Coiterative Synchronous Semantics
Part I

Marc Pouzet

Ecole normale supérieure
Paris

Marc.Pouzet@ens.fr

Course notes, MPRI, Octobre 2023

Today

- Define a reference semantics for a synchronous data-flow language;
- that is executable and preferably constructive
- it can be defined as a function in typed lambda calculus with strong normalisation (where all program terminate), e.g., the programming language of Coq.
- to get a reference interpreter.
- What for?
- to execute programs independently of a compiler; be an oracle for compiler testing; to prove compilation steps (e.g., the equivalence of some source-to-source transformations, the correctness of compile-time checks).

References

These notes are based on two technical papers:

1. Christine Paulin-Mohring. Circuits as streams in Coq, verification of a sequential multiplier. Technical report, Laboratoire de l'Informatique du Parallélisme, September 1995. Available at `http://www.ens-lyon.fr:80/LIP/lip/publis/`.
2. Paul Caspi and Marc Pouzet. A Co-iterative Characterization of Synchronous Stream Functions. VERIMAG technical report, 2007. Available at: `www.di.ens.fr/~pouzet/bib/bib.html`.

with an implementation in OCaml:

`https://github.com/marcpouzet/zrun/master`

A more advanced version with languages features from Zelus is under way. Branch `work`. See:

`https://github.com/marcpouzet/zrun/work`

I recommand you to read those two old papers and look at the code. Be free to re-use/modify it (but cite your sources).

The language kernel

We consider the following language kernel.

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

- e denotes an expression. E is a set of equations.
- e_1 `fby` e_2 is a *unit delay* (synchronous register).
- Two kinds of functions: *combinatorial* versus *sequential*.

Two classical representation for infinite streams

1. An infinite stream is a value of the following (co-inductive) type:

$$\mathit{stream}(T) = \mathit{Cons} : T \times \mathit{stream}(T) \to \mathit{stream}(T)$$

   with *Cons* the constructor (injective function).

2. Equivalently, an infinite stream is a function from the natural numbers to values, that is:

$$\mathit{stream}(T) = \mathbb{N} \to T$$

Streams

These two representations can be used to define the semantics of Kahn process
networks [Kahn, 1974, Kahn and MacQueen, 1977, Paulin-Mohring, 2009].

Also for the language Lustre [Boulmé and Hamon, 2001,
Bourke et al., 2017, Bourke et al., 2020].

The definition/existence of fix-point is complex because chains are of infinite lengths and every element in the CPO is an infinite object.

A solution is to define a relational semantics and not a functional one. Then, to prove that under some sufficient condition, the semantics exists and it unique.

This approach has pros and cons. Pros: very simple; Cons: no interpreter; sufficient conditions for existence/unicity ad-hoc.

Moreover, a stream semantics is not convenient to establish and prove some important properties like "this program runs in bounded time and space".

Can we define a reference semantics that is functional and construtive, e.g., expressed in a typed lambda calculus with strong normalization?

Streams as Sequential Processes

Instead, take a more operational interpretation of streams as sequential processes.

It was used by Paulin to model synchronous circuits [Paulin-Mohring, 1995] in Coq.

By by Caspi and Pouzet to characterize synchronous stream functions [Caspi and Pouzet, 1998].

We build on these two to define the semantics of our kernel language.

Concrete Streams

A *concrete stream* producing values in the set T is a a pair made of a step function $f : S \to T \times S$ and an initial state $s : S$.

$$coStream(T, S) = CoF(S \to T \times S, S)$$

Given a concrete stream $v = CoF(f, s)$, $nth(v)(n)$ returns the n-th element of the corresponding stream process:

$$\begin{array}{lcl} nth(CoF(f,s))(0) & = & let\ v, s = f\ s\ in\ v \\ nth(CoF(F,s))(n) & = & let\ v, s = f\ s\ in\ nth(CoF(f,s))(n-1) \end{array}$$

Two stream processes $CoF(f, s)$ and $CoF(f', s')$ are equivalent iff they compute the same streams, that is,

$$\forall n \in \mathbb{N}. nth(CoF(f,s))(n) = nth(CoF(f',s'))(n)$$

Equivalence

It amounts at finding a binary relation R between states and checking that it is preserved by the transition function.

Let R such that $R(s_f, s_g)$ means that s_f and sg are in relation by R. Two concrete streams $CoF(f, s_f)$ and $CoF(g, s_g)$ are equivalent by R if the following holds:

1. The relation holds for the initial states: $R(s_f, s_g)$;
2. It is preserved by the transitions functions:

$$\forall s_f, s_g, v, s'_f, s'_g. R(s_f, s_g) \Rightarrow ((v, s'_f = f\ s_f) \Leftrightarrow (v, s'_g = g\ s_g)) \\ \wedge\ R(s'_f, s'_g)$$

Lifting

Two operations: one turn a constant into a constant stream; one apply a function point-wise.

$$
\begin{array}{lcl}
const(v) & = & CoF(\lambda s.(v, s), ()) \\
extend(CoF(f, s))(CoF(e, se)) & = & CoF\ \lambda s.\ let\ v_f, s = f\ s\ in \\
 & & \quad let\ v_e, se = e\ se\ in \\
 & & \quad (v_f\ v_e), (s, se) \\
 & & (s, se)
\end{array}
$$

with:

$$
\begin{array}{lcl}
const(.) & : & T \rightarrow coStream(T, Unit) \\
extend(.)(.) & : & coStream(T \rightarrow T', S) \rightarrow coStream(T, S') \\
 & & \quad \rightarrow coStream(T', S \times S')
\end{array}
$$

At every step, $extend(.)(.)$ executes one step of its first argument $f\ s$ to get a value v_f and a new state s; one step of its second argument to get a value v_e and a new state se. The result is that of the application $v_f(v_e)$ and the new state (s, se).

The combination of those two operators can be used to lift a n-ary combinatorial function. E.g.,:

$$liftTwo\ f\ x_1\ x_2 = extend(extend(const(f))(x_1))(x_2)$$

with:

$$\begin{aligned} liftTwo : (T \to T' \to T'') &\to coStream(T, S) \\ &\to coStream(T', S') \\ &\to coStream(T'', S \times S') \end{aligned}$$

$$pair\ x_1\ x_2 = extend(extend(const(\lambda x, y.(x, y)))(x_1))(x_2)$$

with:

$$\begin{aligned} pair : (T \to T' \to (T \times T') &\to coStream(T, S) \\ &\to coStream(T', S') \\ &\to coStream(T \times T', S \times S') \end{aligned}$$

Length Preserving Functions

Synchronous Stream Processes

A stream function should be a value from:

$$stream(T) \to stream(T')$$

that is:

$$coStream(T, S) \to coStream(T', S')$$

We consider a particular class of stream functions that we call length preserving functions.

A *length preserving function*, from inputs in set T to outputs in set T' is a pair, made of a step function and an initial state.

$$sNode(T, T', S) = CoP(S \to T \to T' \times S, S)$$

That is, it only need the current value of its input in order to compute the current value of its output.

A value $s : coStream(T, S)$ can be represented by a value of the set $sNode(Unit, T, S)$ with $Unit$ the set with a single element `()`.

