, such that the output z satisfies $z = f(x)$.

[5] The operator `when` can also be programmed directly by removing the `else` branch of a `split`. This operator is itself a composition of two `when`.

d	::=	$\mid$ `let node` f $pat = e$	node definition
		$\mid$ `let clock` $c = ce$	clock definition
		$\mid$ d d	sequence of definitions
pat	::=	x $\mid$ $(pat, \ldots, pat)$	pattern
e	::=	$\mid$ i	constant flow
		$\mid$ x	flow variable
		$\mid$ $(e, \ldots, e)$	tuple
		$\mid$ $\texttt{get}_i(e)$	i-th component of a tuple
		$\mid$ e op e	imported operator
		$\mid$ `if` e `then` e `else` e	mux operator
		$\mid$ f e	node application
		$\mid$ e `where rec` eqs	local definitions
		$\mid$ e `fby` e	initialized delay
		$\mid$ e `when` ce $\mid$ e `whenot` ce	sampling
		$\mid$ `merge` ce e e	merging
		$\mid$ `buffer` e	buffering
eqs	::=	$pat = e$ $\mid$ eqs `and` eqs	mutually recursive equations
ce	::=	e	clock expressions

Figure 4: Language kernel.

2 A Language of Streams and Stream Functions

We consider a first-order synchronous dataflow language reminiscent of Lustre and Lucid Synchrone but extended with an explicit buffering operator. The syntax is given in Figure 4. A program (d) is a sequence of definitions of stream functions called *nodes* and definitions of clock names (c). The inputs of a node are described by a pattern (pat) and its body by an expression (e). The operators are the basic ones of Lucid Synchrone and their intuitive semantics is detailed later. The expression e_1 op e_2 denotes the pointwise application of a binary operator; `if` e_1 `then` e_2 `else` e_3 is the pointwise application of a conditional; $f\,e$ is the application of a node f to an expression e; e_1 `fby` e_2 conses the head of e_1 to e_2 and thus corresponds to an initialized delay; e `when` ce samples a stream e according to a boolean expression ce (`whenot` samples when the expression is `0`). We call this boolean expression a clock. The operator `merge` ce e_1 e_2 merges two streams according to a clock. Finally `buffer` e buffers e. We write e `where rec` eqs for an expression defined by a collection of mutually recursive equations (eqs). The basic data-flow primitives of this language kernel are those of Figure 2. For the language of clocks ce, we take any boolean expression. We shall later consider particular cases of this language.

2.1 Denotational Semantics

We first give a denotational semantics based on possibly infinite sequences, following the interpretation given by Kahn. In this setting, the operator `buffer` is simply the identity function. We then define a synchronous semantics which characterises the evolution of the streams and the contents of the buffers.

Notation for the semantics. We write ρ for an environment and $\rho(x)$ for the value associated to the variable x in the environment ρ. The environment $\rho + [x \leftarrow v]$ is the environment ρ to which has been added the binding of x to v. The environment $\rho + \rho'$ is the environment that contains the associations of the environment ρ and the associations of the environment ρ' provided that no single variable appears in both ρ and ρ'.

$$
\begin{array}{lcl}
op^\sharp(v_1.s_1, v_2.s_2) & = & v.op^\sharp(s_1, s_2) \text{ where } v = op(v_1, v_2) \\
\texttt{fby}^\sharp(v_1.s_1, s_2) & = & v_1.s_2 \\
\texttt{when}^\sharp(v_1.s_1, \texttt{1}.w) & = & v_1.\texttt{when}^\sharp(s_1, w) \\
\texttt{when}^\sharp(v_1.s_1, \texttt{0}.w) & = & \texttt{when}^\sharp(s_1, w) \\
\texttt{whenot}^\sharp(v_1.s_1, \texttt{0}.w) & = & v_1.\texttt{whenot}^\sharp(s_1, w) \\
\texttt{whenot}^\sharp(v_1.s_1, \texttt{1}.w) & = & \texttt{whenot}^\sharp(s_1, w) \\
\texttt{merge}^\sharp(\texttt{1}.w, v_1.s_1, s_2) & = & v_1.\texttt{merge}^\sharp(w, s_1, s_2) \\
\texttt{merge}^\sharp(\texttt{0}.w, s_1, v_2.s_2) & = & v_2.\texttt{merge}^\sharp(w, s_1, s_2)
\end{array}
$$

Figure 5: Then Kahn semantics for the primitives

The interpretation of an expression e in an environment ρ is written $[\![e]\!]_\rho$. This notation will also be used for the denotation of equations and declarations.

Finally, when presenting the interpretation of the primitives as stream functions, we shall use the notation $\sharp$ as an exponent to distinguish syntactic constructs from their interpretations.

2.1.1 A Kahn semantics

Every node declaration `let node` $f\ pat = e$ is interpreted as a continuous function over sequences. The Kahn semantics for the primitives of the language is given in Figure 5.

- op applies an imported operator pointwise on scalar values;
- `fby` is the unit delay; it appends the head of its first argument onto the value of its second argument;[6]
- `when` is the sampling operator: it passes its input to its output only if the condition is true (value `1`) and otherwise does not produce any output;
- `whenot` is the complementary sampling operator which passes its input to its output only if the second input is false (value `0`) and otherwise does not produce any output;
- `merge` merges two input streams according to a boolean condition. It passes its first input to its output when the boolean condition is true and the second input otherwise.

These definitions must be completed to deal with the empty sequence ϵ. All operators return ϵ if one of their arguments is the empty sequence (ϵ is absorbing), except for the operator `fby` which is such that $\texttt{fby}^\sharp(v.s, \epsilon) = v.\epsilon$. We also have to deal with possible type errors. Several solutions can be taken: (1) complete all the definitions by returning ϵ in case of a type error; (2) add a special `TypeError` value to the set of streams and transmit this value; (3) define the semantics for well-typed expressions only. For the sake of simplicity, we apply the first solution.

All the primitives are monotonic and continuous [10].

The semantics of expressions of the language is defined in Figure 6. The definition uses the interpretations of the primitives given previously. We write $[\![e]\!]_\rho$ to denote the value of e in the environment ρ. We ensure that the language is first order by using two distinct namespaces: one that maps local variables to (stream) values or tuples of values, and a second that maps global variables to functions. The environment ρ is thus a pair (ρ_s, ρ_n) where ρ_s associates a value to every free variable of e and ρ_n associates a value to every function. Letting Var_s denote the set of variable names and Var_n the set of node names, we have

$$
\begin{array}{lcll}
Stream(T) & = & T^\infty & \text{sequences} \\
V & = & Stream(T_1) + \cdots + Stream(T_n) + V \times \cdots \times V & \text{values for local variables} \\
\rho_s & : & Var_s \to V & \text{local environment} \\
\rho_n & : & Var_n \to (V \to V) & \text{global environment}
\end{array}
$$

[6] It corresponds to the `A` operator of [21]. The `fby` operator was introduced in Lucid [2] and used in [10].

$$
\begin{array}{lcl}
[\![i]\!]_\rho & = & i.[\![i]\!]_\rho \\
[\![x]\!]_\rho & = & \rho_s(x) \\
[\![(e_1, \ldots, e_n)]\!]_\rho & = & ([\![e_1]\!]_\rho, \ldots, [\![e_n]\!]_\rho) \\
[\![\mathtt{get}_i(e)]\!]_\rho & = & s_i \text{ if } [\![e]\!]_\rho = (s_1, \ldots, s_n) \\
[\![e_1\ op\ e_2]\!]_\rho & = & op^\sharp([\![e_1]\!]_\rho, [\![e_2]\!]_\rho) \\
[\![f\ e]\!]_\rho & = & \rho_n(f)\ [\![e]\!]_\rho \\
[\![e \texttt{ where rec } eqs]\!]_\rho & = & [\![e]\!]_{\rho+\rho'} \text{ where } \rho' = ([\![eqs]\!]_\rho, \emptyset) \\
[\![e_1 \texttt{ fby } e_2]\!]_\rho & = & \mathtt{fby}^\sharp([\![e_1]\!]_\rho, [\![e_2]\!]_\rho) \\
[\![e \texttt{ when } ce]\!]_\rho & = & \mathtt{when}^\sharp([\![e]\!]_\rho, [\![ce]\!]^{\mathsf{ce}}_\rho) \\
[\![e \texttt{ whenot } ce]\!]_\rho & = & \mathtt{whenot}^\sharp([\![e]\!]_\rho, [\![ce]\!]^{\mathsf{ce}}_\rho) \\
[\![\texttt{merge } ce\ e_1\ e_2]\!]_\rho & = & \mathtt{merge}^\sharp([\![ce]\!]^{\mathsf{ce}}_\rho, [\![e_1]\!]_\rho, [\![e_2]\!]_\rho) \\
[\![e]\!]^{\mathsf{ce}}_\rho & = & [\![e]\!]_\rho \\
[\![\texttt{buffer } e]\!]_\rho & = & [\![e]\!]_\rho
\end{array}
$$

Figure 6: The Kahn semantics for the language expressions

If $\rho = (\rho_s, \rho_n)$ and $\rho' = (\rho'_s, \rho'_n)$ then $\rho + \rho' = (\rho_s + \rho'_s, \rho_n + \rho'_n)$. We write $\rho + [z \leftarrow v]$ to add the association $z \leftarrow v$ in the appropriate part of the pair ρ.

The interpretation of e `where rec` *eqs* uses the interpretation of the set of equations *eqs* as the supplementary environment. If *eqs* is $x_1 = e_1$ `and` $\cdots$ `and` $x_k = e_k$, its interpretation is an environment that associates every variable x_i with the interpretation of e_i:

$$
\begin{array}{c}
[\![x_1 = e_1 \texttt{ and } \cdots \texttt{ and } x_k = e_k]\!]_\rho = [x_1 \leftarrow {x_1}^\sharp, \ldots, x_k \leftarrow {x_k}^\sharp] \\
\text{where } {x_1}^\sharp, \ldots, {x_k}^\sharp = \text{fix}\left(\lambda s_1, \ldots, s_k.\ [\![e_1]\!]_{\rho+[x_1 \leftarrow s_1, \ldots, x_k \leftarrow s_k]}, \ldots, [\![e_k]\!]_{\rho+[x_1 \leftarrow s_1, \ldots, x_k \leftarrow s_k]}\right)
\end{array}
$$

The interpretation of the operators `when`, `whenot` and `merge` uses the interpretation of their clock argument *ce*. In this basic language, we consider that a clock expression can be any boolean expression, hence $[\![ce]\!]_\rho = [\![e]\!]_\rho$. In the second part of these notes, we introduce a dedicated sublanguage of boolean expressions.

The operation `buffer` copies its input into its output, possibly delaying it. Since the Kahn semantics is unable to express timing, the interpretation here is simply the identity function.

