Synchronous Programming of Reactive Systems

or

Turning a mathematical model into executable code

Marc Pouzet
ENS/INRIA
Marc.Pouzet@ens.fr

August 28, 2020

ALGO 2020
Program the reactive control software, e.g., that of a plane.
• fly-by-wire ($\approx$ 1.5M LOC)
• real-time
• must not fail (critical)
Write assembly/C/C++/Java/... code by hand?
Write assembly/C/C++/Java/… code by hand?

but what to compare it to?
Write assembly/C/C++/Java/... code by hand?

but what to compare it to?

What is the specification of the system?
And if we expect a mathematically precise specification of the software,
And if we expect a mathematically precise specification of the software,

what language/logic would be more adequate?
And if we expect a mathematically precise specification of the software,

what language/logic would be more adequate?

how to ensure it is correct?
And if we expect a mathematically precise specification of the software,

what language/logic would be more adequate?

how to ensure it is correct?

and the executable code implements it faithfully
And if we expect a mathematically precise specification of the software,

what language/logic would be more adequate?

how to ensure it is correct?

and the executable code implements it faithfully

ensuring the absence of run-time failure?
An idea born from engineering practice in the 80’s
SAO (Spécification Assistée par Ordinateur) — Airbus 80’s
Those drawing are very precise maths

Control engineers described control/command systems with very precise mathematics even before the arrival of computers.

E.g., difference and stream equations, finite state machines, etc.

Example: a linear filter

\[
Y_0 = bX_0 , \quad \forall n \quad Y_{n+1} = aY_n + bX_{n+1}
\]

but they are not executable.

Write code and prove that it is correct.
How to make those maths executable?
node COUNT (init, incr: int; reset: bool)
  returns (n: int);
let
  n = init ->
    if reset then init else pre(n) + incr;
tel;
Data-flow programming

A signal is a stream; a system is a stream function. All streams advance synchronously.

<table>
<thead>
<tr>
<th>X</th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>...</td>
</tr>
<tr>
<td>X + Y</td>
<td>3</td>
<td>6</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>...</td>
</tr>
<tr>
<td>pre X</td>
<td>nil</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>...</td>
</tr>
<tr>
<td>Y -&gt; X</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>...</td>
</tr>
<tr>
<td>C</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>...</td>
</tr>
<tr>
<td>X when C</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>1</td>
<td>2</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>current(Z)</td>
<td>nil</td>
<td>nil</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>...</td>
</tr>
</tbody>
</table>
The synchronous abstraction

- Arithmetic operations (e.g., +) apply pointwise.
- \texttt{pre} is the non initialised unit delay; \texttt{-} the initialization operator.
- \texttt{when} is the sampling operator : it builds a sub-stream from a stream ;
- \texttt{current} is a 0-holder.
- An equation $Z = X + Y$ means $\forall n. Z_n = X_n + Y_n$.

Time is logical : inputs $X$ and $Y$ arrive “at the same time” ; the output $Z$ is produced “at the same time”

Restrict the expressiveness to functions that run in \textit{bounded time and space}. 
An example\textsuperscript{1} : a linear filter

\begin{verbatim}
let node f(static c)(static c')(x) = y where
rec
  y = c * z
and
  z = (0 -> pre y) + c' * x;
\end{verbatim}

$f$ is a stream function such that $y = f(c)(c')(x)$ where $x = (x_n)_{n \in \mathbb{N}}$ and $y = (y_n)_{n \in b\mathbb{N}}$ defined by :

\[(y_n = c.z_n) \land (z_n = y_{n-1} + c'.x_n) \land (z_0 = c'.x_0)\]

$f$ can then be composed with any other stream function.

\textsuperscript{1} Written in Zelus
Some programs are monsters...
that must be statically rejected.
Mixing several time scales
Synchronisation between slow and fast processes.