Synchronous Application

A value $f = CoP(f_t, f_s)$ can be interpreted as a stream function thanks to the function $run(.)(.)$:

$$run(CoP(f_t, f_s))(CoF(x, x_s)) = CoF\ \lambda(m, s).\ let\ v, x_s = x\ x_s\ in \\ let\ v, m = f_t\ m\ v\ in \\ v, (m, x_s) \\ (f_s, x_s)$$

with

$$run(.)(.) : sNode(T, T', S') \rightarrow coStream(T, S) \\ \rightarrow coStream(T', S' \times S)$$

It enlights the fact that, in order to produce the current value of the output, it only reads the current value of the input.

Question: give an example of a non length preserving function?

Feedback (fixpoint)

Consider a stream function:

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

and the following feedback loop written in the kernel language:

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

We would like to define a function $fix\,(.)$ such that $fix\,(f)$ is a fixpoint of f, that is, $fix\,(f) = f(fix\,(f))$.

Suppose that f is length preserving, that is, it exists a step function f_t and initial state s_0 such that $f\,x = run(CoP(f_t, s_0))(x)$. If $v_n = nth(v)(n)$, it should verify the equation:

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

A lazy functional language like Haskell allows for writting such a recursively defined value:

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

where v is defined recursively.

$cy = CoF(feedback(f_t), s)$ is a concrete stream such that
$y = \lambda n.nth(cy)(n)$ is solution of equation $y = f(y)$.

We have replaced a recursion on time, that is, a stream recursion, by a recursion on a value produced at every instant.

The abstraction, application and recursion operators can be implemented in a functional language with call-by-need (e.g, Haskell or OCaml with explicit `+lazy/force`).

This gives an interpreter for free!

This idea was introduced in [Caspi and Pouzet, 1998]. I suggest you to read it and implement it.

Where are the monsters?

feedback(.) is not a total function; it may diverge for some functions f_t.

For example, *feedback*$(f_t)()$(i)s not defined when $f_t\, s\, x = x + 1, s$. It corresponds to the stream equation written in the kernel language:

$$\texttt{let rec } x = x + 1 \texttt{ in } x$$

On the contrary, *feedback*$(f_t)()$(e)xists when $f_t\, s\, x = 1 + s, (x + 2)$. It corresponds to the stream equation:

$$\texttt{let rec } x = 1 + (0 \texttt{ fby } (x + 2)) \texttt{ in } x$$

feedback$(f_t)()$(i)s not defined for $f_t\, s, (x, y) = (y, x), s$ which corresponds to

$$\texttt{let rec } x, y = y, x \texttt{ in } x, y$$

but not for $f_t\, s\, (x, y) = 1 + s, (y + 2, x + 3)$ which corresponds to: [1]

$$\texttt{let rec } x, y = 1 + (0 \texttt{ fby } (y + 2)), x + 3 \texttt{ in } x, y$$

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

Existence of the fixpoint

The function $\mathit{feedback}(()(.)())$ cannot be defined as a function in Coq, for example, where all computations must terminate.

We want the semantics to be constructive in the sense that it is definable as a total function in a typed lambda calculus with strong normalization.

Given an initial state $s : S$, $\mathit{feedback}(f_t)$ should be a solution of:

$$X(s) = \mathit{let}\ v, s' = X(s)\ \mathit{in}\ f_t\ s\ v$$

Existence of the fixpoint

We study now the conditions for *feedback*(.) to be a total function.

To study its existence, we make a step back to denotational semantics [Reynolds, 1998], making all functions total by completing the sets of value with a special value $\perp$.

$\perp$ represents an undefined value or divergence.

Flat Domain

Given a set T, the flat domain $D = T_\perp = T + \{\perp\}$, with $\perp$ a minimal element and $\leq$ the flat order, i.e., $\forall x \in T. \perp \leq x$.

If $f : T \to T'$ is a (total) function, $f_\perp(\perp) = \perp$ and $f_\perp(x) = f(x)$ otherwise.

$(D, \perp, \leq)$ is a complete partial order (CPO). It is lifted to:

Products:

$$(v_1, v_2) \leq (v_1', v_2') \text{ iff } (v_1 \leq v_1') \wedge (v_2 \leq v_2')$$

with $(\perp, \perp)$ for the bottom element.

Functions:

$$f \leq g \text{ iff } \forall x. f(x) \leq g(x)$$

with $\lambda x. \perp$ for the bottom element.

Stream processes:

$$CoF(f, s_f) \leq CoF(g, s_g) \text{ iff } f \leq g \wedge s_f \leq s_g$$

with $CoF(\lambda s.(\perp, s), \perp)$ the bottom element, that is, the process that stuck.

Bounded Fixpoint:

If D_1 an D_2 are two CPOs. $f : D_1 \to D_2$ is monotonous iff $\forall x, y \in D_1 . x \leq_{D_1} y \Rightarrow f(x) \leq_{D_2} f(y)$.

f is continuous iff $f(lub(X)) = lub(f(X))$ where $lub(X)$ is the least upper bound of a set X.

By the Kleene theorem, a continuous function $f : D \to D$ has a minimal fix-point ($fix\,(f) = \lim_{n\to\infty}(f^n(\bot))$.

Yet, this does not lead to a computational definition because D may contain chains (comparable elements) of unbounded length.

When D is of bounded heigth, the fixpoint can be reached in a finite number of steps.

We exploit this simple observation for the computation of the fix-point

Bounded Fixpoint

The unbounded iteration for the fixpoint is replaced by a bounded one.

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

with:

$$\mathit{feedback}(.) : \mathbb{N} \to (S \to T_\bot \to T_\bot \times S) \to S \to \mathit{coStream}(T_\bot, S)$$

or the equivalent form $\mathit{feedback}(f)(s)(n)(\bot)$ with:

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

or one that stops as soon as the fixpoint is reached. $<$ is the strict order ($x < y$ iff $(x \leq y) \wedge (x \neq y)$):

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

with:

$$
\begin{array}{l}
\mathit{feedback}(.) : \mathbb{N} \to (S \to T_\bot \to T_\bot \times S) \to (T_\bot \to T_\bot \to \mathit{bool}) \\
\qquad\qquad \to S \to \mathit{coStream}(T_\bot, S)
\end{array}
$$

How many iterations are sufficient to get a fixpoint? It depends on T. Several cases can happen:

1. Either the first element v' of the pair $f_t\, v\, s$ depends on v, that is, $v' = \bot$ whenever $v = \bot$. The program contains a *causality loop*. In a lazy functional language, this would corresponds to an unbounded recursion when computing the value of v where $v, s' = f_t\, s\, v$.
2. or it does not, that is, $\bot < v'$.

In the first case, only $1 + 1$ iterations are sufficient to get the fixpoint (possibly equal to $\bot$).

$$\begin{array}{lcl} \|int\| & = & 0 \\ \|T_\bot\| & = & 1 + \|T\| \\ \|T_1 \times T_2\| & = & \|T_1\| + \|T_2\| \end{array}$$

Then, it is enough to do $\|T\| + 1$ iterations for a fixpoint on a value of type T. This idea of bounded iteration was used in [Edward and Lee, 2003] to give a denotational semantics for a synchronous block diagram language.