The semantics of a program is defined as follows:

$$
\begin{array}{lcl}
[\![\texttt{let node } f\ x = e]\!]_\rho & = & \rho + [f \leftarrow (\lambda s.\ [\![e]\!]_{\rho+[x \leftarrow s]})] \\
[\![\texttt{let clock } c = ce]\!]_\rho & = & \rho + [c \leftarrow [\![ce]\!]^{\mathsf{ce}}_\rho] \\
[\![d_1\ d_2]\!]_\rho & = & [\![d_2]\!]_{\rho+\rho_1} \text{ where } \rho_1 = [\![d_1]\!]_\rho
\end{array}
$$

The evaluation of a program d having f as the main node in an environment where the input stream is I is defined by:

$$\rho_n(f)\ I \text{ where } (\rho_s, \rho_n) = [\![d]\!]_{(\emptyset,\emptyset)}$$

2.1.2 Reformulating the Kahn semantics using sequences

The semantics can also be formulated using the notation $(u_i)_{i \in N}$ with $N \subseteq \mathbb{N}$ being an initial section of $\mathbb{N}$, that is, a sequence. If $A \subseteq \mathbb{N}$, there exists a one-to-one function ϕ_A between an initial section I_A and A.

$$
\begin{array}{lcll}
\mathit{lift_0}^\sharp(v) & = & (u)_{n\in\mathbb{N}} & \mathit{with} \quad \forall n \in \mathbb{N}.\, u_n = v \\
\mathit{lift_1}^\sharp(\mathit{op})((u_n)_{n\in N}) & = & (v_n)_{n\in N} & \mathit{with} \quad \forall n \in N.\, v_n = \mathit{op}(u_n) \\
\mathit{lift_2}^\sharp(\mathit{op})((u_n)_{n\in N}, (v_n)_{n\in N}) & = & (w_n)_{n\in N} & \mathit{with} \quad \forall n \in N.\, w_n = \mathit{op}(u_n, v_n) \\
\mathtt{fby}^\sharp((u_n)_{n\in N}, (v_n)_{n\in N}) & = & (w_n)_{n\in N} & \mathit{with} \quad w_0 = u_0 \\
 & & & \mathit{and} \quad \forall n \in N\backslash\{0\}.\, w_n = v_{n-1}
\end{array}
$$

If $(h_n)_{n\in N}$ is a boolean sequence, define N_h and $N_{\overline{h}}$ as a partition of N:

$$
N_h = \{k \in N \mid h_k = \mathtt{1}\} \quad \text{and} \quad N_{\overline{h}} = \{k \in N \mid h_k = \mathtt{0}\}
$$

The filter operator `when` and complement `merge` are defined in the following way:

$$
\begin{array}{lcll}
\mathtt{when}^\sharp((u_n)_{n\in N}, (h_n)_{n\in N}) & = & (v_n)_{n\in I_{N_h}} & \mathit{with} \quad v_n = u_{\phi_{N_h}(n)} \\
\mathtt{merge}^\sharp((h_n)_{n\in N}, (u_n)_{n\in I_{N_h}}, (v_n)_{n\in I_{N_{\overline{h}}}}) & = & (w_n)_{n\in N} & \mathit{with} \quad w_n = u_n \ \mathit{if}\, n \in N_h \\
 & & & \mathit{and} \quad w_n = v_n \ \mathit{if}\, n \in N_{\overline{h}}
\end{array}
$$

A set of equations over sequences becomes a set of mutually recursive functions, from natural numbers to values. Figure 7 gives a possible implementation in OCaml.

We use the functions `index` and `cumul` which are, respectively, the index and cumulative functions, written I and O in [25]. If h is a boolean stream, $O(h)(n)$ is the sum of 1s up to index n; $I(h)(n)$ is the index of the nth 1 in h.

$$
O(h)(n) = \sum_{i=0}^{n} h(i) \quad I(h)(n) = \min\left(\{k \in \mathbb{N} \mid O_h(k) = n\}\right)
$$

This implementation, however, only addresses sequences that are total over $\mathbb{N}$. Trying to compute `index(x when (constant false))` for any n results in a stack overflow. This is no surprise since the domain of `when` can be finite. We shall see later how this problem can be addressed.

Moreover, even in the case where all sequences are infinite, the implementation is extremely inefficient. While it is useful for reasoning about programs, it is not a practical implementation. Indeed, the value of a sequence x at instant n is computed recursively, possibly back to index 0, with no reuse of previously computed values. It is possible, though, to program the initial Kahn semantics almost directly using infinite data structures and lazy evaluation.

2.1.3 An Implementation in Haskell with Lazy Data-structures

The definitions in Figure 5 can be implemented with potentially infinite data structures and lazy evaluation. Figure 8 gives an implementation in Haskell. For example,

```
plusl x y = lift2 (+) x y
minusl x y = lift2 (-) x y

-- integers greaters than n
from n =
  let nat = n `fby` (plusl nat (constant 1)) in nat

-- a resettable counter
reset_counter res input =
  let output = ifthenelse res (constant 0) v
      v = ifthenelse input
                     (pre 0 (plusl output (constant 1)))
                     (pre 0 output)
  in output
```

```
(* signal as sequences, i.e., functions from natural numbers to values *)
type 'a sequence = int -> 'a

let const v n = v
let extend f x n = (f(n)) (x(n))

let notl x n = extend (const (fun x -> not x)) x n
let plusl x y n = extend (extend (const (fun x y -> x + y)) x) y n
let andl x y n = extend (extend (const (fun x y -> x && y)) x) y n

let pre v x n = if n = 0 then v else x(n-1)
let fby x y n = if n = 0 then x 0 else y(n-1)

(* cumulative and index functions *)
let rec cumul(h)(n) = (if h(n) then 1 else 0) +
                      (if n = 0 then 0 else cumul(h)(n-1))

let rec index(h)(n) = index_aux(h)(0)(n)
and index_aux(h)(i)(n) =
      if h(i) then if n = 1 then i else index_aux(h)(i+1)(n-1)
      else index_aux(h)(i+1)(n)

(* filtering and merge *)
let whenc x h n = x(index(h)(n+1))
let merge h x y n = if h(n) then x(cumul(h)(n)) else y(cumul(notl h)(n))

(* Fixpoint operator *)
let fix : ('a sequence -> 'a sequence) -> 'a sequence =
    fun n -> let rec o n = f o n in o n

(* Examples *)
let half () = let rec half n = pre false (notl half) n in half

let from v =
  let rec f n = pre 0 (plusl f (const 1)) n in f

let incr v x n = pre v (plusl x (const 1)) n
let from v = fix (incr 0)

(* Deadlock / infinite loop *)
let id x n = x n
let deadlock n = fix id n
(* to test, type [deadlock 42] *)

let deadlock n =
  let x = const true in
  whenc x (const false) n

(* non synchronous *)
(* see how long it is to compute unbounded 10000 ! *)
let unbounded =
  let x = const true in
  let h = half () in
  andl (whenc x h) x

(* the bottom element *)
let rec eps n = eps n
```

Figure 7: An implementation of sequences as functions in OCaml

```
module Streams where

data ST a = Cons a (ST a) deriving Show
-- lifting constants
constant x = Cons x (constant x)

-- pointwise application
extend (Cons f fs) (Cons x xs) = Cons (f x) (extend fs xs)

lift2 f xs ys = extend (extend (constant f) xs) ys

-- delays
(Cons x xs) `fby` y = Cons x y
pre x y = Cons x y

-- sampling
(Cons x xs) `when` (Cons True cs)  = (Cons x (xs `when` cs))
(Cons x xs) `when` (Cons False cs) = xs `when` cs

merge (Cons True c)  (Cons x xs) y  = Cons x (merge c xs y)
merge (Cons False c) x (Cons y ys)  = Cons y (merge c x ys)
```

Figure 8: A Haskell implementation with (lazy) lists

```
-- a periodic clock
every n =
  let o = reset_counter (pre 0 o = plusl n (constant 1)) (constant True)
  in o

filter n top = top `when` (every n)

hour_minute_second top =
  let second = filter (constant 10) top in
  let minute = filter (constant 60) second in
  let hour = filter (constant 60) minute in
  hour, minute, second
```

Yet, we have essentially just written Lustre functions that pass the compilation checks. The two following functions cannot be written in Lustre. The first one computes the sequence $(o_n)_{n\in\mathbb{N}}$ from an input $(x_n)_{n\in\mathbb{N}}$ such that $o_{2n} = x_n$ and $o_{2n+1} = x_n$.

```
-- the half clock
half = (constant True) `fby` notl half

-- double its input
stutter x =
  o = merge half x ((pre 0 o) when notl half) in o
```

This is an example of an oversampling function: its internal rate is faster than the rate of its input. This program can be implemented in bounded memory and time. But the Lustre compiler rejects oversampling functions. Another example of an oversampling function is one that computes the root of an input x using the Newton method.[7] It mimics an internal while loop.

$$u_n = u_{n-1}/2 + x/2u_{n-1} \qquad u_1 = x$$

[7]This example is due to Paul Le Guernic and was originally written in Signal.

```
eps = constant 0.001
`div` = lift2 (\x y -> x div y)
`minus` = lift2 (\x y -> x - y)
`lessthan` = (lift2 (\x y -> x <= y)

root input =
  let ic = merge ok input ((pre 0.0 ic) `when` (notl ok))
      uc = ((pre 0.0 uc) `div` (constant 2.0)) `plusl`
            (ic `div` ((constant 2.0) `times` (pre 0.0 uc)))
      ok = (constant true) `->`
              ((uc `minus` (pre 0.0 uc)) `lessthan` eps
      output = uc `when` ok
  in output
```

Of course, there are many other valid programs that cannot be written in Lustre, in particular those that exploit the expressiveness of Haskell and its type system, like the possibility to write higher order functions. Some of them are clearly not real-time, that is, they cannot be implemented into code that run in bounded time and memory; but some are. The language Lucid Synchrone [31] explored this research direction.

Appendix **??** define an OCaml implementation of streams as lazy data-structures. The code may appear less elegant because of the explicit play with lazyness (through operators `lazy/force`. Yet, it has the advantage to be more explicit.

2.1.4 Completing Streams with an Explitit Silent Element

In the above two encodings — a stream either implemented as a function from natural numbers to values or as a lazy data-structure — there is no explicit representation of the bottom stream, that is, the minimum element for the prefix order. It is usually noted ϵ. ϵ is the stream that stuck, i.e., that deadlocks. A classic way to represent ϵ explicitly as a stream is to complete the set of stream with a special constructor, say `Eps` so that ϵ becomes the solution of the equation $\epsilon = Eps(\epsilon)$. `Eps` can be interpreted as "not yet". This approach was used in [29], with bottom represented as $\texttt{Eps}^{\infty}$. We follow this latter notation. A similar approach was used in [8]. The set of present values was extended with a special value noted `Fail` to model a deadlock (or bottom value at the current instant) with `Fail` being absorbing. This way, the bottom element over streams is represented by the infinite stream $\texttt{Fail}^{\infty}$. We define streams in the following way:

```
data Stream a = Cons a (Stream a) | Eps a

eps = Cons Eps eps -- bottom element on streams
one = Cons 1 one
```

We adapt the definition of the data-flow primitives accordingly in Fig 9

The new definitions for `when` and `merge` are now guarded. At every instant, they either produce a "cons" or an "epsilon". Yet, the operator `fix` which computes the least fix-point of a function is not a total function, e.g., `fix tl` will loop infinitely. Can we define a *constructive definition of the least fixpoint*, that is, given `f` of the right type, always returns its leas fixpoint?

We do not resist to write the same definitions in OCaml and in Coq! For the later, the fixpoint construction should be adapted so that it is constructive, that is, `fix` must be a total function which, given f computes its least fix-point. The two implementations are given in Appendix **??** for the OCaml code and Appendix **??** for the Coq version.

We shall see, however, that the encoding a stream as a lazy data-structure hides two difficult issues: deadlocks and unboundedness of buffers.