<table>
<thead>
<tr>
<th>X</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>half</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>...</td>
</tr>
<tr>
<td>X when half</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X + (X when half)</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

let half = true -> not (pre half);
    o = x + (x when half);
tel

It defines the stream: \( \forall n \in \mathbb{N}. o_n = x_n + x_{2n} \)

- It cannot be implemented with bounded memory (buffer).
- It must be rejected statically.
- Express when the value of a stream is present/absent by a type.
- Check synchrony by a type system.
Analyse causality/dependences between signals

Some programs have no solution (*deadlock*) or too many (*non determinacy*)

In Lustre/Signal

- \( x = y + 1 \) and \( y = x + 2 \)
- \( y = x \) and \( x = y \)

In Esterel

- present \( S \) else emit \( S \)
- present \( S1 \) then emit \( S2 \) || present \( S2 \) else emit \( S1 \)

Discovery

A “good” notion of causality is that of *electricity*.

If we build a circuit from a synchronous program and feed it with constant inputs, do the output stabilize?
The beautiful idea of Lustre

Program in a mathematical, precise and executable language.

Analyze/transform/simulate/test/verify.

Automatically translate to executable code.
Many years have passed...
Many years have passed...

The synchronous data-flow model became a standard for implementing critical control software.
Scade/KCG 6 \textsuperscript{[TASE’17]} ²

- A new language in 2008; an original combination of features from Lustre, Lucid Synchrone and Esterel.
- Used in +100 \textit{qualified} projects of critical software (planes, trains).

A focus : what is the semantics of Lustre ?
A focus: what is the semantics of Lustre?

Our purpose is to mechanise it into a proof assistant.

In order to mechanically check the compiler’s correctness.
The two works we used

The (old) work with Paul Caspi, “a Coiterative Characterization of Synchronous Stream Functions” [CP98].

The paper “Circuits as streams in Coq, verification of a sequential multiplier” by Christine Paulin [PM95].
The language kernel

A first-order, Lustre-like kernel.

\[
\begin{align*}
  d &::= \text{let } f = e \mid \text{let node } f \; x = e \mid d \; d \\
  e &::= c \mid x \mid (e,e) \mid f \; e \mid \text{run } f \; e \mid \text{pre}_c(e) \mid e \; \text{fby } e \\
  &\quad \mid \text{fst}(e) \mid \text{snd}(e) \\
  &\quad \mid \text{let } x = e \; \text{in } e \mid \text{let rec } x = e \; \text{in } e \\
  &\quad \mid \text{if } e \; \text{then } e \; \text{else } e \\
  &\quad \mid \text{present } e \; \text{do } e \; \text{else } e \mid \text{reset } e \; \text{every } e
\end{align*}
\]

- \( f \; e \) is the application of a combinatorial function.
- \( \text{run } f \; e \) is the application of a node.
- \( \text{pre}_c(e) \) is the delay initialised with the constant \( c \).
- \( e_1 \rightarrow e_2 \) is a shortcut for \( \text{if } \text{pre}_{\text{true}}(\text{false}) \; \text{then } e_1 \; \text{else } e_2 \).
Semantics
To simplify the presentation, we suppose that expressions are well-typed.
Streams processes

A *stream process* producing values of type $T$ is a pair made of a step function of type $S \rightarrow T \times S$ and an initial state $S$.

$$CoStream(T, S) = CoF(S \rightarrow T \times S, S)$$

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

$$Nth(CoF(f, s))(0) = \text{let } v, s = f \ s \text{ in } v$$
$$Nth(CoF(F, s))(n) = \text{let } v, s = f \ s \text{ in } Nth(CoF(f, s))(n - 1)$$

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

$$\forall n \in \mathbb{N}. Nth(CoF(f, s))(n) = Nth(CoF(f', s'))(n)$$
Synchronous Stream Processes

A stream function should be a value from:

\[ \text{CoStream}(T, S) \rightarrow \text{CoStream}(T', S') \]

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

A *synchronous stream function*, from inputs of type $T$ to outputs of type $T'$ is a pair, made of a step function and an initial state.

\[
\text{type } \text{SFun}(T, T', S) = \text{CoP}(S \rightarrow T \rightarrow T' \times S, S)
\]

It only needs the current value of its input in order to compute the current value of its output.