Splitting the step function

An alternative and equivalent representation for a concrete stream is to split the step function $f : S \to T \times V$ in two:

- an *output* function $f_o : S \to T$;
- an *update* function $f_u : S \to S$

Consequently a length preserving stream function f can be represented as a triple (s, f_o, f_u) where:

- s is the initial state;
- an *output* function $f_o : S \to T \to T$;
- an *update* function $f_u : S \to T \to S$

The feedback of f is now:

- an initial state s;
- output function $(\lambda s.fix\,(f_o\,s)) : S \to T$;
- update function $(\lambda s.let\ v = fix\,(f_o\,s)\ in\ f_u\,s\,v) : S \to S$

Semantics of the kernel language. [Colaco et al., 2023]

Semantics

We define an untyped semantics. Let V

$$V = \mathbb{Z} + \mathbb{B} + \mathbb{F} + V^*$$

where $\mathbb{Z}$ ranges for relative numbers, $\mathbb{B}$ for Booleans, $\mathbb{F}$ for floatting points and $V^* = \sum_{i \in \mathbb{N}} V^n$ where $V^0 = \{\emptyset\}$ and $V^n = V \times V^{n-1}$. The set of states S is:

$$S = V_\perp + S^*$$

A local environment ρ is a function which associate a value to a variable. It is an element of $(env\,(T_\perp))_\perp$ where:

$$env\,(T) = names \to T$$

A global environment γ is an element of $(genv\,(Global(T_\perp, S_\perp), S))_\perp$. It associate a global value to a name:

$$genv\,(T, S) = names \to Global(T, S)$$

where:

$$Global(T, S) = T + (T^* \to T^*) + sNode(T^*, T^*, S)$$

The semantics of an expression e is:

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

We use two auxiliary functions. If e is an expression and ρ an environment which associates a value to a variable name:

- $[\![e]\!]_\rho^{Init}$ is the initial state of the transition function associated to e;
- $[\![e]\!]_\rho^{State}$ is the step function.

To simplify the notation, we keep γ implicit in the definitions.

$$
\begin{array}{lcl}
[\![\texttt{pre}\ e]\!]_\rho^{Init} & = & (nil, [\![e]\!]_\rho^{Init}) \\
[\![\texttt{pre}\ e]\!]_\rho^{State} & = & \lambda(m,s).m, [\![e]\!]_\rho^{State}(s) \\
[\![x]\!]_\rho^{Init} & = & () \\
[\![x]\!]_\rho^{State} & = & \lambda s.(\rho(x), s) \\
[\![c]\!]_\rho^{Init} & = & () \\
[\![c]\!]_\rho^{State} & = & \lambda s.(c, s) \\
[\![(e_1, ..., e_2)]\!]_\rho^{Init} & = & ([\![e_1]\!]_\rho^{Init}, ..., [\![e_2]\!]_\rho^{Init}) \\
[\![(e_1, ..., e_2)]\!]_\rho^{State} & = & \lambda s.\, let\ (v_i, s_i = [\![e_i]\!]_\rho^{State}(s_i))_{i\in[1..n]}\ in \\
 & & \quad (v_1, ..., v_n), (s_1, ..., s_n)
\end{array}
$$

For this first semantics, we take $nil = \bot$.

A more precise account of errors can be taken by completing the set of values, e.g.:

$$V + \{\bot\} + \{nil\} + \{TypError\} + \{DynError\}$$

$\bot$ is the minimal element, all other being incomparable.

$$
\begin{array}{lcl}
[\![\texttt{run}\ f\ (e_1, ..., e_n)]\!]^{Init}_{\rho} & = & \rho(f)_I, [\![e_1]\!]^{Init}_{\rho}, ..., [\![e_n]\!]^{Init}_{\rho} \\
[\![\texttt{run}\ f\ (e_1, ..., e_n)]\!]^{State}_{\rho} & = & \lambda(m, s).let\ (v_i, s_i = [\![e_i]\!]^{State}_{\rho}(s_i))_{i\in[1..n]}\ in \\
 & & \quad let\ r, m' = \rho(f)_S\ m\ (v_1, ..., v_n)\ in \\
 & & \quad r, (m', s) \\
[\![f\ (e_1, ..., e_n)]\!]^{Init}_{\rho} & = & [\![e_1]\!]^{Init}_{\rho}, ..., [\![e_n]\!]^{Init}_{\rho} \\
[\![f\ (e_1, ..., e_n)]\!]^{State}_{\rho} & = & \lambda s.\ let\ (v_i, s_i = [\![e_i]\!]^{State}_{\rho}(s_i))_{i\in[1..n]}\ in \\
 & & \quad f(v_1, ..., v_n), s
\end{array}
$$

$$
[\![\texttt{let node}\ f\ (x_1, ..., x_n)\ \texttt{=}\ e]\!]^{Init}_{\gamma} \quad = \quad \gamma + [CoP(p, s)/f]
$$

where $s = [\![e]\!]^{Init}_{\rho}$ and $p = \lambda s, (v_1, ..., v_n).[\![e]\!]^{State}_{\rho+[v_1/x_1,...,v_n/x_n]}(s)$

Control structure

The conditional "if/then/else" always executes its three arguments. The "when/then/else" only execute one branch:

$$
\begin{array}{lcl}
[\![\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2]\!]_\rho^{Init} & = & ([\![e]\!]_\rho^{Init}, [\![e_1]\!]_\rho^{Init}, [\![e_2]\!]_\rho^{Init}) \\
[\![\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2]\!]_\rho^{State} & = & \lambda(s, s_1, s_2).\mathit{let}\ v, s = [\![e]\!]_\rho^{State}(s)\ \mathit{in} \\
 & & \quad \mathit{let}\ v_1, s_1 = [\![e_1]\!]_\rho^{State}(s_1)\ \mathit{in} \\
 & & \quad \mathit{let}\ v_2, s_2 = [\![e_2]\!]_\rho^{State}(s_2)\ \mathit{in} \\
 & & \quad (\texttt{if } v \texttt{ then } v_1 \texttt{ else } v_2, \\
 & & \quad (s, s_1, s_2))
\end{array}
$$

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

Modular Reset

Reset a computation when a boolean condition is true.

$$
\begin{array}{lcl}
[\![\texttt{reset}\ e_1\ \texttt{every}\ e_2]\!]_\rho^{Init} & = & ([\![e_1]\!]_\rho^{Init}, [\![e_1]\!]_\rho^{Init}, [\![e_2]\!]_\rho^{Init}) \\
[\![\texttt{reset}\ e_1\ \texttt{every}\ e_2]\!]_\rho^{State} & = & \lambda(s_i, s_1, s_2). \\
 & & \quad \mathit{let}\ v_2, s_2 = [\![e_2]\!]_\rho^{State}(s_2)\ \mathit{in} \\
 & & \quad \mathit{let}\ v_1, s_1 = [\![e_1]\!]_\rho^{State}(\mathit{if}\ v_2\ \mathit{then}\ s_i\ \mathit{else}\ s_1)\ \mathit{in} \\
 & & \quad v_1, (s_i, s_1, s_2)
\end{array}
$$