2.1.5 Where are the monsters?

Causality monsters In the above encoding, a stream is represented as a lazy data structure. Laziness, however, allows streams to be built in a strange manner. The following definitions are

```
module Streams where

data Stream a = Cons !a (Stream a) | Eps (Stream a)

extend (Cons f fs) (Cons x xs) = Cons (f x) (extend fs xs)
extend (Eps fs) (Eps xs) = Eps (extend fs xs)
extend (Eps fs) xs = Eps (extend fs xs)
extend fs (Eps xs) = Eps (extend fs xs)

pre x xs = Cons x xs

(Cons x xs) `fby` ys = pre x xs

xs `when` (Eps(cs)) = Eps(xs `when` cs)
(Eps(xs)) `when` cs = Eps(xs `when` cs)
(Cons x xs) `when` (Cons c cs) =
  if c then (Cons x (xs `when` cs)) else (Eps (xs `when` cs))

merge (Eps cs) xs ys = Eps(merge cs xs ys)
merge cs (Eps xs) ys = Eps(merge cs xs ys)
merge (Cons True cs) (Cons x xs) ys = Cons x (merge cs xs ys)
merge (Cons False cs) xs (Cons y ys) = Cons y (merge cs xs ys)

let tl (Eps x) = Eps (tl x)
let tl (Cons _ x) = x

fix f = let rec o = Eps(f o) in o
```

Figure 9: A Haskell implementation with lazy lists with an explicit epsilon element

perfectly valid and produce infinite streams for `one`, `x` and `output`.

```
hd (Cons x y) = x
tl (Cons x y) = y
next = tl
incr (Cons x y) = Cons (x+1) (incr y)

one = Cons 1 one
x = Cons (if hd(tl(tl(tl(x)))) == 5 then 3 else 4) (Cons 1 (Cons 2 (Cons 3 one)))
output = Cons (hd(tl(tl(tl(x))))) (Cons (hd(tl(tl(x)))) (Cons (hd(x)) x))
```

The values are $\texttt{x} = 4 : 1 : 2 : 3 : 1 : \cdots$ and $\texttt{output} = 3 : 2 : 4 : 3 : 2 : 4 : \cdots$. Streams are implemented as an inductive data structure, `x` and `output` are computed sequentially:

- $x^0 = \bot$, $x^1 = \bot : 1 : 2 : 3 : one$, $x^2 = 4 : 1 : 2 : 3 : one$.
- $output^0 = \bot$, $output^1 = 3 : 2 : 4 : \cdots$

Another example:[8]

```
next x = tl x
nat = zero `fby` (incr nat)
ifn n x y = if n <= 9 then (Cons (hd(x)) (ifn (n+1) (tl(x)) (tl(y)))) else y
if9 x y = ifn 0 x y

x = if9 (incr (next x)) nat
```

The output stream (computed by Haskell) is $x = 20 : 19 : 18 : 17 : 16 : 15 : 14 : 13 : 12 : 11 : 10 : 11 : 12 : 13 : 14 : 15 : \cdots$ [9]

Are these reasonable programs? Streams are constructed in a reverse manner from the future to the past. They do not obey the intuition that we have of causality issue, that is, streams must be constructed from left to right. This is because the structural order between streams allows to fill in the holes in any order, e.g.:

$$(\bot : \bot) \leq (\bot : \bot : \bot : \bot) \leq (\bot : \bot : 2 : \bot) \leq (\bot : 1 : 2 : \bot) \leq (0 : 1 : 2 : \bot)$$

Note that it is possible to build streams with intermediate holes, that is, with undefined values in the middle, from which one can build other streams without holes:

$$half = 0 : \bot : 0 : \bot : \cdots$$

```
fail = fail
half = Cons 0 (Cons fail half)
fill x = Cons (hd(x)) (fill (tl(tl x)))
ok = fill half
```

The definition of streams in Figure 8 follows the structural order between data structures, which is also the order between functions: $\bot \leq_{\mathsf{struct}} v$ and the structure $(v : w)$ is less defined than $(v' : w')$ when v is less defined than v' and w is less defined than w': $(v : w) \leq_{\mathsf{struct}} (v' : w') \Leftrightarrow v \leq_{\mathsf{struct}} v' \wedge w \leq_{\mathsf{struct}} w'$. It does not model the intuition of causality that values in the stream must be computed from left to right. The prefix order is thus preferable, that is, $\bot \leq x$ and $v : x \leq v : y \Leftrightarrow x \leq y$.

[8] This example is due to Paul Caspi [4]

[9] We print `x:xs` instead of `Cons x xs`.

```
module Streams where
-- only consider streams where the head is always a value (not bot)
data ST a = Cons !a (ST a) deriving Show
constant x = Cons x (constant x)

extend (Cons f fs) (Cons x xs) = Cons (f x) (extend fs xs)

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

(Cons x xs) `when` (Cons True cs)  = (Cons x (xs `when` cs))
(Cons x xs) `when` (Cons False cs) = xs `when` cs

merge (Cons True c)  (Cons x xs) y  = Cons x (merge c xs y)
merge (Cons False c) x (Cons y ys)  = Cons y (merge c x ys)
```

Figure 10: A Haskell implementation with streams that enforces the prefix order between streams

Remark: This order can be adapted to functions from natural numbers to values, allowing to have intermediate holes in results [4].

$$(x \leq' y) \Leftrightarrow (\forall n \in \mathbb{N}.\, x(n) \leq y(n) \Rightarrow \forall m \geq n.\, x(m) = \bot)$$

For example, the following sequence is ordered:

$$\bot \leq (1.\bot) \leq (1.\bot.2.\bot) \leq (1.\bot.2.\bot.3.\bot)$$

Under the prefix ordering, all the previous strange programs denote $\bot$.

Causal function: A function from streams to streams, is said to be *causal* when it is monotonic for the prefix order. This definition may seem too permissive as the function `next` (or `tl`), given below and presented like the following is considered to be causal.

$$\forall n \in \mathbb{N}.\, \texttt{next}(x)(n) = x(n+1)$$

Indeed, the operator `next` can be programmed and is perfectly valid (up to syntactic details) in Lucid Synchrone (and also Lucy-n), for example.

```
let node next x = x when (false fby true)
```

We shall see in the next section how the use of such functions must nonetheless be constrained.

Possibly non-causal streams can be proscribed by forbidding values of the form $\bot.x$. Figure 10 gives a simple modification of the previous definitions in Haskell. The annotation `!a` forces the first argument of the stream constructor `Cons` to be strict, that is, to evaluate to a value. Now all the previous strange, non-causal programs have value $\bot$.

Some "synchrony" monsters Another kind of strange behavior can occur. Consider the input sequence $x = (x_i)_{i \in I\!N}$ and the function `even` such that $\texttt{even}(x) = (x_{2i})_{i \in I\!N}$. Define the equation $y = x \,\&\, \texttt{even}(x)$. It should define the sequence $(x_i \,\&\, x_{2i})_{i \in I\!N}$. In Haskell, given the definitions of Figure 10, we have:

```
even (Cons x (Cons y xs)) = Cons x (even xs)
and_gate (Cons x xs) (Cons y ys) = Cons (x && y) (and xs ys)
```

Figure 11 depicts the corresponding Kahn network. The fork on the left implicitly represents a simple duplication operator. Even though the `even` and & blocks are finite-memory processes, the

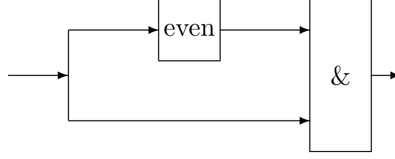

Figure 11: A non synchronous example

composition cannot be executed in bounded memory. As time goes by, the size of the FIFO of the bottom line increases and must eventually overflow.

In real-time applications, sucha compositions must be statically rejected. Moreover, all the synchronization is hidden in communication channels. Finally, even in the case where the overall memory can be statically bounded, our Haskell encoding needs a complicated runtime system, with allocation and deallocation of intermediate stream values at every step and a garbage collector. There are no real surprises here. The Kahn semantics models neither time nor the resources necessary to synchronise values. If bounded FIFOs are explicitly managed, their size has to be determined, and this can lead to possible deadlocks.

2.1.6 Clocked Streams

To account for precise synchronisations between nodes, we introduce a new semantics in which the use of data-flow primitives is restricted. We shall consider that all streams progress synchronously, each producing at global steps either a standard value or the special explicit value *abs* denoting that a value is *absent*, that is, not yet present. The size and content of buffers is also made explicit.

AbsStreamT defines the set of clocked sequences made of values from the set $T_{abs} = T + \{abs\}$.

$$
\begin{array}{lcl}
T_{abs} & = & T + \{abs\} \\
AbsStream(T) & = & (T_{abs})^{\infty}
\end{array}
$$

It is a sequence of present and absent values that can be represented in Haskell as follows.

```
data maybe a = Present a | Absent
data AbsStream a = ST (maybe a)
```

The clock of a sequence s is a boolean sequence that indicates when a value is present. For that, we define the function *clock* between clocked sequences and boolean sequences:

$$
\begin{array}{lcl}
\texttt{bool} & = & \{\texttt{0}, \texttt{1}\} \\
Clock & = & \texttt{bool}^{\infty} \\
clock(\epsilon) & = & \epsilon \\
clock(abs.x) & = & \texttt{0}.clock(x) \\
clock(v.x) & = & \texttt{1}.clock(x)
\end{array}
$$

We now make the link between the clock and the set of present/absent values more precise by defining:

$$ClockedStream(T)(c) = \{s \mid s \in (T^{abs})^{\infty} \wedge clock(s) \leq c\}$$

For a boolean sequence c, $ClockedStream(T)(c)$ is the set of sequences with clock c. It is prefix closed: if s is a prefix of s' with clock c, that is, $s' \in ClockedStream(T)(c)$, $s \in ClockedStream(T)(c)$.

The *synchronous semantics* is defined by reinterpreting the basic primitives over clocked sequences. We can replay the Kahn semantics in Section 2.1.1. It is defined by $[\![e]\!]_{\rho}^{\mathsf{abs}}$ which computes the value of e in an environment $\rho = (\rho_s, \rho_n)$. The set of values is replaced by:

$$V \;=\; AbsStream(T_1) + \cdots + AbsStream(T_n) + V \times \cdots \times V \quad \text{values for local variables}$$

$$
\begin{array}{lcl}
\texttt{const}^\sharp(i, \texttt{1}.w) & = & i.\texttt{const}^\sharp(i, w) \\
\texttt{const}^\sharp(i, \texttt{0}.w) & = & abs.\texttt{const}^\sharp(i, w) \\
\\
op^\sharp(abs.s_1, abs.s_2) & = & abs.op^\sharp(s_1, s_2) \\
op^\sharp(v_1.s_1, v_2.s_2) & = & v.op^\sharp(s_1, s_2) \text{ where } v = op(v_1, v_2) \\
\\
\texttt{fby}^\sharp(abs.s_1, abs.s_2) & = & abs.\texttt{fby}^\sharp(s_1, s_2) \\
\texttt{fby}^\sharp(v_1.s_1, v_2.s_2) & = & v_1.\texttt{fby1}^\sharp(v2, s_1, s_2) \\
\texttt{fby1}^\sharp(v, abs.s_1, abs.s_2) & = & abs.\texttt{fby1}^\sharp(v, s_1, s_2) \\
\texttt{fby1}^\sharp(v, v_1.s_1, v_2.s_2) & = & v.\texttt{fby1}^\sharp(v_2, s_1, s_2) \\
\\
\texttt{when}^\sharp(abs.s_1, abs.w) & = & abs.\texttt{when}^\sharp(s_1, w) \\
\texttt{when}^\sharp(v_1.s_1, \texttt{1}.w) & = & v_1.\texttt{when}^\sharp(s_1, w) \\
\texttt{when}^\sharp(v_1.s_1, \texttt{0}.w) & = & abs.\texttt{when}^\sharp(s_1, w) \\
\texttt{whenot}^\sharp(abs.s_1, abs.w) & = & abs.\texttt{whenot}^\sharp(s_1, w) \\
\texttt{whenot}^\sharp(v_1.s_1, \texttt{0}.w) & = & v_1.\texttt{whenot}^\sharp(s_1, w) \\
\texttt{whenot}^\sharp(v_1.s_1, \texttt{1}.w) & = & abs.\texttt{whenot}^\sharp(s_1, w) \\
\\
\texttt{merge}^\sharp(abs.w, abs.s_1, abs.s_2) & = & abs.\texttt{merge}^\sharp(w, s_1, s_2) \\
\texttt{merge}^\sharp(\texttt{1}.w, v_1.s_1, abs.s_2) & = & v_1.\texttt{merge}^\sharp(w, s_1, s_2) \\
\texttt{merge}^\sharp(\texttt{0}.w, abs.s_1, v_2.s_2) & = & v_2.\texttt{merge}^\sharp(w, s_1, s_2) \\
\\
\texttt{buffer}^\sharp(v.s, n, abs.s_1, \texttt{1}.w) & = & v.\texttt{buffer}^\sharp(s, n+1, s_1, w) \\
\texttt{buffer}^\sharp(v.s, n, v_1.s_1, \texttt{1}.w) & = & v.\texttt{buffer}^\sharp(s.v_1, n, s_1, w) \\
\texttt{buffer}^\sharp(\epsilon, n, v_1.s_1, \texttt{1}.w) & = & v_1.\texttt{buffer}^\sharp(\epsilon, n, s_1, w) \\
\texttt{buffer}^\sharp(s, n, abs.s_1, \texttt{0}.w) & = & abs.\texttt{buffer}^\sharp(s, n, s_1, w) \\
\texttt{buffer}^\sharp(s, n, v_1.s_1, \texttt{0}.w) & = & abs.\texttt{buffer}^\sharp(s.v_1, n-1, s_1, w) \text{ if } n > 0
\end{array}
$$

Figure 12: The clocked semantics for the primitives

The semantics of expressions, equations and global definitions is essentially unchanged. What changes is the interpretation of primitives on which we concentrate now.