Remark that $s : \text{CoStream}(T, S)$ can be represented by a value of type $\text{SFun}(\text{Unit}, T, S)$ with $\text{Unit}$ the type containing a single value (\(\)).
Feedback loop/Fixpoint

Consider a synchronous stream function \( f : S \rightarrow T \rightarrow T \times S \). We want to define the equation (or feedback loop) such that:

\[
v, s' = f s v
\]

Given \( f \), we want \( \text{fix}(f)(s) = v, s' \) with \( \text{fix}(f) : S \rightarrow T \times S \) for the smallest fix-point of \( f \).

Given an initial state \( s : S \), \( \text{fix}(f) \) must be a solution of:

\[
X(s) = \text{let } v, s' = X(s) \text{ in } f s v
\]

This fix-point can be implemented with a recursion on values, for example in Haskell:

\[
\text{fix}(f) = \lambda s. \text{let rec } v, s' = f s v \text{ in } v, s'
\]

The value \( v \) is defined recursively.
Justification of its existence

To apply the Kleene theorem that state the existence of a smallest fix-point, we first make all functions total.

\[
Value(T) = \bot + \text{Some}(T)
\]

\(\bot\) is a short-cut for “Causality Error” or “Deadlock”.

Define lifting functions.

\[
\begin{align*}
\text{lift}_0(v) &= \text{Some}(v) \\
\text{lift}_1(f)(\bot) &= \bot \\
\text{lift}_1(f)(\text{Some}(v)) &= \text{Some}(f(v)) \\
\text{lift}_2(f)(\bot, y) &= \bot \\
\text{lift}_2(f)(x, \bot) &= \bot \\
\text{lift}_2(f)(\text{Some}(v_1), \text{Some}(v_2)) &= \text{Some}(f(v_1)(v_2))
\end{align*}
\]

That is, \(\bot\) is absorbing and all functions applied point-wise are total.
Flat Order

Define \( \leq_T \subseteq (\text{Value}(T) \times \text{Value}(T)) \) such that:

\[
\bot \leq_T \times \text{Some}(v) \leq_T \text{Some}(v)
\]

Shortcut: we write simply \( \leq \).

Pairs:

\( (v_1, v_2) \leq (v'_1, v'_2) \) iff \( v_1 \leq v'_1 \) \( \land \) \( v_2 \leq v'_2 \)
The bottom stream

The bottom element is:

$$\text{CoF}((\lambda s. (\perp, s)), \perp) : \text{CoStream}(\text{Value}(T), \text{Value}(S))$$

Call $\perp_{\text{CoStream}}$ or simply $\perp$, this bottom stream element.

It corresponds to a stream process that stuck: giving an input state, it returns the bottom value.

Define $\leq_{\text{CoStream}}$ such that (noted $\leq$):

$$\text{CoF}(f, s) \leq \text{CoF}(f', s') \iff (s \leq s') \land (\forall s. (f \ s) \leq (f' \ s))$$

Define $\leq_{\text{SFun}}$ such that (noted $\leq$):

$$\text{CoP}(f, s) \leq \text{CoP}(f', s') \iff (s \leq s') \land (\forall s, x : (f \ s \ x) \leq (f' \ s \ x))$$

If $f : \text{SFun}(\text{Value}(T), \text{Value}(T), \text{Value}(S))$ is continuous, $\text{fix}(f)$ exists.
Bounded Fixpoint

How can we define/program the fix-point? It cannot be defined as a function in Coq.