This definition duplicates the initial state. An alternative is:

$$
\begin{array}{lcl}
[\![\texttt{reset}\ e_1\ \texttt{every}\ e_2]\!]_\rho^{Init} & = & ([\![e_1]\!]_\rho^{Init}, [\![e_2]\!]_\rho^{Init}) \\
[\![\texttt{reset}\ e_1\ \texttt{every}\ e_2]\!]_\rho^{State} & = & \lambda(s_1, s_2). \\
 & & \quad \mathit{let}\ v_2, s_2 = [\![e_2]\!]_\rho^{State}(s_2)\ \mathit{in} \\
 & & \quad \mathit{let}\ s_1 = \mathit{if}\ v_2\ \mathit{then}\ [\![e_1]\!]_\rho^{Init}\ \mathit{else}\ s_1\ \mathit{in} \\
 & & \quad \mathit{let}\ v_1, s_1 = [\![e_1]\!]_\rho^{State}(s_1)\ \mathit{in} \\
 & & \quad v_1, (s_1, s_2)
\end{array}
$$

Equations and Local Definitions

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

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

The last case is to make the definition total. $+$ is the union of two environments provided their domain do not intersect (otherwise, it returns $\bot$).

If E is an equation, ρ is an environment, $[\![E]\!]_\rho^{Init}$ is the initial state and $[\![E]\!]_\rho^{State}$ is the step function. The semantics of an equation eq is:

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

$$
\begin{aligned}
[\![p = e]\!]^{Init}_{\rho} &= [\![e]\!]^{Init}_{\rho} \\
[\![p = e]\!]^{State}_{\rho} &= \lambda s.\mathit{let}\ v, s = [\![e]\!]^{State}_{\rho}(s)\ \mathit{in}\ [v|p], s
\end{aligned}
$$

$$
\begin{aligned}
[\![E_1 \texttt{ and } E_2]\!]^{Init}_{\rho} &= ([\![E_1]\!]^{Init}_{\rho}, [\![E_2]\!]^{Init}_{\rho}) \\
[\![E_1 \texttt{ and } E_2]\!]^{State}_{\rho} &= \lambda(s_1, s_2).\mathit{let}\ \rho_1, s_1 = [\![E_1]\!]^{State}_{\rho}(s_1)\ \mathit{in} \\
&\qquad\qquad \mathit{let}\ \rho_2, s_2 = [\![E_2]\!]^{State}_{\rho}(s_2)\ \mathit{in} \\
&\qquad\qquad \rho_1 + \rho_2, (s_1, s_2)
\end{aligned}
$$

$$
\begin{aligned}
[\![\texttt{rec } E]\!]^{Init}_{\rho} &= [\![E]\!]^{Init}_{\rho} \\
[\![\texttt{rec } E]\!]^{State}_{\rho} &= \lambda s.\mathit{fix}\,(\|E\| + 1)\,(\lambda s, \rho'.[\![E]\!]^{State}_{\rho+\rho'}(s))(s)
\end{aligned}
$$

$$
\begin{aligned}
[\![\texttt{let } E \texttt{ in } e']\!]^{Init}_{\rho} &= [\![E]\!]^{Init}_{\rho}, [\![e']\!]^{Init}_{\rho+[\bot/x]} \\
[\![\texttt{let } E \texttt{ in } e']\!]^{State}_{\rho} &= \lambda(s,s').\mathit{let}\ \rho', s = [\![E]\!]^{State}_{\rho}(s)\ \mathit{in} \\
&\qquad \mathit{let}\ v', s' = [\![e']\!]^{State}_{\rho+\rho'}(s')\ \mathit{in} \\
&\qquad v', (s,s')
\end{aligned}
$$

$$
\begin{aligned}
[\![\texttt{let rec } E \texttt{ in } e']\!]^{Init}_{\rho} &= [\![e]\!]^{Init}_{\rho}, [\![e']\!]^{Init}_{\rho+[\bot/x]} \\
[\![\texttt{let rec } E \texttt{ in } e']\!]^{State}_{\rho} &= \lambda(s,s').\mathit{let}\ \rho', s = [\![\texttt{rec } E]\!]^{State}_{\rho}(s)\ \mathit{in} \\
&\qquad \mathit{let}\ v', s' = [\![e']\!]^{State}_{\rho+\rho'}(s')\ \mathit{in} \\
&\qquad v', (s,s')
\end{aligned}
$$

Fix-point for mutually recursive streams

Consider:

```
let node sincos(x) = (sin, cos) where
  rec sin = int(0.0, cos)
  and cos = int(1.0, -. sin)
```

The fix-point construction used in the kernel language is able to deal with mutually recursive definitions, encoding them as:

```
sincos = (int(0.0, snd sincos), int(1.0, -. fst sincos)
```

Encoding mutually recursive streams

A set of *mutually recursive streams*:

$$e \quad ::= \quad \texttt{let rec}\ x = e\ \texttt{and} \ldots \texttt{and}\ x = e\ \texttt{in}\ e$$

is interpreted as the definition of a single recursive definition such that:
$\texttt{let rec}\ x_1 = e_1\ \texttt{and} \ldots \texttt{and}\ x_n = e_n\ \texttt{in}\ e$ means:

$$\texttt{let rec}\ x = (e_1, (e_2, (\ldots, e_n)))[e'_1/x_1, \ldots, e'_n/x_n]\ \texttt{in}$$

with:

$$\begin{array}{lcl} e'_1 & = & \texttt{fst}(x) \\ e'_2 & = & \texttt{fst}(\texttt{snd}(x)) \\ \ldots & & \\ e'_n & = & \texttt{snd}^{n-1}(x) \end{array}$$

That is, if the n variables $x_1, \ldots, x_n$ are streams whose outputs are of type $coStream(T_i, S_i)$ with $i \in [1..n]$, $fix\,(.)$ is applied to a function of type $S \to T_1 \times \ldots \times T_n \to (T_1 \times \ldots \times T_n) \times S$ with $S = (S_1 \times (\ldots \times S_n))$. All streams progress synchronously.

Where are the bottom values?

From the semantics we have given, some equations have the constant bottom stream as minimal fix-point. E.g.:

```
let node f(x) = o where rec o = o
```

Indeed:

$$fix\left(\lambda s, v.[\![o]\!]^{State}_{\rho+[v/o]}(s)\right) = fix\left(\lambda s, v.(v, s)\right) = \lambda s, v.(\bot, s)$$

An other example is:

```
let node f(z) = (x, y) where rec x = y and y = x
```

Indeed:

$$\begin{aligned} fix\left(\lambda s, v.[\![(\mathtt{snd}(v), \mathtt{fst}(v))]\!]^{State}_{\rho+[v/x]}(s)\right) &= fix\left(\lambda s, v.(\mathtt{snd}(v), \mathtt{fst}(v)), s\right) \\ &= \lambda s.(\bot, \bot), s \end{aligned}$$

We are interesting in finding sufficient conditions to ensure that the output is not bottom or, even more, that it does not contain bottom.