In the following, we write ϵ for the empty sequence; v for a present value and abs for an absent value. Hence, $v.s$ denotes a clocked sequence whose head is present and $abs.s$ denotes a sequence whose head is absent.

The interpretation over clocked sequences for the primitives of the language is summarised in Figure 12. We start with the simplest operators, the generator of a constant sequence from a scalar value and the operator to lift a scalar function pointwise over input sequences.

To give a clocked semantics for the constant generator, we need an extra argument to determine whether the current value is present or not, that is:

$$
\begin{array}{lcl}
\texttt{const}^\sharp(i, \texttt{1}.w) & = & i.\texttt{const}^\sharp(i, w) \\
\texttt{const}^\sharp(i, \texttt{0}.w) & = & abs.\texttt{const}^\sharp(i, w)
\end{array}
$$

Thus, $\texttt{const}^\sharp(i, w)$ defines a constant stream with clock w, that is, $clock(\texttt{const}^\sharp(i, w)) = w$.

Consider now the semantics of $s + s'$, for example. At least two situations can occur. If the two inputs are absent, we propagate the absent on the output. If the two inputs are present, we output the sum of the two.

$$
\begin{array}{lcl}
op^\sharp(abs.s_1, abs.s_2) & = & abs.op^\sharp(s_1, s_2) \\
op^\sharp(v_1.s_1, v_2.s_2) & = & v.op^\sharp(s_1, s_2) \text{ where } v = op(v_1, v_2)
\end{array}
$$

It is tempting to add:

$$
\begin{array}{lcl}
op^\sharp(abs.s_1, v_2.s_2) & = & abs.op^\sharp(s_1, v_2.s_2) \\
op^\sharp(v_1.s_1, abs.s_2) & = & abs.op^\sharp(v_1.s_1, s_2)
\end{array}
$$

to complete with the default rule for absent values as in the initial Kahn semantics. A benefit of having added an absent value to the set of instantaneous values is that we no longer need to deal with both finite and infinite sequences. The empty sequence is simply represented as the infinite sequence abs^ω and finite sequences are simply completed to infinite ones by suffixing them with abs^ω. The synchronous monsters, however, have not been eradicated!

The synchronous aspect comes from the absence of certain definitions. For example, there is no definition to evaluate $op^\sharp(v_1.s_1, abs.s_2)$ nor $op^\sharp(abs.s_1, v_2.s_2)$, that is, both inputs must be simulataneously present or absent. Otherwise, one of them should be buffered.

2.1.7 Dealing with partial definitions: the clock calculus

What happens when one element is present and the other is absent? One idea is to statically reject these cases by requiring + to have the following type:

$$(+) : \forall cl : Clock.\, ClockedStream(\texttt{int})(cl) \times ClockedStream(\texttt{int})(cl) \rightarrow ClockedStream(\texttt{int})(cl)$$

In words, (+) expects its input streams to be on the same clock cl and guarantees to produce its output on that clock. These conditions are expressed in the form of a type that must be verified statically. This idea is exploited in [8] by defining clocked sequences in Coq as a coinductive dependent type: the type constraint for (+) and other operators, and the clock constraints for expressions, equations and functions are performed directly by the Coq type checker.

Remark 2.1 (Two type systems versus a single one)**.** There has been long debate about whether the so-called clock calculus for Lustre, Scade 6, Lucid Synchrone and Signal should merge both classical type information about data and presence/absence information. For Lustre and Signal, the clock calculus was not expressed as a type system and was applied after (classical) static typing. Separating the two, we have two signatures for (+), computed by two different type systems:

$$\begin{array}{lcll} (+) & : & \texttt{int} \times \texttt{int} \rightarrow \texttt{int} & \text{type signature} \\ (+) & : & \forall cl.\, cl \times cl \rightarrow cl & \text{clock signature} \end{array}$$

For Lucid Synchrone,[10] we also decided to separate the type system for data from that for clocks; the compiler thus calculates the two types given above. One of the reasons is that the compiler implements two other type systems, one that ensures the absence of instantaneous loops and another that analyses uses of the uninitialised delay `pre`. After much trial and error, we found it simpler to implement the various systems separately. Moreover, typing occurs sequentially (datatypes, clocks, causality, initialization) so that the information produced by earlier passes is reused by later ones. In particular, the skeleton for types is used to simplify the inference of clock types, causality types and initialization types.

Nonetheless, having several type systems adds useless redundancy in the implementation. It also complicates the formulation of correctness properties. Each of the systems precludes a particular kind of error. It also adds redundancy in interfaces, for example, if one wants to declare a data structure or function that requires a specific clock type. The debate is unfinished. In his thesis [17] and paper [18], Guatto follows an alternative approach that mixes regular type information and clock information, where the clocks express a form of modality in the spirit of guarded types.

In the following, we only consider clock types. Let us consider the case of the unit delay `fby`.

$$\begin{array}{lcl} \texttt{fby}^\sharp(abs.s_1, abs.s_2) & = & abs.\texttt{fby}^\sharp(s_1, s_2) \\ \texttt{fby}^\sharp(v_1.s_1, v_2.s_2) & = & v_1.\texttt{fby1}^\sharp(v_2, s_1, s_2) \\ \texttt{fby1}^\sharp(v, abs.s_1, abs.s_2) & = & abs.\texttt{fby1}^\sharp(v, s_1, s_2) \\ \texttt{fby1}^\sharp(v, v_1.s_1, v_2.s_2) & = & v.\texttt{fby1}^\sharp(v_2, s_1, s_2) \end{array}$$

Here again, the arguments and the result of the `fby` operator must have the same clock. A `fby` is a two-state machine: while its two arguments are initially absent, it returns an absent

[10] https://www.di.ens.fr/~pouzet/lucid-synchrone/

value and remains in the initial state ($\texttt{fby}^\sharp$). When both are present, it returns the value of its first argument and enters the steady state ($\texttt{fby1}^\sharp$) which stores the previous value of its second argument, emitting it whenever both arguments are present.

$$(\texttt{fby}) \quad : \quad \forall cl : Clock.\, cl \times cl \rightarrow cl \quad \text{clock signature}$$

Remark 2.2 (Is `fby` length preserving?)**.** It may be surprising to consider that `fby` is a length preserving function. In particular, if its second argument is empty but not the first one, it is able to return a value. But if its first argument is the empty sequence, its output is also empty.

The clock type signature does not express that the output at instant n does not depend on the second input at instant n. Hence, both the following two equations are well clocked:

$$x = x + 1 \quad \text{or} \quad x = 0 \;\texttt{fby}\; (x + 1)$$

The causality information could be embedded in the clock type system as in [26], in the case of a simple Lustre-like language (or systems with guarded types) but this calls either for adding subtyping constraints or explicit conversions. This make the clock calculus more complicated or leads to programs, in the case of a Lustre-like language, that are inelegant and not very modular.

In Lustre, Scade 6 and Lucid Synchrone, the detection of instantaneous dependences is ensured by the causality analysis, which is performed after the clock calculus. The consequence is that some valid programs cannot be written. The Signal language mixes clock inference and causality analysis [1].

We now consider the filtering (sampling) operator `when` and the combination operator `merge`.

$$\begin{array}{lcl}
\texttt{when}^\sharp(abs.s_1, abs.w) & = & abs.\texttt{when}^\sharp(s_1, w) \\
\texttt{when}^\sharp(v_1.s_1, \texttt{1}.w) & = & v_1.\texttt{when}^\sharp(s_1, w) \\
\texttt{when}^\sharp(v_1.s_1, \texttt{0}.w) & = & abs.\texttt{when}^\sharp(s_1, w) \\
\texttt{whenot}^\sharp(abs.s_1, abs.w) & = & abs.\texttt{whenot}^\sharp(s_1, w) \\
\texttt{whenot}^\sharp(v_1.s_1, \texttt{0}.w) & = & v_1.\texttt{whenot}^\sharp(s_1, w) \\
\texttt{whenot}^\sharp(v_1.s_1, \texttt{1}.w) & = & abs.\texttt{whenot}^\sharp(s_1, w) \\
\\
\texttt{merge}^\sharp(abs.w, abs.s_1, abs.s_2) & = & abs.\texttt{merge}^\sharp(w, s_1, s_2) \\
\texttt{merge}^\sharp(\texttt{1}.w, v_1.s_1, abs.s_2) & = & v_1.\texttt{merge}^\sharp(w, s_1, s_2) \\
\texttt{merge}^\sharp(\texttt{0}.w, abs.s_1, v_2.s_2) & = & v_2.\texttt{merge}^\sharp(w, s_1, s_2)
\end{array}$$

The result of the sampling operator `when` is present only when its first input is present and the sampling condition is present and true. The definition of `merge` says that the first branch must be present and the second must be absent when the condition is true; the first branch must be absent and the second present when the condition is false. Again, some rules are lacking. What is the clock of the result?

We need to define an operator on clocks.

$$\begin{array}{lcl}
cl \;\texttt{on}\; c & = & \epsilon \text{ if } cl = \epsilon \text{ or } c = \epsilon \\
(\texttt{1}.cl) \;\texttt{on}\; (\texttt{1}.c) & = & \texttt{1}.(cl \;\texttt{on}\; c) \\
(\texttt{1}.cl) \;\texttt{on}\; (\texttt{0}.c) & = & \texttt{0}.(cl \;\texttt{on}\; c) \\
(\texttt{0}.cl) \;\texttt{on}\; (abs.c) & = & \texttt{0}.(cl \;\texttt{on}\; c)
\end{array}$$

Using it, the clock type of `when` and `merge` can be expressed as:

$$\begin{array}{lcl}
\texttt{when} & : & \forall cl.\, \forall x : cl.\, \forall c : cl.\, cl \;\texttt{on}\; c \\
\texttt{merge} & : & \forall cl.\, \forall c : cl.\, \forall x : cl \;\texttt{on}\; c.\, \forall y : cl \;\texttt{on}\; (\textit{\textbf{not}}\;\; c).\, cl
\end{array}$$

The first signature says that, for any clock cl, if the first input of `when` is x and it has clock cl, the second input c has clock cl, then the result of x `when` c has clock $cl \;\texttt{on}\; c$. The rule for `whenot` is similar. The signature for `merge` says that if the first input c has clock cl, the second input x has clock $cl \;\texttt{on}\; c$ and third input y has clock $cl \;\texttt{on}\; (\textit{\textbf{not}}\;\; c)$, then the result of `merge` $c\ x\ y$ has clock cl.

The last operator we consider is the buffer. As for the definition of `const`, the production or not of a value by the operator `buffer` depends on the environment. The definition is given in Figure 12. The first parameter (s) of the operator is the contents of the buffer, the second (n) is the number of places remaining in the buffer, the third is the input stream, and the fourth is the clock (w) of the output. The semantics only gives a meaning to programs that use bounded buffers. The operator returns a value when the output clock is `1`, provided that there is at least one stored value or an input value, and it stores input values as they arrive, provided that the number of remaining places is greater than zero. Moreover, it is not possible to store a value when the buffer is full, nor to pop a value when the buffer is empty.