A trick. Define the bounded iteration $\text{fix } (f)(n)$ as:

$$
\begin{align*}
\text{fix } (f)(0)(s) &= \bot, s \\
\text{fix } (f)(n)(s) &= \text{let } v, s' = \text{fix } (f)(n-1)(s) \text{ in } f s v
\end{align*}
$$

Suppose that $f x : \text{CoStream}(T, S)$. Compute $\| T \|$ such that:

$$
\begin{align*}
\| bt \| &= 1 \\
\| \alpha \| &= 1 \\
\| t_1 \times t_2 \| &= \| t_1 \| + \| t_2 \|
\end{align*}
$$

Give only a credit of $\| T \| + 1$ iterations for a fix-point on a value of type $T$. 
The semantics of an expression $e$ is:

$$\llbracket e \rrbracket_{\rho} = CoF(f, s) \text{ where } f = \llbracket e \rrbracket_{\rho}^{State} \text{ and } s = \llbracket e \rrbracket_{\rho}^{Init}$$

We use two auxiliary functions.

- $\llbracket e \rrbracket_{\rho}^{Init}$ is the initial state of the transition function associated to $e$;
- $\llbracket e \rrbracket_{\rho}^{State}$ is the step function.

$\rho$ map values to identifiers.
\[
\begin{align*}
\llbracket \text{pre}_c(e) \rrbracket^\text{Init}_\rho & = (c, \llbracket e \rrbracket^\text{Init}_\rho) \\
\llbracket \text{pre}_c(e) \rrbracket^\text{State}_\rho & = \lambda (m, s). m, \llbracket e \rrbracket^\text{State}_\rho(s) \\
\llbracket f e \rrbracket^\text{Init}_\rho & = \llbracket e \rrbracket^\text{Init}_\rho \\
\llbracket f e \rrbracket^\text{State}_\rho & = \lambda s. \text{let } v, s = \llbracket e \rrbracket^\text{State}_\rho(s) \text{ in } f(v), s \\
\llbracket x \rrbracket^\text{Init}_\rho & = () \\
\llbracket x \rrbracket^\text{State}_\rho & = \lambda s. (\rho(x), s) \\
\llbracket c \rrbracket^\text{Init}_\rho & = () \\
\llbracket c \rrbracket^\text{State}_\rho & = \lambda s. (c, s) \\
\llbracket (e_1, e_2) \rrbracket^\text{Init}_\rho & = (\llbracket e_1 \rrbracket^\text{Init}_\rho, \llbracket e_2 \rrbracket^\text{Init}_\rho) \\
\llbracket (e_1, e_2) \rrbracket^\text{State}_\rho & = \lambda (s_1, s_2). \text{let } v_1, s_1 = \llbracket e_1 \rrbracket^\text{State}_\rho(s_1) \text{ in} \\
& \quad \text{let } v_2, s_2 = \llbracket e_2 \rrbracket^\text{State}_\rho(s_2) \text{ in} \\
& \quad (v_1, v_2), (s_1, s_2)
\end{align*}
\]
\[\begin{align*}
[\text{run } f \ e]^{\text{Init}}_{\rho} &= f_s, [e]^{\text{Init}}_{\rho} \\
[\text{run } f \ e]^{\text{State}}_{\rho} &= \lambda(m, s).\text{let } v, s = [e]^{\text{State}}_{\rho}(s) \text{ in} \\
&\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{let } r, m' = f_t m v \text{ in} \\
&\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad r, (m', s) \\
\text{where } \rho(f) &= \text{CoP}(f_t, f_s) \\
\end{align*}\]

\[\begin{align*}
[\text{let node } f \ x = e]^{\text{Init}}_{\rho} &= \rho + [\text{CoP}(p, s)/f] \\
\text{such that } s &= [e]^{\text{Init}}_{\rho} \\
\text{and } p &= \lambda s, v. [e]^{\text{State}}_{\rho+[v/x]}(s)
\end{align*}\]
Using a recursion on value, it corresponds to:

\[
\begin{align*}
\llbracket \text{let rec } x = e \text{ in } e' \rrbracket^{\text{State}}_{\rho} &= \lambda(s, s').\text{let rec } v, ns = \llbracket e \rrbracket^{\text{State}}_{\rho + [v/x]}(s) \text{ in } \\
&\quad \text{let } v', s' = \llbracket e' \rrbracket^{\text{State}}_{\rho + [v/x]}(s') \text{ in } \\
&\quad v', (ns, s')
\end{align*}
\]