Def-use chains

In term of def-use chains of variables based on the occurrence of variables in expression, there is a cyclic dependence in both examples:

`x` depends on `y` which depends on `x`

The following definition does not define a bottom stream (provided that inputs are non bottom streams). [2]

```
let node euler_forward(h, x0, xprime) = x where
  rec x = x0 fby (x +. h *. xprime)
```

[2] We suppose that all imported functions are total.

Break the dependence cycles with a unit delay

The graphical argument we used — the dependence graph between variables is cyclic or not — can be adapted to take the unit delay into account.

Say that $\texttt{pre}_c(e)$ does not depend on variable in e; hence, a variable x defined by an equation $x = e$ only depends on the variables in e which do not appear on the right of a unit delay.

The dependence relation between variables in the `euler_backward` is acyclic.

Is that enough?

The dependence graph is a rough abstraction of the dependence relation. It does not take into account the actual values of expressions. It over approximate instantaneous dependences.

Some sets of equations whose associate dependence graph is cyclic do define non bottom streams. E.g.,:

```
let node f(y) = x
  where
    rec x = if false then x else 0
```

The conditional only needs the current value of its first (or second) argument with the condition is true (or false).

This program is rejected by Lustre compilers.

Mutually recursive definitions

The notion of dependence is subtle. All function below are such that if `x` is non bottom, outputs `z` and `t` are non bottom.

```
let node good1(x) = (z, t) where
  rec z = t and t = 0 fby z

let node good2(x) = (z, t) where
  rec (z, t) = (t, 0 fby z)

let node good3(x) = (fst r, snd r) where
  rec r = (snd r, 0 fby (fst r))

let node pair(r) = (snd r, 0 fby (fst r))

let node good4(x) = r where
  rec r = pair(r)
```

Do we want to accept all of them? What is the criterium to accept them or not? The next lesson will be devoted to the precise definition of what is a dependence and its exploitation to generate sequential code.

The following is a classical example that is "constructively causal" but is also rejected by Lustre compilers.

```
let node mux(c, x, y) = present c then x else y

let node constructive(c, x) = y
  where rec
    rec x1 = mux(c, x, y2)
    and x2 = mux(c, y1, x)
    and y1 = f(x1)
    and y2 = g(x2)
    and y = mux(c, y2, y1)
```

If we look at the def-use chains of variables, there is a cycle in the dependence graph:

- `x1` depends on `c`, `x` and `y2`;
- `x2` depends on `c`, `y1` and `x`;
- `y1` depends on `x1`; `y2` depends on `x2`;
- `y` depends on `c`, `y2` and `y1`.

By transitivity, `y2` depends on `y2` and `y1` depends on `y1`.

Yet, if `c` and `x` are non bottom streams, the fix-point that defines `(x1,x2,y1,y2,y)` is a non bottom stream.

It can be proved to be equivalent to:

```
let node constructive(c, x) = y
  where rec
    rec y = mux(c, g(f(x)), f(g(x)))
```

Question: can you prove it? How?

In term of an implementation into a circuit, the cyclic version has a single occurrence of `f` and `g` whereas the second has two copies of each.

A cyclic combinatorial circuit can be exponentially smaller than its non cyclic counterpart. [3]

The *causality analysis* ensures that an expression does not produce bottom and can be translated into an expression with no fix-point.

[3] See notes for references.

The following example (written in Zelus) also defines a node whose output is non bottom:

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

that can be interpreted as the following program in the language kernel:

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

Is it causal?

Supposing the c1, c2 and y are not bottom values, taking e.g., true for c1 and c2, starting with $x_0 = \bot$, $z_0 = \bot$, $t_0 = \bot$ and $r_0 = \bot$, the fixpoint is the limit of the sequence:

$$x_n = y + 1 \wedge z_n = t_{n-1} + 1 \wedge t_n = x_{n-1} + 1 \wedge r_n = z_{n-1} + 2$$

and is obtained after 4 iterations.

This program is causal: if inputs are non bottom values, all outputs are non bottom values and this is the case for all computations of it.

The inpact of static code generation

Nonetheless, if we want to generate statically scheduled sequential code, the control structure must be duplicated:
(1) test `c1` to compute `x`; (2) test `c2` to compute `t`; (3) test (again) `c1` to compute `z`; (4) test (again) `c2` to compute `r`

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

Accepting program with interwined dependences has an impact on code size and efficiency.

It is possible to overconstraint the causality analysis and control structures to be *atomic* (outputs all depend on all inputs).

Removing Recursion

Yet, the semantics we have given computes a step function which must be evaluated lazilly. Is this really a progress w.r.t the co-inductive semantics?

Some recursive equations can be translated into non recursive definitions.

Consider the stream equation:

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

Can we get rid of recursion in this definition? Surely we can, since it can be compiled into a finite state machine corresponding to the co-iterative process:

$$nat = Co(\lambda s.(s, s+1), 0)$$

First: let us unfold the semantics

Consider the recursive equation:

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

Let us try to compute the solution of this equation manually by unfolding the definition of the semantics.

Let $x = CoF(f, s)$ where f is a transition function of type $f : S \to X \times S$ and $s : S$ the initial state, we write: $x.step$ for f and $x.init$ for $x : init$ for s.

The bottom stream, to start with, is:

$$x^0 = CoF(\lambda s.(\bot, s), \bot)$$

The equation that defines `nat` can be rewritten as
$let\ rec\ nat = f(nat)\ in\ nat$ with `let node f x =` $(0$ `fby` $x) + 1$.

The semantics of f is:

$$f = CoP(f_s, s_0) = CoP(\lambda s, x.(s + 1, x), 0)$$

Solving $nat = f(nat)$ amount at finding a stream X such that:

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

Let us proceed iteratively by unfolding the definition of the semantics. We have:

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

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

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

We have reached the fix-point $CoF(\lambda s.(s+1, s+1), 0)$ in three steps.

Syntactically Guarded Stream Equations

We give now a simple, syntactic condition under which the semantics of mutually recursive stream equations does not need any fix point.

Consider a node $f : coStream(T, S) \rightarrow coStream(T, S')$ whose semantics is (f_t, s_t) with $f_t : S' \rightarrow T \rightarrow T' \times S'$ and $s_t : S'$.

The semantics of an equation $y = f(y)$ is: [4]

$$[\![\texttt{let rec } y = f(y) \texttt{ in } y]\!]_\rho^{Init} = s_t$$

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

The recursion on time (a stream recursion) is transformed into a recursion on the instant.

[4] We reason upto bisimulation, that is, independently on the actual representation of the internal state.

Two cases can happen:

- We deal with a 0-order expression (a stream expression or product of 0-order expressions), then:
  - Either the first element of the pair $f_t\, v\, s$, that is v, s' depends on v and we have an unbounded recursion — the program contains a causality loop —;
  - or it does not and the evaluation succeeds.
- the expression is an higher order one and its boundedness depends on semantic conditions to be checked in each case.

For example, the following equation:

```
let rec nat = nat + 1 in nat
```

is not causal since `x` depends instantaneously on itself and its evaluation have an unbounded recursion.

When the program does not contain any causality loop, it means that indeed the recursive evaluation of the pair v, s' can be split into two non recursive ones.