The rule must be completed to deal with the empty sequence ϵ. As for the Kahn semantics, the operators $op^\sharp$, $\texttt{when}^\sharp$, $\texttt{whenot}^\sharp$ and $\texttt{merge}^\sharp$ return ϵ if one of their argument is ϵ: ϵ is absorbing. The definitions for the operators `fby` and `buffer` applied to at least one ϵ argument are:

$$\begin{array}{lcl@{\qquad\qquad}lcl}
\texttt{fby}^\sharp(\epsilon, s_2) & = & \epsilon & \texttt{fby1}^\sharp(\epsilon, s_1, s_2) & = & \epsilon \\
\texttt{fby}^\sharp(v_1.s_1, \epsilon) & = & v_1.\epsilon & \texttt{fby1}^\sharp(v, \epsilon, s_2) & = & v.\epsilon \\
\texttt{fby}^\sharp(abs.s_1, \epsilon) & = & abs.\epsilon & \texttt{fby1}^\sharp(v, s_1, \epsilon) & = & v.\epsilon
\end{array}$$

$$\texttt{buffer}^\sharp(s, n, \epsilon, w) \quad = \epsilon$$

All these functions on clocked streams are continuous. In particular, the function $\texttt{buffer}^\sharp$ is monotonic: given a memory s and a number of remaining cells n (two parameters which are not inputs of the program), for any pair of inputs (s_1, w) and (s'_1, w') such that $s_1 \leq s'_1$ and $w \leq w'$, we have $\texttt{buffer}^\sharp(s, n, s_1, w) \leq \texttt{buffer}^\sharp(s, n, s'_1, w')$. Continuity follows because $\texttt{buffer}^\sharp$ is length preserving.

The semantics is not directly defined on the language kernel but on a slight variation where each constant takes an extra argument specifying the clock of its result. The buffer operator also takes extra arguments: one giving the clock of its input — when a value must be pushed, — another giving the clock of its output — when a value must be popped, — and another for the size of the buffer. The following translation defines the passage from the source language:

$$\begin{array}{lcl}
i & \longrightarrow & \texttt{const}(i, w) \\
\texttt{buffer}\ (e) & \longrightarrow & \texttt{buffer}\ (n, e', w) \text{ where } e \longrightarrow e'
\end{array}$$

The semantics for expressions, equations and programs are defined in the same way as for the Kahn semantics, except for constants and the buffer for which we take:

$$\begin{array}{lcl}
[\![\texttt{const}(i, w)]\!]_\rho^{\mathsf{abs}} & = & \texttt{const}^\sharp(i, w) \\
[\![\texttt{buffer}(n, e, w)]\!]_\rho^{\mathsf{abs}} & = & \texttt{buffer}^\sharp(\epsilon, n, [\![e]\!]_\rho^{\mathsf{abs}}, w)
\end{array}$$

These operators produce or not according to the operators that consume their output. This is why we add an extra argument giving the expected clock of the result. Moreover the `buffer` operator is initialized with an empty memory (written ϵ). The maximum size n of this memory is synthesized by the clock calculus and passed as an extra argument.

Checking Synchrony The example given in Figure 11 is now wrongly typed according to the composition of operator typing rules. Let *half* be the infinite periodic sequence $\texttt{1.0.1.0}\ldots = (\texttt{1.0})$. To fulfil the typing rule for pointwise function applications, the expression $x \,\&(x \texttt{ when } half)$ is only correct if the clock of x, say cl, equals the clock of $x \texttt{ when } half$, that is $cl\,\textsf{on}\,half$. This is impossible and this program must thus be rejected. The compiler for Lucid Synchrone [31] emits the error message:

```
let node even x = x when half
let node non_synchronous x = x & (even x)
                                 ^^^^^^^^^^^^

This expression has clock 'a on half,
but is used with clock 'a
```

In the kernel language we consider, every stream s is associated to a boolean sequence or *clock* with value `1` at the instants where s is present and `0` otherwise. Two streams can be composed (e.g., added together) without any buffer when their clocks are equal. This is essentially a typing problem [11]. As we mentionned, it was later formulated as a *shallow embedding* in Coq, showing that clock type verification could be implemented by Coq type verification.

The successive versions of Lucid Synchrone experimented with different extensions of the initial type system. We realised that having a powerful equivalence between expressions when comparing clock types $c\,\texttt{on}\,e_1$ and $c'\,\texttt{on}\,e_2$ was not very useful. In version 2, we experimented with a very simple clock calculus reminiscent of the simple ML type system with polymorphism but extended with the rule of Laufer and Odersky [23] for existential quantification [14]. This was the basis of the clock calculus used in the Scade 6 language.

The clock calculus is not only used to reject programs. Once the clock calculus is performed, every expression is annotated with a clock type. Those clocks are then used to generate efficient imperative code, in particular to factorise control structures by grouping computations that are activated on the same clock.

Remark 2.3 (Embedding the clock calculus in ML)**.** It seems possible to do a shallow embedding of a language of streams similar to the one considered in these course notes, with an encoding of both the dynamic semantics and the static clock constraints by using the Generalized Abstract Data Types (GADTs) of OCaml. We do not know if such an experiment has been completed.

In essence, the rule for typing an expression $e_1 + e_2$ is:

$$\frac{H \vdash e_1 : ck \qquad\qquad H \vdash e_2 : ck}{H \vdash e_1 \;\texttt{+}\; e_2 : ck}$$

This rule states that under the typing environment H, if e_1 has type ck and if e_2 has type ck, then $e_1 \;\texttt{+}\; e_2$ has type ck. Recall that a clock type for a stream is of the form:

$$ck \quad :: \quad \alpha \mid ck \;\texttt{on}\; e$$

where α is a clock variable and e is a boolean expression. In the synchronous case, $ck_1 \;\texttt{on}\; e_1 = ck_2 \;\texttt{on}\; e_2$ if $ck_1 = ck_2$ and $e_1 = e_2$. Equality of types ensures equality of clocks. Hence, the composition of two flows of the same type can be defined without buffering.

2.2 From synchrony to n-synchrony

In Lustre and its relatives, two input streams can be composed with a point-wise operator only when they have the same clock. This ensures that no buffer is need for the composition. This is quite constraining for video applications that are easy to describe as a Kahn process networks. If a buffer is needed, a synchronous compiler is of any help: the place where to put the buffer, its size, its input and output clock of the buffer must be determined by the programmer.

Consider for example a Picture-in-Picture as depicted in figure 13 which incrusts an image into another one. This kind of system is well modeled as a Kahn process network but the manual computation of buffer sizes is mostly manual and difficult to determine.

The PiP takes a high definition image (1920×1080 pixels), downscales it into and small definition image (720×480 pixels); it takes an other high definition image and merges it with the small definition one. The downscaler introduces a delay, hence a buffer is needed for the second image. We would like that this size be computed automatically as well as the delay (latency) for the first pixer of the output image to come up.

Can we compose non strictly synchronous streams provided their clocks are closed from each other? Can we allow for the communication between systems which are "almost" synchronous, e.g., for modeling bounded jittering or bounded delays? Can we relax the clocking rule to give more freedom to the compiler so that it can generate more efficient code, translate into regular synchronous code if necessary?

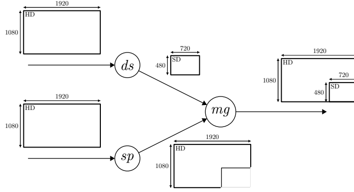

Figure 13: Picture in Picture

The n-synchronous model [12] relaxes the classical constraints of a synchronous language like to allow for the composition of streams whose clocks are not equal but can be synchronized through the introduction of a bounded buffer. It is obtained by relaxing the clock calculus with a subtyping rule. If a stream x with type ck can be consumed later with type ck' using a bounded buffer, we shall say that ck is a subtype of ck' and we write $ck <: ck'$. This allow to type a synchronous language extended with a `buffer` construct which indicates the points where the subtyping rule should be applied.

$$\frac{H \vdash e : ck \qquad ck <: ck'}{H \vdash \texttt{buffer}\ e : ck'}$$

In terms of sequences of values, `buffer` e is equivalent to e but it may delay its input using a bounded buffer. The `buffer` construct gives more freedom to the designer while preserving an execution in bounded memory.

Here, we consider a simple definition for $<:$ allowing to compare two types if they are of the form α `on` w_1 and α `on` w_2 only, with $w_1 <: w_2$. w_1 and w_2 are two boolean expressions.

Definition 2.1 (Ultimately periodic clocks)**.** We consider a particular clock language ce that define ultimately periodic boolean sequences only:

$$\begin{array}{lcl} ce & ::= & c \mid u(v) \\ u & ::= & \varepsilon \mid 0.u \mid 1.u \\ v & ::= & 0 \mid 1 \mid 0.v \mid 1.v \end{array}$$

It can be a variable name (c) or a periodic word ($u(v)$) made of a finite prefix (u) followed by the infinite repetition of a binary word (v). For example, `(10)` defines the half sequence `101010`...

2.2.1 Clock Adaptability

Here is the intuition of adaptability: *a clock w_1 is adaptable to clock w_2 if any stream with clock w_1 can be consumed with clock w_2 up to the insertion of a bounded buffer.*

To properly define this relation, we introduce the *cumulative function* of a binary word: for any binary word w, $\mathcal{O}_w(i)$ counts the number of `1`s up to the index i. Figure 14 shows the cumulative functions of $w_1 =$ `(11010)` and $w_2 =$ `0(00111)`.

Definition 2.2 (Elements and Cumulative Function of w)**.**
Let $w = b.w'$ with $b \in \{0, 1\}$. We write $w[i]$ for the i-th element of w:

$$\begin{array}{rcl} w[1] & \stackrel{def}{=} & b \\ \forall i > 1.\ w[i] & \stackrel{def}{=} & w'[i-1] \end{array}$$

We write $\mathcal{O}_w$ for the cumulative function of w:

$$\begin{array}{rcl} \mathcal{O}_w(0) & \stackrel{def}{=} & 0 \\ \forall i \geq 1.\, \mathcal{O}_w(i) & \stackrel{def}{=} & \begin{cases} \mathcal{O}_w(i-1) & \text{if } w[i] = 0 \\ \mathcal{O}_w(i-1)+1 & \text{if } w[i] = 1 \end{cases} \end{array}$$

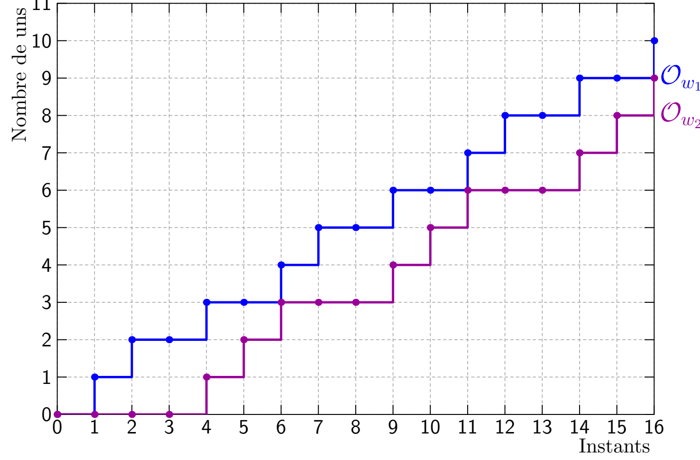

Figure 14: Cumulative functions for $w_1 = \texttt{(11010)}$ and $w_2 = \texttt{0(00111)}$.

Adaptability is the conjunction of two relations: *precedence* and *synchronizability*. Precedence ensures that there is no read in an empty buffer, that is at each instant, more values have been written than read in the buffer. Synchronizability ensures that the number of values present in the buffer during the execution is bounded.