Note that \(v\) is recursively defined
Control structure

\[
\text{[if } e \text{ then } e_1 \text{ else } e_2\text{]}^{\text{Init}}_{\rho} = (\llbracket e \rrbracket_{\rho}^{\text{Init}}, \llbracket e_1 \rrbracket_{\rho}^{\text{Init}}, \llbracket e_2 \rrbracket_{\rho}^{\text{Init}})
\]

\[
\text{[if } e \text{ then } e_1 \text{ else } e_2\text{]}^{\text{State}}_{\rho} = \lambda(s, s_1, s_2).\text{let } v, s = \llbracket e \rrbracket_{\rho}^{\text{State}}(s) \text{ in}
\]

\[
\text{let } v_1, s_1 = \llbracket e_1 \rrbracket_{\rho}^{\text{State}}(s_1) \text{ in}
\]

\[
\text{let } v_2, s_2 = \llbracket e_2 \rrbracket_{\rho}^{\text{State}}(s_2) \text{ in}
\]

\[
(\text{if } v \text{ then } v_1 \text{ else } v_2, (s, s_1, s_2))
\]

\[
\text{[present } e \text{ do } e_1 \text{ else } e_2\text{]}^{\text{Init}}_{\rho} = (\llbracket e \rrbracket_{\rho}^{\text{Init}}, \llbracket e_1 \rrbracket_{\rho}^{\text{Init}}, \llbracket e_2 \rrbracket_{\rho}^{\text{Init}})
\]

\[
\text{[present } e \text{ do } e_1 \text{ else } e_2\text{]}^{\text{State}}_{\rho} = \lambda(s, s_1, s_2).\text{let } v, s = \llbracket e \rrbracket_{\rho}^{\text{State}}(s) \text{ in}
\]

\[
\text{if } v \text{ then let } v_1, s_1 = \llbracket e_1 \rrbracket_{\rho}^{\text{State}}(s_1) \text{ in}
\]

\[
v_1, (s, s_1, s_2)
\]

\[
\text{else let } v_2, s_2 = \llbracket e_2 \rrbracket_{\rho}^{\text{State}}(s_2) \text{ in}
\]

\[
v_2, (s, s_1, s_2)
\]

The “if/then/else” always executes its arguments but not the “present” :
### Modular Reset

Reset a computation when a boolean condition is true.

\[
\begin{align*}
\left[ \text{reset } e_1 \text{ every } e_2 \right]_{\rho}^{\text{Init}} &= \left( \left[ e_1 \right]_{\rho}^{\text{Init}}, \left[ e_1 \right]_{\rho}^{\text{Init}}, \left[ e_2 \right]_{\rho}^{\text{Init}} \right) \\
\left[ \text{reset } e_1 \text{ every } e_2 \right]_{\rho}^{\text{State}} &= \lambda(s_i, s_1, s_2).
\end{align*}
\]

\[
\begin{align*}
\text{let } v_2, s_2 &= \left[ e_2 \right]_{\rho}^{\text{State}}(s_2) \text{ in} \\
\text{let } v_1, s_1 &= \left[ e_1 \right]_{\rho}^{\text{State}}(\text{if } v_2 \text{ then } s_i \text{ else } s_1) \text{ in} \\
v_1, (s_i, s_1, s_2)
\end{align*}
\]