This case appears, for example, when every stream recursion appears on the right of a unit delay `pre`. A synchronous compiler takes advantage of this in order to produce non recursive code like the co-iterative *nat* expression given above.

Yet, if we are interested in defining an interpreter only, the co-iterative semantics can be used for that purpose.

For example, consider the equation $y = f(v \texttt{ fby } x)$. Its semantics is:

$$
\begin{aligned}
[\![\texttt{let rec } x = f(v \texttt{ fby } x) \texttt{ in } x]\!]_{\rho}^{Init} &= (v, s_t) \\
[\![\texttt{let rec } x = f(v \texttt{ fby } x) \texttt{ in } x]\!]_{\rho}^{State}(m, s) &= \mathit{let\ rec}\ v, s' = f_t\ m\ s\ \mathit{in} \\
&\quad\ v, (v, s')
\end{aligned}
$$

But this time, the recursion is no more necessary, that is:

$$
[\![\texttt{let rec } x = f(v \texttt{ fby } x) \texttt{ in } x]\!]_{\rho}^{State}(m, s) = \mathit{let}\ v, s' = f_t\ m\ s\ \mathit{in}\ v, (v, s')
$$

Putting Mutually Recursive Equation in Normal Form

Consider:

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

Rewrite it into:

```
let rec sin = 0.0 fby sin_next
    and pre_cos = 0.0 fby cos
    and sin_next = sin +. h *. cos
    and cos = 1.0 -> pre_cos +. h *. sin
  sin, cos
```

All the unit delay are un-nested; their argument is a variable.
Gather equations on delays on the top; statically schedule other equations according to read/write variables.

The transition function is:

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

and initial state:

$$(0.0, 0.0, \mathit{true})$$

There is no more recursion in the transition function.

The Semantics for Normalised Equations

Consider a set of mutually recursive equations such that it can be put under the following form:

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

where

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

where $Var(p)$ and $Var(e)$ are the set of variable names appearing in p and e.

Its transition function is:

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

with initial state:

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

if $[\![e_i]\!]_\rho^{Init} = s_i$ and $[\![e]\!]_\rho^{Init} = s$.

When a set of mutually recursive streams can be put in the above form, its transition function does not need a fix-point. It can be statically scheduled into a function that can be evaluated eagerly.

This removing of the recursion is the basis of generation of statically scheduled code done by a synchronous language compiler.

Question: prove that the new semantics for the `let`/`rec` operation is correct, that is, it produces the same stream as the original semantics.

Non length-preserving stream functions

Non length-preserving stream functions

The stream functions we have considered are *length preserving*: to produce one output, their step function needs only one input. This is what allowed us to implement a stream function with type:

$$coStream(T, S) \to coStream(T', S')$$

by a value of type:

$$(S' \to T \to T' \times S') \times S'$$

Hence, it is not possible to represent non length preserving functions like the function `even` which removes one element over two of the input stream. In Haskell, with `:` the operation on lists: [5]

```
even (x : (x' : xs)) = x : (even xs)
```

The destructor function of the input `hd,tl` has to be applied twice in the transition function of the result.

[5] See notes of the previous class.

This would also be the case of filter-like functions like `when` defined as:

$$
\begin{array}{lcl}
(x : xs)\ \texttt{when}\ (\mathit{true} : cs) & = & x : xs\ \texttt{when}\ cs \\
(x : xs)\ \texttt{when}\ (\mathit{false} : cs) & = & xs\ \texttt{when}\ cs
\end{array}
$$

Complementing Streams with Absent Values

An obvious idea to overcome the problem and turn these functions into synchronous ones would be to consider the functor F:

$$F_T(S) = S + (T \times S)$$

with the two value constructors (injective functions):

$$\mathtt{S} : S \to F_T\, S \text{ and } \mathtt{P} : T \times S \to F_T\, S$$

where `P` stands for "present" and `S` for "silent". The set of streams is now:

$$\mathit{clockedStream}(T, S) = (S \to (S + (T \times S)) \times S$$

Given $t, s : \mathit{clockedStream}(T, S)$, the process $(t\, s)$ can be silent, that is, it only updates its state without outputing values and return the next state or output a value and returns the next state.

Then a transition function for even could be:

$$\begin{aligned}\texttt{even}\ (\mathit{CoF}(t, s)) \quad = \quad \mathit{CoF}\ \ \lambda(e, s).\ & \texttt{match}\ t\ s\ \texttt{with} \\ & \mid \mathtt{S}(sx) \to \mathtt{S}(e, sx) \\ & \mid \mathtt{P}(vx, sx) \to \mathit{if}\ e\ \mathit{then}\ \mathtt{P}(vx, (\mathit{false}, sx)) \\ & \qquad\qquad\qquad\qquad \mathit{else}\ \mathtt{S}(\mathit{true}, sx) \\ & (\mathit{true}, sx)\end{aligned}$$

where e is a boolean state condition telling whether the current step is an even one or not.

However, the question is now: does this functor still define streams? An answer to this question is as follows:

The co-algebra of clocked streams

Theorem (Co-algebra of clocked streams)

The terminal co-algebra associated to the functor $F_T(S) = S + (T \times S)$

- *as ground set the set of streams of values in* $value(T) = 1 + T$*, the set* T *complemented with an empty value with* $1 = \{()\}$ *with the value constructors:* $\mathtt{E} : value(T)$ *and* $\mathtt{V} : T \to value(T)$*:*

$$stream(T) = (value(T))^{\mathbb{N}}$$

- *and as destructor,*

$$dest(v : vs) = \ \mathtt{match}\ v\ \mathtt{with}\ \mathtt{E} \to \mathtt{S}(vs)\ |\ \mathtt{V}(v') \to \mathtt{P}(v', vs)$$

Proof:

Given $tx : S \to F\ T\ S$ a transition function, let us denote by $next(.,.)$ the iterated next state function $next(.)$:

$$\begin{array}{lcl} next(s) & = & \texttt{match}\ tx\ s\ \texttt{with}\ \mathrm{S}(s') \to s'\ \texttt{|}\ \mathrm{P}(v, s') \to s' \\ next(n, s) & = & \mathit{if}\ n = 0\ \mathit{then}\ next(s) \\ & & \mathit{else}\ next(n-1, next(s)) \end{array}$$

Any function *run* which makes the following diagram commute:

$$\begin{array}{ccc} S & \text{---}\mathit{run}\longrightarrow & (\mathit{value}(T))^{\mathbb{N}} \\ \Big\vert & & \Big\vert \\ tx & & \mathit{dest} \\ \downarrow & & \downarrow \\ F_T\ S & \text{---}F\ \mathit{id}\ \mathit{run}\longrightarrow & F_T\ (\mathit{value}(T))^{\mathbb{N}} \end{array}$$

yields:

$$\begin{aligned} dest(run(s)) &= \texttt{match}\ run(s)(0)\ \texttt{with} \\ &\quad \mid \texttt{E} \rightarrow \texttt{S}(\lambda n.run(s)(n+1)) \\ &\quad \mid \texttt{V}(v) \rightarrow \texttt{P}(v, \lambda n.(run(s)(n+1))) \\ &= \texttt{match}\ t\ s\ \texttt{with} \\ &\quad \mid \texttt{S}(s') \rightarrow \texttt{S}(run(s')) \\ &\quad \mid \texttt{P}(v, s') \rightarrow \texttt{P}(v, run(s')) \end{aligned}$$

that is:

$$run(s)(0) = \texttt{match}\ t\ s\ \texttt{with}\ \texttt{S}(s') \rightarrow \texttt{E} \mid \texttt{P}(v, s') \rightarrow \texttt{V}(v)$$

$$\begin{aligned} run(s)(n+1) &= \texttt{match}\ t\ s\ \texttt{with} \\ &\quad \mid \texttt{S}(s') \rightarrow run(s')(n) \\ &\quad \mid \texttt{P}(v, s') \rightarrow run(s')(n) \\ &= run(next(s))n \end{aligned}$$