Definition 2.3 (Synchronizability $\bowtie$, Precedence $\preceq$, Adaptability $<:$)**.**

$$\begin{array}{lcl} w_1 \bowtie w_2 & \overset{def}{\Leftrightarrow} & \exists b_1, b_2 \in \mathbb{Z}, \forall i \geq 0.\, b_1 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq b_2 \\ w_1 \preceq w_2 & \overset{def}{\Leftrightarrow} & \forall i > 0.\, \mathcal{O}_{w_1}(i) \geq \mathcal{O}_{w_2}(i) \\ w_1 <: w_2 & \overset{def}{\Leftrightarrow} & w_1 \preceq w_2 \ \wedge \ w_1 \bowtie w_2 \end{array}$$

In Figure 14, $w_1 \bowtie w_2$ since the vertical distance between the two curves is bounded and $w_1 \preceq w_2$ since the curve $\mathcal{O}_{w_1}$ is always above the one of $\mathcal{O}_{w_2}$.

Buffer Size. Consider a buffer with an input clock w_1 and output clock w_2. For every instant i, the number of elements present in the buffer is:

$$size_i(w_1, w_2) = \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i)$$

A negative value means that there were more reads than writes and this case should not appear. A sufficient size for the buffer is the maximal number of values present in the buffer during the execution:

$$size(w_1, w_2) = \max_{i \geq 1}(\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i))$$

Thus, if w_1 is adaptable to w_2, a stream with clock w_1 can be safely consumed on the clock w_2 by insertion of a bounded buffer. Otherwise, the size of the buffer may be infinite.

The purpose of the extended clock calculus is to check that bounds exist for buffer sizes and to compute them. To this aim, subtyping constraints have to be solved and it can be done for clock that are ultimately periodic (see 2.1) [12].

To reduce the algorithmic complexity of constraint resolutions and deal with non periodic clocks, it is possible to reason with *clock envelopes*. These clock envelopes are sets of concrete clocks which are not necessarily periodic. They can model various features that exist in embedded systems such as bounded jittering, logical execution time (lower and upper bounds on the numbers of atomic steps done by a process), latencies (between when an input data is read and a output is produced), scheduling resources (a process is activated a certain number of time during a period)

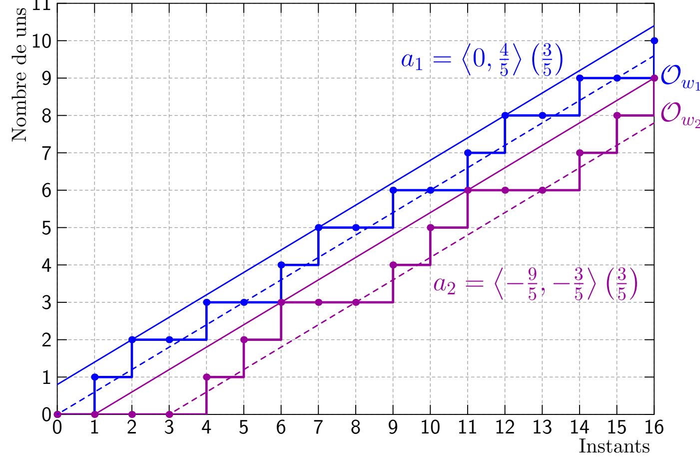

Figure 15: Envelopes of w_1 and w_2.

and the communication through buffers. Said differently, an envelope is an over abstraction of the exact clocks of the system. Hence, instead of comparing two exact clocks, we compare envelopes.

The abstraction introduced in [13] consists in reasoning on sets of clocks (or *envelopes*) defined by an asymptotic rate and two shifts bounding the potential delay with respect to this rate. It was made more precise (in the sense that it over approximates less) in [25]. Then, subtyping constraints can be replaced by linear constraints on those rates and shifts, and solved with a tool such as Glpk. We only give here an intuition of this abstraction. It was implemented for a language called Lucy-n that includes an explicit `buffer` construction and whose syntax and semantics is exactly that of the language introduced in 2. On several examples such as the *Picture in Picture*, the over-estimation due to the abstraction is small with respect to the exact solution.

2.2.2 Abstraction of Binary Words

The idea behind abstraction is to reason on sets of binary words. An abstraction bounds the cumulative function of a set of words by two linear curves with the same slope. Thus, the abstraction of an infinite binary word w keeps only the asymptotic proportion r of `1`s in w and two values b^0 and b^1 which give the minimum and maximum shift of `1`s in w compared to r. This abstract information is called an *envelope* and noted $\left\langle b^0, b^1 \right\rangle (r)$.

Definition 2.4 (Concretization)**.**

$$\mathit{concr}\left(\left\langle b^0, b^1 \right\rangle (r)\right) \stackrel{def}{=} \left\{ w \mid \forall i \geq 1, \quad \wedge \begin{array}{lcl} w[i] = \mathtt{1} & \Rightarrow & \mathcal{O}_w(i) \leq r \times i + b^1 \\ w[i] = \mathtt{0} & \Rightarrow & \mathcal{O}_w(i) \geq r \times i + b^0 \end{array} \right\}$$

with $b^0, b^1, r \in \mathbb{Q}$ and $0 \leq r \leq 1$.

The words $w_1 = \mathtt{(11010)}$ and $w_2 = \mathtt{0(00111)}$ seen previously are respectively in envelopes $a_1 = \left\langle 0, \frac{4}{5} \right\rangle \left(\frac{3}{5}\right)$ and $a_2 = \left\langle -\frac{9}{5}, -\frac{3}{5} \right\rangle \left(\frac{3}{5}\right)$ shown in Figure 15. In chronograms, an abstract value $\left\langle b^0, b^1 \right\rangle (r)$ is represented by two lines $\Delta^1 :\ r \times i + b^1$ and $\Delta^0 :\ r \times i + b^0$ that bound the cumulative functions of a set of binary words. The definition states that any rising edge must be below the line Δ^1 (solid line) and any absence of a rising edge must be above the line Δ^0 (dashed line).

For the set of words defined by an envelope to be non-empty, the line Δ^1 must be above the line Δ^0. At each instant, there must be a discrete value between the two lines. It is the case if the distance between them respects the following constraint.

Proposition 2.1 (Non-empty envelope)**.**

$$\forall a = \left\langle \frac{k^0}{\ell}, \frac{k^1}{\ell} \right\rangle \left(\frac{n}{\ell}\right) . \frac{k^1}{\ell} - \frac{k^0}{\ell} \geq 1 - \frac{1}{\ell} \Rightarrow \mathit{concr}\,(a) \neq \varnothing$$

The abstraction of a periodic binary word can be computed automatically.

Definition 2.5 (Abstraction of a Periodic Word)**.**
Let $p = u(v)$ a periodic binary word. We define $abs\,(p) \stackrel{def}{=} \langle b^0, b^1 \rangle\,(r)$ with:

$$\begin{array}{rcl} r & = & rate(p) \;=\; \frac{|v|_1}{|v|} \\ b^0 & = & \min_{i=1..|u|+|v| \text{ with } p[i]=0} (\mathcal{O}_p(i) - r \times i) \\ b^1 & = & \max_{i=1..|u|+|v| \text{ with } p[i]=1} (\mathcal{O}_p(i) - r \times i) \end{array}$$

where $|u|$ is the length of u and $|u|_1$ its number of `1`s.

The asymptotic rate r corresponds to the ratio between the number of `1`s in the periodic pattern and its length. To compute b^0 and b^1, the word must be traversed. The shift b^0 is the minimum difference when a `0` occurs between the number of `1`s seen at instant i and the ideal value $r \times i$. The shift b^1 is the maximal difference between these values when a `1` occurs.

The interest of the abstraction is to reduce the complexity of exact computations and decisions on binary words by transforming them into arithmetic manipulations on rational numbers. For example, the computation of *on* on envelopes only needs three multiplications and two additions:

Definition 2.6 (*on*~ Operator)**.** Let ${b^0}_1 \leq 0$ and ${b^0}_2 \leq 0$.[11] We define:

$$\begin{array}{cl} & \langle\; {b^0}_1 \;,\; {b^1}_1 \;\rangle\,(\; r_1 \;) \\ \textit{on}^\sim & \langle\; {b^0}_2 \;,\; {b^1}_2 \;\rangle\,(\; r_2 \;) \\ \stackrel{def}{=} & \langle\, {b^0}_1 \times r_2 + {b^0}_2 \;,\; {b^1}_1 \times r_2 + {b^1}_2 \,\rangle\,(\, r_1 \times r_2 \,) \end{array}$$

The elements of w_1 *on* w_2 are the elements of w_1 filtered by the elements of w_2. The rate of `1` in w_1 *on* w_2 is thus the product of the rate of w_1 and the one of w_2. When w_1 is sampled by w_2, its shifts are multiplied by r_2. The shifts of w_2 are added to those of w_1.

All the proofs on algebraic properties of binary sequences and abstractions have been done in Coq [25] and are available publicly. Full proofs on paper are available in the PhD. thesis of Florence Plateau [30].

3 Conclusion

In these course notes, we considered a simple first-order functional language that manipulates streams and functions that transform streams into streams. This language is reminiscent of the language Lustre invented by Caspi and Halbwachs, which was the basis for the development of the industrial language and environment SCADE, now regularly used in the development of critical control software, as well as the academic language Lucid Synchrone.

We showed that this language corresponds to a particular kind of Kahn process network that can be executed synchronously. This is expressed by associating a clock to every stream to indicate when the current value is present or not. Stream functions must then fullfil certain static rules to ensure that when a value is expected to be present (or absent), it is indeed present (or absent). Clocks can be understood as types and the associated static constraints as typing constraints in a type system with dependent types. Finally, we relax the synchronous constraint to allow communications through bounded buffers by adding sub-typing rules for when buffers are used. A relaxed clock calculus can infer the size of these communication buffers.

These course notes are far from exhaustive. In particular, they do not detail the actual clock calculus for the language, and notably the restrictions made in both Lucid Synchrone and SCADE about clock equality. In these two languages, the clock language *ce* is limited essentially to names so that clock equality reduces to name equality. These notes also sweep under the carpet the important question of causality. E.g., equations like $x = x$ or $x = x + 1$ are perfectly valid

[11] We can always lose precision on the envelopes to satisfy this condition. More details are given in [30].

from a clock calculus point-of-view but must be rejected because x depends instantaneously on itself and no sequential code can be generated: we say that x is not causal. The detection of instantaneous loops or dependencies can also be handled by static typing. Finally, these notes did not address the important question of generating sequential code. Clocks are also fundamental to code generation [16]. The clock constraints can be interpreted as dedicated techniques to ensure the perfect fusion of all the intermediate streams.

Acknowledgment

I warmly thank Peter Muller and Alexander Pretschner for their remarkable organisation of the 2018 Marktoberdorf summer school and the atmosphere they created during the moments of work and relaxation; the invited speakers who gave splendid lectures that were all different in style; the students for their very relevant and stimulating questions. I also warmly thank Timothy Bourke for his careful reading and the suggestions he made for improving these notes.

References

[1] T. Amagbegnon, L. Besnard, and P. Le Guernic. Implementation of the data-flow synchronous language signal. In *Programming Languages Design and Implementation (PLDI)*, pages 163–173. ACM, 1995.

[2] E. A. Ashcroft and W. W. Wadge. *Lucid, the data-flow programming language.* A.P.I.C. Studies in Data Processing, Academic Press, 1985.

[3] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.

[4] A. Benveniste, P. Caspi, R. Lublinerman, and S. Tripakis. Actors without directors: a kahnian view of heterogeneous systems. Technical report, Verimag, Centre Équation, 38610 Gières, September 2008. Extended version of HSCC'10.

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

[6] G. Berry. Real time programming: Special purpose or general purpose languages. *Information Processing*, 89:11–17, 1989.