This definition duplicates the initial state. An alternative is:

\[
\begin{align*}
\left[ \text{reset } e_1 \text{ every } e_2 \right]_{\rho}^{\text{Init}} &= \left( \left[ e_1 \right]_{\rho}^{\text{Init}}, \left[ e_2 \right]_{\rho}^{\text{Init}} \right) \\
\left[ \text{reset } e_1 \text{ every } e_2 \right]_{\rho}^{\text{State}} &= \lambda(s_1, s_2).
\end{align*}
\]

\[
\begin{align*}
\text{let } v_2, s_2 &= \left[ e_2 \right]_{\rho}^{\text{State}}(s_2) \text{ in} \\
\text{let } s_1 &= \text{if } v_2 \text{ then } \left[ e_1 \right]_{\rho}^{\text{Init}} \text{ else } s_1 \text{ in} \\
\text{let } v_1, s_1 &= \left[ e_1 \right]_{\rho}^{\text{State}}(s_1) \text{ in} \\
v_1, (s_1, s_2)
\end{align*}
\]
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:

\[ e ::= \text{let rec } \& \& x = e \ldots x = e \text{ in } e \]

is interpreted as the definition of a single recursive definition such that:

\[ \text{let rec } \& \& x_1 = e_1 \ldots x_n = e_n \text{ in } e \]

means:

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

with:

\[
\begin{align*}
e'_1 &= \text{fst}(x) \\
e'_2 &= \text{fst}(\text{snd}(x)) \\
\vdots & \\
e'_n &= \text{snd}^{n-1}(x)
\end{align*}
\]
Where are the bottom values?
Examples

Some equations have the constant bottom stream as minimal fix-point.

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

Indeed:

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

Or:

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

Indeed:

$$fix(\lambda s, v.\left[ (\text{snd}(v), \text{fst}(v)) \right]_{\rho + [v/x]}^{State}(s)) = fix(\lambda s, v.(\text{snd}(v), \text{fst}(v)), s)$$
$$= \lambda s.(\bot, \bot), s$$
Def-use chains

The two previous examples have an instantaneous feedback.

Some functions are “strict”, i.e., a function $g$ such that $\text{fst}(g \, s \, \bot) = \bot$.

Some are not, e.g.:

```ml
let node mypre(x) = 1 + (0 fby (x+2)
```

Its semantics is $\text{CoP}(f, 0)$ with:

$$f = \lambda s, x. (1 + s, x + 2)$$

Hence $\text{fst}(f \, s \, \bot) = 1 + s$, that is, $\bot < \text{fst}(f \, s \, \bot)$

$f$ is strictly increasing.

Build a dependence relation from the call graph. If this graph is cyclic, reject the fix-point definition.
What is really a dependence? How modular is it?

The notion of dependence is subtle. All function below are such that if \( x \) is non bottom, outputs \( z \) and \( t \) are non bottom. Do we want to accept them and how?

```plaintext
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)

let node f(y) = x where
    rec x = if false then x else 0
```
The following is a classical example that is “constructively causal” but is rejected by Lustre and Zelus compilers.

```plaintext
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 $(x_1, x_2, y_1, y_2, y)$ is a non bottom stream.

It can be proved to be equivalent to:

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

Question : is the semantics enough to prove they are equivalent? How?
The following example also defines a node whose output is non bottom:

```plaintext
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:

```plaintext
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 \( c_1, c_2, y \) are not bottom values, taking true for \( c_1 \) and \( c_2 \), for example.

Starting with \( x_0 = \perp, z_0 = \perp, t_0 = \perp \) and \( r_0 = \perp \), the fixpoint is the limit of the sequence:

\[
x_n = y + 1 \land z_n = t_{n-1} + 1 \land t_n = x_{n-1} + 1 \land r_n = z_{n-1} + 2
\]

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 impact on 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

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

The semantics is executable, lazilly or by computing fix point iteratively.

Some recursive equations can be translated into non recursive definitions.

Consider the stream equation :

\[
\text{let rec } \text{nat} = 0 \ \text{fby} \ (\text{nat} + 1) \ \text{in} \ \text{nat}
\]

Can we get rid of recursion in this definition? Surely yes. Its stream process is :

\[
nat = \text{Co}(\lambda s. (s, s + 1), 0)
\]
First: let us unfold the semantics

Consider the recursive equation:

\[
\text{rec } x = (0 \text{ fby } x) + 1
\]

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 \( f : S \rightarrow X \times S \) and \( s : S \) the initial state.

Write \( x\).\text{step} for \( f \) and \( x\).\text{init} for \( x \). \text{init} for \( s \).
The equation that defines nat can be rewritten as

\[ \text{let rec } \text{nat} = f(\text{nat}) \text{ in } \text{nat} \text{ with let node } f \ x = (0 \text{ fby } x) + 1. \]

The semantics of \( f \) is:

\[ f = \text{CoP}(f_s, s_0) = \text{CoP}(\lambda s, x.(s + 1, x), 0) \]

Solving \( \text{nat} = f(\text{nat}) \) amount at finding a stream \( X \) such that:

\[ X(s) = \text{let } v, s' = X(s) \text{ in } f_s \ s \ v \]

The bottom stream, to start with, is:

\[ x^0 = \text{CoF}(\lambda s.(\bot, s), \bot) \]
Let us proceed iteratively by unfolding the definition of the semantics. We have:

\[
\begin{align*}
    x^1.\text{step} & = \lambda s.\text{let } v, s' = x^0.\text{step } s \text{ in } f_s \ s \ v \\
                     & = \lambda s. f_s \ s \ \bot \\
                     & = \lambda s. s + 1, \bot \\
    x^1.\text{init} & = 0
\end{align*}
\]

\[
\begin{align*}
    x^2.\text{step} & = \lambda s.\text{let } v, s' = x^1.\text{step } s \text{ in } f_s \ s \ v \\
                     & = \lambda s.\text{let } v = s + 1 \text{ in } f_s \ s \ v \\
                     & = \lambda s.\text{let } v = s + 1 \text{ in } s + 1, v \\
                     & = \lambda s. s + 1, s + 1 \\
    x^2.\text{init} & = 0
\end{align*}
\]

\[
\begin{align*}
    x^3.\text{step} & = \lambda s.\text{let } v, s' = x^2.\text{step } s \text{ in } f_s \ s \ v \\
                     & = \lambda s.\text{let } v = s + 1 \text{ in } f_s \ s \ v \\
                     & = \lambda s.\text{let } v = s + 1 \text{ in } s + 1, v \\
                     & = \lambda s. s + 1, s + 1 \\
    x^3.\text{init} & = 0
\end{align*}
\]

We have reached the fix-point $CoF(\lambda s.(s + 1, s + 1), 0)$ in three steps.
A simple, syntactic, condition under which the semantics of mutually recursive stream equations does not need any fix point.

Consider a node $f : CoStream(T, S) \to CoStream(T, S')$ whose semantics is $CoP(f_t, s_t)$.

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

\[
\begin{align*}
\left[\text{let rec } y = f(y) \text{ in } y\right]^{\text{Init}}_{\rho} &= s_t \\
\left[\text{let rec } y = f(y) \text{ in } y\right]^{\text{State}}_{\rho} &= \lambda s.\text{let rec } v, s' = f_t s v \text{ in } v, s'
\end{align*}
\]

3. We reason upto bisimulation, that is, independently on the actual representation of the internal state.