This uniquely defines *run* as:

$$run(s)(n) = \texttt{match}\ t\ (next(n, s))\ \texttt{with}\ \texttt{S}(s') \rightarrow \texttt{E} \mid \texttt{P}(v, s') \rightarrow \texttt{V}(v)$$

Definition (Clocks)

The clock of a clocked stream $s : (1 + T)^{\mathbb{N}}$ is the boolean stream:

$$\begin{aligned} clock(s) = \lambda n.\ & \texttt{match}\ s(n)\ \texttt{with} \\ & \mid \texttt{E} \rightarrow \texttt{false} \\ & \mid \texttt{V}(v) \rightarrow \texttt{true} \end{aligned}$$

Note that clocks are just ordinary streams, *i.e.* without E elements.
Yet this result shows also that we can as well assimilate clocked streams with ordinary streams with "empty" values [6]. This allows us to easily reuse the result for length preserving streams developed previously. We thus will adopt this point of view in the sequel, by taking:

$$\begin{array}{lcl} value(T) & = & \texttt{E} + \texttt{V}(T) \\ clockedStream(T, S) & = & coStream(value(T), S) \end{array}$$

[6] This quite obvious result has been used and rediscovered many times since the pioneering work of F. Boussinot [Boussinot, 1992]. Yet, the above proof may bring some insight about the need for "empty" values.

New Definitions for Primitives

We can now revisit our previously defined operators as well as create new ones. When defining binary operators, like `extend` we now find the following problem: what to do if one argument yields a value while the other one does not?

At least three possibilities are open:

1) store the value in a state variable implementing a FIFO queue, until it matches an incoming value of the other argument,
2) generate an execution error,
3) or statically reject this situation.

As an extension of what is done for Lustre [Halbwachs et al., 1991] we choose the third solution and write:

$$
\begin{array}{l}
(CoF(tf, if))\,((CoF(te, ie))) = \\
\quad CoF\ \ (\lambda(sf, se).\ \texttt{match}\ (tf\ sf), (te\ se)\ \texttt{with} \\
\qquad\qquad\qquad\qquad \mid (\mathtt{E}, sf'), (\mathtt{E}, se') \rightarrow \mathtt{E}, (sf', se') \\
\qquad\qquad\qquad\qquad \mid (\mathtt{V}(vf), sf'), (\mathtt{V}(ve), se') \rightarrow \mathtt{V}(vf\ ve), (sf', se'), \\
\qquad\quad (if, ie))
\end{array}
$$

Under the condition that the clocks of the two arguments are the same. Otherwise, the program should raise an execution error (a pattern-matching failure).

The purpose of the *clock calculus* is to statically ensure that such errors do not occur.

When expressions have passed the analysis, clock information is used to remove the dynamic test of presence/absence.

Primitive Functions

When: The co-iterative definition for the filter is as follows, assuming its two arguments share the same clock:

$$
\begin{array}{l}
(CoF(tx, ix))\ \texttt{when}\ (CoF(tc, ic)) = \\
CoF\ \ (\lambda(sx, sc).\ \texttt{match}\ (tx\ sx), (tc\ sc)\ \texttt{with} \\
\qquad\qquad\qquad\qquad \mid (\texttt{E}, sx'), (\texttt{E}, sc') \rightarrow \texttt{E}, (sx', sc') \\
\qquad\qquad\qquad\qquad \mid (\texttt{V}(vx), sx'), (\texttt{V}(\texttt{true}), sc') \rightarrow \texttt{V}(vx), (sx', sc') \\
\qquad\qquad\qquad\qquad \mid (\texttt{V}(vx), sx'), (\texttt{V}(\texttt{false}), sc') \rightarrow \texttt{E}, (sx', sc'), \\
\qquad (ix, ic))
\end{array}
$$

The clock of the result depends on the boolean condition.

If the clock of the two arguments is $(CoF(tcl, scl))$, the clock of the result is $CoF(tcl, scl)$ on $CoF(tc, sc)$:

$$
\begin{array}{l}
CoF(tcl, icl)\ \texttt{on}\ CoF(tc, ic) = \\
\quad CoF\ \ \lambda(scl, sc).\ \texttt{match}\ tcl\ scl\ \texttt{with} \\
\qquad\qquad\qquad\qquad \mid\ \texttt{false}, scl' \rightarrow let\ \texttt{E}, sc' = tc\ sc\ in \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{false}, (scl', sc') \\
\qquad\qquad\qquad\qquad \mid\ \texttt{true}, scl' \rightarrow let\ \texttt{V}(vc), sc' = tc\ sc\ in \\
\qquad\qquad\qquad\qquad\qquad\qquad vc, (scl', sc') \\
\qquad\qquad (icl, ic))
\end{array}
$$

Note that, according to the definition, a clock is an ordinary stream which has no "silent" move.

Merge The converse of `when` whose abstract definition is:

$$\begin{aligned}\texttt{merge}\ (\mathit{false} : \mathit{cs})\ \mathit{xs}\ (\mathit{y} : \mathit{ys}) &= \mathit{y} : \texttt{merge}\ \mathit{cs}\ \mathit{xs}\ \mathit{ys}\\ \texttt{merge}\ (\mathit{true} : \mathit{cs})\ (\mathit{x} : \mathit{xs})\ \mathit{ys} &= \mathit{x} : \texttt{merge}\ \mathit{cs}\ \mathit{xs}\ \mathit{ys}\end{aligned}$$

and whose co-iterative one is:

$$\begin{aligned}&\texttt{merge}\ (\mathit{CoF}(\mathit{tc}, \mathit{ic}))\ (\mathit{CoF}(\mathit{tx}, \mathit{ix}))\ (\mathit{CoF}(\mathit{ty}, \mathit{iy})) =\\ &\quad \mathit{CoF}\ \ \lambda(\mathit{sc}, \mathit{sx}, \mathit{sy}).\\ &\qquad \texttt{match}\ (\mathit{tc}\ \mathit{sc}), (\mathit{tx}\ \mathit{sx}), (\mathit{ty}\ \mathit{sy})\ \texttt{with}\\ &\qquad\ \mid (\texttt{E}, \mathit{sc}'), (\texttt{E}, \mathit{sx}'), (\texttt{E}, \mathit{sy}') \to \texttt{E}, (\mathit{sc}', \mathit{sx}', \mathit{sy}')\\ &\qquad\ \mid (\texttt{V}(\mathit{true}), \mathit{sc}'), (\texttt{V}(\mathit{vx}), \mathit{sx}'), (\texttt{E}, \mathit{sy}') \to \texttt{V}(\mathit{vx}), (\mathit{sc}', \mathit{sx}', \mathit{sy}')\\ &\qquad\ \mid (\texttt{V}(\mathit{false}), \mathit{sc}'), (\texttt{E}, \mathit{sx}'), (\texttt{V}(\mathit{vy}), \mathit{sy}') \to \texttt{V}(\mathit{vy}), (\mathit{sc}', \mathit{sx}', \mathit{sy}')\\ &\qquad (\mathit{ic}, \mathit{ix}, \mathit{iy})\end{aligned}$$

This definition does not raise any execution error if the true branch produces a value when the false branch produces no value and the condition is true, and conversely, the true branch does not produce any value when the false branch produces its value and the condition is false.