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

[8] Sylvain Boulmé and Grégoire Hamon. Certifying Synchrony for Free. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 2250, La Havana, Cuba, December 2001. Lecture Notes in Artificial Intelligence, Springer-Verlag. Short version of *A clocked denotational semantics for Lucid-Synchrone in Coq*, available as a Technical Report (LIP6), at `www.di.ens.fr/~pouzet/bib/bib.html`.

[9] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model.* PhD thesis, EECS Department, University of California, Berkeley, 1993.

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

[11] Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Philadelphia, Pensylvania, May 1996.

[12] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. N-Synchronous Kahn Networks: a Relaxed Model of Synchrony for Real-Time Systems. In *ACM International Conference on Principles of Programming Languages (POPL'06)*, Charleston, South Carolina, USA, January 2006.

[13] Albert Cohen, Louis Mandel, Florence Plateau, and Marc Pouzet. Abstraction of Clocks in Synchronous Data-flow Systems. In *The Sixth ASIAN Symposium on Programming Languages and Systems (APLAS)*, Bangalore, India, December 2008.

[14] Jean-Louis Colaço and Marc Pouzet. Clocks as First Class Abstract Types. In *Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, Pennsylvania, USA, october 2003.

[15] Jean-Louis Colaco, Bruno Pagano, and Marc Pouzet. Scade 6: A Formal Language for Embedded Critical Software Development. In *Eleventh International Symposium on Theoretical Aspect of Software Engineering (TASE)*, Sophia Antipolis, France, September 13-15 2017.

[16] Gwenael Delaval, Alain Girault, and Marc Pouzet. A Type System for the Automatic Distribution of Higher-order Synchronous Dataflow Programs. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, Arizona, June 2008.

[17] Adrien Guatto. *A Synchronous Functional Language with Integer Clocks.* PhD thesis, École normale supérieure, École normale supérieure, 45 rue d'Ulm, 75230 Paris, France, 7 janvier 2016.

[18] Adrien Guatto. A generalized modality for recursion. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 482–491, 2018.

[19] N. Halbwachs. Synchronous programming of reactive systems, a tutorial and commented bibliography. In *Tenth International Conference on Computer-Aided Verification, CAV'98*, Vancouver (B.C.), June 1998. LNCS 1427, Springer Verlag.

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

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

[22] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *IFIP Congress*, pages 993–998, 1977.

[23] Konstantin Läufer and Martin Odersky. An extension of ML with first-class abstract types. In *ACM SIGPLAN Workshop on ML and its Applications, San Francisco, California*, pages 78–91, June 1992.

[24] E. Lee and D. Messerschmitt. Synchronous dataflow. *IEEE Trans. Comput.*, 75(9):1235–1245, 1987.

[25] Louis Mandel, Florence Plateau, and Marc Pouzet. Lucy-n: a n-Synchronous Extension of Lustre. In *10th International Conference on Mathematics of Program Construction (MPC'10)*, Manoir St-Castin, Québec, Canada, June 2010. Springer LNCS.

[26] Louis Mandel, Florence Plateau, and Marc Pouzet. Static Scheduling of Latency Insensitive Designs with Lucy-n. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Austin, Texas, USA, October 30 – November 2 2011.

[27] T. M. Parks, J. L. Pino, and E. A. Lee. A comparison of synchronous and cycle-static dataflow. In *ASILOMAR '95: Proceedings of the 29th Asilomar Conference on Signals, Systems and Computers (2-Volume Set)*, page 204, Washington, DC, USA, 1995. IEEE Computer Society.

[28] Thomas Martyn Parks. *Bounded scheduling of process networks*. PhD thesis, EECS Department, University of California, Berkeley, Berkeley, CA, USA, 1995.

[29] Christine Paulin-Mohring. A constructive denotational semantics for Kahn networks in Coq. In Yves Bertot, Gérard Huet, Jean-Jacques Lévy, and Gordon Plotki, editors, *From Semantics to Computer Science*, pages 383–413. Cambridge University Press, 2009.

[30] Florence Plateau. *Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire bornée*. PhD thesis, Université Paris-Sud 11, Orsay, France, 6 janvier 2010.

[31] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006. Distribution available at: https://www.di.ens.fr/~pouzet/lucid-synchrone/.

[32] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*, 2002.

A Some examples

Question A.1 (Reasonning about processes)**.** As motivated by Kahn in [21], the denotational interpretation of processes as stream function can be used to reason and prove properties about stateful systems. Consider the example in Figure 16.

- Propose an interpretation for processes `p`, `q`, `m` and `main` as continuous functions. Propose an alternative implementation of the network that use, for example, the primitives given in Figure 2. How would you prove it equivalent to the initial one?
- What does it change to remove line `(* init *)`?
- Prove that the program `main` is non blocking, i.e, if input `x` is an infinite stream, `z` is an infinite stream. It can be done with a length argument, taking $|\epsilon| = 0$ and $|v.s| = 1 + |s|$.
- Propose a sequential, equivalent implementation of `main`, made of a single elementary process with an input channel `x` and output channel `z`.

B Appendix

B.1 An Implementation of Lazy Streams in OCaml

The definition of primitives is given in Figures 17,18 and 19.

C Streams with an Explicit Silent Element in OCaml

D Streams with an Explicit Silent Element in Coq

```
type 'a buff = { push: 'a -> unit; pop: unit -> 'a }

let buffer () =
  let b = Queue.create () in
  let t = Mutex.create () in
  let push v = Mutex.lock t; Queue.push v b; Mutex.unlock t in
  let pop () = Mutex.lock t; Queue.pop b; Mutex.unlock t in
  { push = push; pop = pop }

(* Process P *)
let process_p x r y () =
  y.push 0; (* init *)
  let memo = ref 0 in
  while true do
    let v = x.pop () in
    let w = r.pop () in
    memo := if v then 0 else !memo + w;
    y.push !memo
  done

(* Process Q *)
let process_q y t z () =
  while true do
    let v = y.pop () in
    t.push v; z.push v
  done

(* Process R *)
let process_m t r () =
  while true do
    let v = t.pop () in
    r.push (v + 1)
  done

(* Put them in parallel. *)
let main x z () =
  let r = buffer () in
  let y = buffer () in
  let t = buffer () in
  ignore (Thread.create (process_p x r y) ());
  ignore (Thread.create (process_q y t z) ());
  ignore (Thread.create (process_m t r) ())
```

Figure 16: A simple implementation of KPN with threads in OCaml

```
(* A stream is a lazy data-structure *)
type 'a lazy_stream = 'a stream Lazy.t
and 'a stream = | Cons : 'a * 'a lazy_stream -> 'a stream

type ('a, 'b) system = 'a lazy_stream -> 'b lazy_stream

(* Constant generation *)
let rec const : 'a -> 'a lazy_stream = fun v -> lazy (Cons(v, const v))

(* Combinatorial function lifting *)
let rec extend :
          ('a -> 'b) lazy_stream -> 'a lazy_stream -> 'b lazy_stream =
  fun fs xs ->
  lazy
    (match Lazy.force fs, Lazy.force xs with
     | Cons(f, fs), Cons(x, xs) ->
        Cons(f x, extend fs xs))

(* Unit delay *)
let pre : 'a -> 'a lazy_stream -> 'a lazy_stream =
  fun v xs -> lazy (Cons(v, xs))

(* Initialized delay. The fby operator comes from the old Lucid language *)
let rec fby : 'a lazy_stream -> 'a lazy_stream -> 'a lazy_stream =
  fun xs ys ->
  match Lazy.force xs with
  | Cons(v, _) -> pre v ys

(* Initialization. [e1 -> e2] written here [e1 --> e2] *)
(* [e1 -> e2] is a shortcut for [if (true fby false) then e1 else e2] *)
let (-->) : 'a lazy_stream -> 'a lazy_stream -> 'a lazy_stream =
  fun xs ys -> mux (fby (const true) (const false)) xs ys

(* stream of pairs and pairs of streams *)
let pair : 'a lazy_stream * 'b lazy_stream -> ('a * 'b) lazy_stream =
  fun (xs, ys) ->
  extend (extend (const (fun x y -> (x, y))) xs) ys

let plus1 xs = extend (const (fun x -> x + 1)) xs

let fst xs = extend (const (fun (x, _) -> x)) xs
let snd xs = extend (const (fun (_, y) -> y)) xs

let notl xs = extend (const (fun x -> not x)) xs
let andl xs ys = extend (extend (const (fun x y -> x && y)) xs) ys

let eql xs ys = extend (extend (const (fun x y -> x = y)) xs) ys

let mux cs xs ys =
  extend (extend (extend
                  (const (fun c x y -> if c then x else y)) cs) xs) ys

let plusl xs ys = extend (extend (const (fun x y -> x + y)) xs) ys
```

Figure 17: An OCaml implementation of lazy streams

```
(* Filtering a stream. Operator introduced in Lustre. Noted [whenc] here *)
(* because [when] is a keyword in OCaml *)
let rec whenc : 'a lazy_stream -> bool lazy_stream -> 'a lazy_stream =
  fun xs cs ->
  lazy
    (match Lazy.force xs, Lazy.force cs with
     | Cons(x, xs), Cons(c, cs) ->
        if c then Cons(x, whenc xs cs) else Lazy.force (whenc xs cs))

let when_not x c = whenc x (notl c)

(* Union of two streams. [merge] was introduced by Kahn in 74 *)
let rec merge : bool lazy_stream -> 'a lazy_stream ->
                'a lazy_stream -> 'a lazy_stream =
  fun cs xs ys ->
  lazy
    (match Lazy.force cs with
     | Cons(true, cs) ->
        (match Lazy.force xs with
          Cons(x, xs) -> Cons(x, merge cs xs ys))
     | Cons(false, cs) ->
        (match Lazy.force ys with
           Cons(y, ys) -> Cons(y, merge cs xs ys)))


(* Least fix-point operator for streams *)
let fix: ('a lazy_stream -> 'a lazy_stream) -> 'a lazy_stream =
  fun f -> let rec y = lazy (Lazy.force (f y)) in
  y

let fix2:
      ('a lazy_stream * 'b lazy_stream ->
       'a lazy_stream * 'b lazy_stream) ->
          'a lazy_stream * 'b lazy_stream =
  fun f ->
  let xy = fix (fun xy -> pair (f (fst xy, snd xy))) in
  fst xy, snd xy

(* A representation of bottom (divergence) *)
(* [Lazy.force eps] raises exception CamlinternalLazy.Undefined. *)
let rec eps = lazy (Lazy.force eps)
let bot = eps
```

Figure 18: An OCaml implementation of lazy streams

```
(* examples *)
(* 1. half *)
let half1 () =
  let f = fun x -> notl (pre false x) in
  fix f

let half2 () =
  let f = fun x -> (pre true (notl x)) in
  fix f

(* 2. natural numbers *)
let from v =
  let f = fun x -> pre v (plusl x (const 1)) in
  fix f

(*- integration: integr(x')(n) = sum_{i = 0}^n(x'(i)) *)
let integr: (int, int) system =
  fun x' -> fix (fun o -> plusl x' (pre 0 o))

let zero =
  let f = fun x -> pre 0 x in
  fix f

(* 2. unbounded memory. *)
(* run [out print_bool max_int (unbounded ())] *)
(* and see who allocated memory grows *)
let unbounded () =
  let t = const true in
  let h = half1 () in
  andl t (whenc t h)

(* 3. Causality loop *)
let deadlock1 () =
  let deadlock = fix (fun x -> x) in
  deadlock

let deadlock2 () =
  let deadlock = fix (fun x -> plusl x (const 1)) in
  deadlock

(* 3. N-synchrony [read e.g., paper at POPL'06] *)
(* This example cannot be writting in Lustre/Scade/Lucid Synchrone *)
(* yet, it executes in bounded memory (with a buffer of size 1 *)
(* t          = 0 1 2 3 4 5 6 7 8 9  ...
 *-t when h   = 0   2   4   6   8 9  ...
 *-t whenot h =   1   3   5   7   9  ...
 *-result     =   1   5   9   13  17 ... *)
let n_synchrony () =
  let t = from 0 in
  let h = half1 () in
  plusl (whenc t (notl h)) (whenc t h)

(* auxiliary functions *)
let rec list_of n xs =
  if n = 0 then []
  else match Lazy.force xs with
       | Cons(x, xs) -> x :: (list_of (n-1) xs)
```