Constant This operator is polymorphic in the sense that it may produce or not depending on its environment. For this reason, `const` should have an extra argument giving its clock. We write it:

$$\begin{aligned}\texttt{ClConst}(v)(\mathit{CoF}(\mathit{cl},\mathit{icl})) \quad = \quad \mathit{CoF}\ \ (\lambda \mathit{scl}.\ &\texttt{match}\ \mathit{cl}\ \mathit{scl}\ \texttt{with}\\ &\mid \texttt{true}, s \to (\texttt{V}(v), s)\\ &\mid \texttt{false}, s \to (\texttt{E}, s),\\ &\mathit{icl})\end{aligned}$$

The clock plays an essential role since this is the way to give a deterministic operational semantics to the generator `const`. The clock calculus can infer a clock for the constant so that it does not have to be explicitly passed.

The unit delay We can wonder whether the previous definition for `pre` extends naturally for programs which do not preserve length. Indeed, we could simply write:

$$\begin{array}{rcl} \texttt{pre}(v)((\mathit{CoF}(t,i))) & = & \mathit{CoF}\ \lambda(\mathit{pre},s).\ \texttt{match}\ t\ s\ \texttt{with} \\ & & \quad \mid \texttt{E}, s' \to \texttt{E}, (\mathit{pre}, s') \\ & & \quad \mid \texttt{V}(v), s' \to \texttt{V}(\mathit{pre}), (v, s') \\ & & (v, i) \end{array}$$

Unfortunately, this definition cannot be combined with recursion in a satisfactory way. Running the co-iterative process:

$$\mathit{fix}\,(\lambda x.\texttt{pre}(0)(x))$$

implementing the stream equation:

$$x = \texttt{pre}(0)(x)$$

leads to a deadlock (corresponding to a "stack overflow" in Haskell).

This is due to the fact that the input of `pre` is connected to its output and `pre` emits a value iff its input emits a value. This deadlock can be eliminated by adding an extra argument — an input clock — to `pre` controlling the production. The new definition becomes:

$$
\begin{array}{l}
\texttt{pre}(v)(CoF(cl, icl))(CoF(t, i)) = \\
CoF\ \ \lambda(pre, s, scl).\ \texttt{match}\ cl\ scl\ \texttt{with} \\
\qquad\qquad\qquad\qquad \mid \texttt{false}, scl \rightarrow \texttt{E}, let\ \texttt{E}, s' = t\ s\ in\ (pre, s', scl) \\
\qquad\qquad\qquad\qquad \mid \texttt{true}, scl \rightarrow \texttt{V}(pre), let\ \texttt{V}(v), s' = t\ s\ in\ (v, s', scl) \\
\qquad (v, i, icl)
\end{array}
$$

This time, programs are deadlock free if recursions appear on the right of a `pre` . The use of this new `pre` instead of the previous one is satisfactory if it is possible to built a system inferring the clock. This will be considered later.
The definitions for application, abstraction and recursion remain unchanged.

To conclude

This co-iterative semantics interprets a stream as a process made of an initial state and a step function.

A length preserving function only need its current input to produce its current output.

A fix-point on time is replaced by a fix-point on the instant.

If equations are syntactically guarded by unit delays, no fix point is needed.

In this case, the step function can be expressed in a simple language with `let/in` construct and a call-by-value semantics.

Non length preserving functions are treated by complementing instantaneous values with a explicit "absent".

Non synchrony means that an input is expected to be present but is present (or the converse).

Program must fullfil static type constraint (clock calculus).

- It extends to a richer language (see extra notes).
- Can/how this semantics can replace a relational one for proving some compiler steps?
- Can it be adapted to define a set-based simulation of a program that computes a flowpipe instead of a single trace?
- Can it be used to do *compiler validation* of some compilation steps?

References I

Boulmé, S. and Hamon, G. (2001).

Certifying Synchrony for Free.
In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 2250, La Havana, Cuba. 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.

Bourke, T., Brun, L., Dagand, P., Leroy, X., Pouzet, M., and Rieg, L. (2017).

A formally verified compiler for lustre.
In *PLDI*. ACM.

Bourke, T., Brun, L., and Pouzet, M. (2020).

Mechanized semantics and verified compilation for a dataflow synchronous language with reset.
In *POPL*. ACM.

Boussinot, F. (janvier 1992).

Réseaux de processus réactifs.
Technical Report 1588, INRIA Sophia-Antipolis.

Caspi, P. and Pouzet, M. (1998).

A Co-iterative Characterization of Synchronous Stream Functions.
In *Coalgebraic Methods in Computer Science (CMCS'98)*, Electronic Notes in Theoretical Computer Science.
Extended version available as a VERIMAG tech. report no. 97–07 at www.di.ens.fr/∼pouzet/bib/bib.html.

Colaco, J.-L., Mendler, M., Pauget, B., and Pouzet, M. (2023).

A Constructive State-based Semantics and Interpreter for a Synchronous Data-flow Language with State machines.
In *International Conference on Embedded Software (EMSOFT'23)*, Hamburg, Germany. ACM.

References II

Edward, S. A. and Lee, E. A. (2003).

The semantics and execution of a synchronous block-diagram language.
Science of Computer Programming, 48:21–42.

Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. (1991).

The synchronous dataflow programming language lustre.
Proceedings of the IEEE, 79(9):1305–1320.

Kahn, G. (1974).

The semantics of a simple language for parallel programming.
In *IFIP 74 Congress*. North Holland, Amsterdam.

Kahn, G. and MacQueen, D. B. (1977).

Coroutines and networks of parallel processes.
In *IFIP Congress*, pages 993–998.

Paulin-Mohring, C. (1995).

Circuits as streams in Coq, verification of a sequential multiplier.
Technical report, Laboratoire de l'Informatique du Parallélisme.
Available at `http://www.ens-lyon.fr:80/LIP/lip/publis/`.

Paulin-Mohring, C. (2009).

A constructive denotational semantics for Kahn networks in Coq.
In Bertot, Y., Huet, G., Lévy, J.-J., and Plotki, G., editors, *From Semantics to Computer Science*, pages 383–413. Cambridge University Press.

Reynolds, J. C. (1998).

Theories of Programming Languages.
Cambridge University Press.