Figure 19: An OCaml implementation of lazy streams

```
type 'a lazy_stream = 'a stream Lazy.t
and 'a stream =
  | Eps :
      'a lazy_stream -> 'a stream
  | Cons :
      'a * 'a lazy_stream -> 'a stream

let rec eps = lazy (Eps(eps))

(* Constant generation *)
let rec const: 'a -> 'a lazy_stream = fun v -> lazy (Cons(v, const v))

let rec extend : ('a -> 'b) lazy_stream -> 'a lazy_stream -> 'b lazy_stream =
  fun fs xs ->
  lazy
    (match Lazy.force fs, Lazy.force xs with
     | Eps(fs), Eps(xs) ->
        (* This case is subsumed by the last two. Yet we add it *)
        (* purposely because only the last two reveil that inputs *)
        (* are (implicitely) buffered when one is present only *)
        Eps(extend fs xs)
     | Cons(f, fs), Cons(x, xs) -> Cons(f x, extend fs xs)
     | Eps(fs), _ -> Eps(extend fs xs)
     | _, Eps(xs) -> Eps(extend fs xs))

(* Unit delay *)
let pre : 'a -> 'a lazy_stream -> 'a lazy_stream =
  fun v xs -> lazy (Cons(v, xs))
```

Figure 20: An OCaml implementation of lazy streams with Silent

```
(* Initialized delay. The fby operator comes from the old Lucid language *)
let rec fby : 'a lazy_stream -> 'a lazy_stream -> 'a lazy_stream =
  fun xs ys ->
  match Lazy.force xs with
  | Eps(xs) -> lazy (Eps(fby xs ys))
  | Cons(v, _) -> pre v ys

(* stream of pairs and pairs of streams *)
let pair (xs, ys) =
  extend (extend (const (fun x y -> (x, y))) xs) ys

let fst xs = extend (const (fun (x, _) -> x)) xs
let snd xs = extend (const (fun (_, y) -> y)) xs

let notl xs = extend (const (fun x -> not x)) xs
let andl xs ys = extend (extend (const (fun x y -> x && y)) xs) ys

let plusl xs ys = extend (extend (const (fun x y -> x + y)) xs) ys

let rec whenc xs cs =
  lazy
    (match Lazy.force xs, Lazy.force cs with
     | Eps(xs), Eps(cs) ->
        Eps(whenc xs cs)
     | Cons(x, xs), Cons(c, cs) ->
        if c then Cons(x, whenc xs cs) else Eps(whenc xs cs)
     | Eps(xs), _ -> Eps(whenc xs cs)
     | _, Eps(cs) -> Eps(whenc xs cs))
```

Figure 21: An OCaml implementation of lazy streams with Silent

```
let rec merge cs xs ys =
  lazy
    (match Lazy.force cs, Lazy.force xs, Lazy.force ys with
     | Eps(cs), Eps(xs), Eps(ys) -> Eps(merge cs xs ys)
     | Cons(true, cs), Cons(x, xs), _ ->
        Cons(x, merge cs xs ys)
     | Cons(false, cs), _, Cons(y, ys) ->
        Cons(y, merge cs xs ys)
     | Eps(cs), _, _ ->
        Eps(merge cs xs ys)
     | Cons _, Eps(xs), _ -> Eps(merge cs xs ys)
     | Cons _, _, Eps(ys) -> Eps(merge cs xs ys))


(* Least fix-point operator for streams *)
let fix f =
  let rec y = lazy (Eps(f y)) in
  y

let fix2 f =
  let xy = fix (fun xy -> pair (f (fst xy, snd xy))) in
  fst xy, snd xy
```

Figure 22: An OCaml implementation of lazy streams with Silent

```
(* a constructive fix-point *)
(* ask for the [j-th] element of [xs]; upto [n] *)
let rec ask_option xs j n =
  (* return the j-ith value of [xs], with j<=n if it is present;
   *- None otherwise *)
  if n = 0 then None
  else
    match Lazy.force xs with
    | Eps(xs) -> ask_option xs j (n-1)
    | Cons(x, xs) -> if j = 0 then Some(x) else ask_option xs (j-1) (n-1)

(* Least fix-point *)
let lfp f =
  let rec lfp v j n =
    let v = f v in
    let vn = ask_option v j n in
    lazy
      (match vn with
       | None -> Eps(lfp v j (n+1))
       | Some(vn) -> Cons(vn, lfp v (j+1) (n+1))) in
  lfp eps 0 0
```

Figure 23: An OCaml implementation of lazy streams with Silent

```
From Coq Require Import Extraction.
Set Implicit Arguments.
Require Import Coq.Init.Datatypes.

Require Import List.
Import ListNotations.

CoInductive Stream ( A : Type ) : Type :=
| Eps : Stream A -> Stream A
| Cons : A -> Stream A -> Stream A.

(* bottom/epsilon element on streams *)
CoFixpoint bot (A : Type) : Stream A := Eps (bot A).

(* The case operator was used by Ch. Paulin in his paper of 2009] *)
(* not sure it is really useful *)
CoFixpoint case {A B : Type}
           (f : A -> Stream A -> Stream B) : Stream A -> Stream B :=
  fun s =>
    match s with
    | Eps s => Eps (case f s)
    | Cons a s => f a s
    end.

CoFixpoint const {A : Type} (a : A) : Stream A := Cons a (const a).

(* the tail of a stream *)
Definition next {A : Type} : Stream A -> Stream A := case (fun a s => s).

Definition fby {A : Type} : Stream A -> Stream A -> Stream A :=
  fun X Y => case (fun a X => Cons a (next Y)) X.

(* Unit delay *)
Definition pre {A : Type} : A -> Stream A -> Stream A :=
  fun v xs => Cons v xs.
```

Figure 24: A Coq implementation of lazy streams with Silent

```
CoFixpoint extend {A B : Type } :
  Stream (A -> B) -> Stream A -> Stream B :=
  fun fs xs =>
  match fs, xs with
  | (Eps fs), (Eps xs) =>
      (* This case is subsumed by the last two. Yet we add it *)
      (* purposely because only the last two reveil that inputs *)
      (* are (implicitely) buffered when one is present only *)
      Eps (extend fs xs)
  | (Cons f fs), (Cons x xs) => Cons (f x) (extend fs xs)
  | (Eps fs), _ => Eps (extend fs xs)
  | _, (Eps xs) => Eps (extend fs xs)
end.

CoFixpoint when {A : Type } :
  Stream A -> Stream bool -> Stream A :=
  fun xs cs =>
    match xs, cs with
    | Eps(xs), Eps(cs) =>
        Eps(when xs cs)
    | (Cons x xs), (Cons c cs) =>
        if c then Cons x (when xs cs) else Eps(when xs cs)
    | Eps(xs), _ => Eps(when xs cs)
    | _, Eps(cs) => Eps(when xs cs)
    end.

CoFixpoint merge {A : Type } :
  Stream bool -> Stream A -> Stream A -> Stream A :=
    fun cs xs ys =>
      match cs, xs, ys with
      | Eps(cs), Eps(xs), Eps(ys) => Eps(merge cs xs ys)
      | (Cons true cs), (Cons x xs), _ =>
          Cons x (merge cs xs ys)
      | (Cons false cs), _, (Cons y ys) =>
          Cons y (merge cs xs ys)
      | Eps(cs), _, _ =>
          Eps(merge cs xs ys)
      | Cons _ _, Eps(xs), _ => Eps(merge cs xs ys)
      | Cons _ _, _, Eps(ys) => Eps(merge cs xs ys)
      end.
```

Figure 25: A Coq implementation of lazy streams with Silent

```
(* Constructive fixpoint operation *)
(* As we cannot define lim_{n->infty}(f^n(eps)), we use *)
(* a diagonal argument. Intuitively, build the stream such that element *)
(* of index j is the jth-element of f^n(eps) with j <= n *)
(* n defines the diagonal *)
(* This fix point expect [f] to be monotonous/continuous in order to have *)
(* [lfp(f) <= f(lfp(f))] and [f(lfp(f)) <= lfp(f)] with [<=] the Kahn order *)
Fixpoint ask_option { A : Type } (xs : Stream A) (j : nat) (n : nat) : option A :=
  (* return the j-ith value of [xs], with j<=n if it is present;
   *- None otherwise *)
  match n with
  | 0 =>  None
  | S n =>
      match xs with
      | Eps(xs) => ask_option xs j n
      | Cons x xs =>
          match j with
          | 0 => Some(x)
          | S j => ask_option xs j n
          end
      end
  end.

CoFixpoint lfpaux { A : Type } (f : Stream A -> Stream A) (v : Stream A)
           (j : nat) (n : nat) : Stream A :=
  let v := f v in
  let vn := ask_option v j n in
  match vn with
  | None => Eps(lfpaux f v j (n+1))
  | Some(vn) => Cons vn (lfpaux f v (j+1) (n+1))
  end.

Definition lfp { A : Type } (f : Stream A -> Stream A) := lfpaux f (bot A) 0 0.

(* stream of pairs and pairs of streams *)
Definition pair { A B : Type } : Stream A -> Stream B -> Stream (A * B) :=
  fun xs ys =>
    extend (extend (const (fun x y => (x, y))) xs) ys.

Definition fst { A B : Type } : Stream (A * B) -> Stream A :=
  fun xs => extend (const (fun x => fst x)) xs.

Definition snd { A B : Type } : Stream (A * B) -> Stream B :=
  fun xs => extend (const (fun x => snd x)) xs.

Definition notl : Stream bool ->  Stream bool :=
  fun xs => extend (const (fun x => negb x)) xs.

Definition andl : Stream bool -> Stream bool -> Stream bool :=
  fun xs ys => extend (extend (const (fun x y => andb x y)) xs) ys.

Definition plusl : Stream nat -> Stream nat -> Stream nat :=
  fun xs ys => extend (extend (const (fun x y => x + y)) xs) ys.
```

Figure 26: A Coq implementation of lazy streams with Silent

```
Definition half0 :=
  let f := fun x => notl (pre false x) in
  lfp f.

Definition half1 :=
  let f := fun x => (pre true (notl x)) in
  lfp f.

(* 2. nat *)
Definition pre_incr v x := pre v (plusl x (const 1)).

Definition from v :=
  let f := pre_incr v in
  lfp f.

(* 2. unbounded memory *)
Definition unbounded :=
  let t := const true in
  let h := half0 in
  andl t (when t h).


(* 3. Causality loop *)
Definition deadlock1 { A : Type } : Stream A :=
  let deadlock := lfp (fun x => x) in
  deadlock.

Definition deadlock2 :=
  let deadlock := lfp (fun x => plusl x (const 1)) in
  deadlock.

(* 3. N-synchrone *)
Definition n_synchrone :=
  let t := from 0 in
  let h := half0 in
  plusl (when t (notl h)) (when t h).

Fixpoint list_of { A : Type } (n : nat) (xs : Stream A) :=
  match n with
  | 0 => []
  | S n =>
      match xs with
      | Eps(xs) => None :: (list_of n xs)
      | Cons x xs => Some(x) :: (list_of n xs)
      end
  end.

Eval vm_compute in (list_of 100 (deadlock1)).
Eval vm_compute in (list_of 100 (n_synchrone)).
Eval vm_compute in (list_of 100 (unbounded)).
Eval vm_compute in (list_of 1000 (from 0)).
Eval vm_compute in (list_of 100 (half0)).
Eval vm_compute in (list_of 100 (half1)).
```

Figure 27: A Coq implementation of lazy streams with Silent