Two cases can happen:

- Either $f_t$ is strictly increasing and the evaluation succeeds.
- or there is an instantaneous loop.
When $f_t s \nu$ does not need $\nu$ to return the value part, the recursive evaluation of the pair $\nu, s'$ can be split into two non recursive definitions.

This case appears, for example, when every stream recursion appears on the right of a unit delay $\text{pre}$.

A synchronous compiler takes advantage of this in order to produce non recursive code like the co-iterative $\text{nat}$ expression given above.
For example, consider the equation \( y = f(v \ fby \ x) \). Its semantics is:

\[
\begin{align*}
&\left[ \text{let rec } x = f(v \ fby \ x) \text{ in } x \right]_{\rho}^{\text{Init}} = (v, s_t) \\
&\left[ \text{let rec } x = f(v \ fby \ x) \text{ in } x \right]_{\rho}^{\text{State}}(m, s) = \text{let rec } v, s' = f_t s m \text{ in } v, (v, s')
\end{align*}
\]

The recursion is no more necessary, that is:

\[
\begin{align*}
&\left[ \text{let rec } x = f(v \ fby \ x) \text{ in } x \right]_{\rho}^{\text{State}}(m, s) = \text{let } v, s' = f_t s m \text{ in } v, (v, s')
\end{align*}
\]
The Semantics for Normalised Equations

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

\[
\text{let rec } \quad x_1 = v_1 \text{ fby } n x_1 \\
\text{ and } \ldots \\
\text{ and } x_n = v_n \text{ fby } n x_n \\
\text{ and } p_1 = e_1 \\
\text{ and } \ldots \\
\text{ and } p_k = e_k \\
\text{ in } e
\]

where

\[
\forall i, j. (i < j) \implies \text{Var}(e_i) \cap \text{Var}(p_j) = \emptyset
\]

where \(\text{Var}(p)\) and \(\text{Var}(e)\) are the set of variable names appearing in \(p\) and \(e\).
Its transition function is:

\[ \lambda(x_1, \ldots, x_n, s_1, \ldots, s_k, s). \]

\[
\text{let } p_1, s_1 = \llbracket e_1 \rrbracket^\text{State}_\rho(s_1) \text{ in } \\
\text{let } \ldots \text{ in } \\
\text{let } p_k, s_k = \llbracket e_k \rrbracket^\text{State}_\rho(s_k) \text{ in } \\
\text{let } r, s = \llbracket e \rrbracket^\text{State}_\rho(s) \text{ in } \\
r, (nx_1, \ldots, nx_n, s_1, \ldots, s_k, s)
\]

with initial state:

\[(v_1, \ldots, v_n, s_1, \ldots, s_k, s)\]

if \(\llbracket e_i \rrbracket^\text{Init}_\rho = s_i\) and \(\llbracket e \rrbracket^\text{Init}_\rho = s\).

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.

**Question:** Is the semantics adequate to prove correctness of this variant semantics for fix-points?
The Complete Language

This semantics extends to a richer language: local definitions, activation conditions, hierarchical automata.

Causality typing

A type system which summarizes the input/output dependences. The one of Zelus expresses input/output relations [BBC+14].

1. Outputs are non bottom, provided inputs are non bottom.

2. Generate statically scheduled code, a function that works with values of type $T$, not $Value(T)$. 
Non length preserving functions [CP98]

\[
\begin{align*}
    CLValue(T) &= E + V(T) \\
    CLStream(T, S) &= CoStream(CLValue(T), S)
\end{align*}
\]

Add \( \bot \) as “Clocking error”. When a program is well clocked, it does not generate a value \( \bot \).

Higher-order stream functions

Deal with Zelus functions like the following one.

```ocaml
define node pid(int)(derivative)(p, i, d, u) = po +. io +. ddo
    where rec po = p *. u
    and io = run int (i *. u)
    and ddo = run derivative (d *. u)
```
This is on-going work

A comprehensive semantics for Scade can be built this way.
An interpreter in OCaml has been written (this spring).
To be done: an integration to Velus\textsuperscript{4} the formally verified Lustre compiler in Coq.

\textsuperscript{4} https://velus.inria.fr
Albert Benveniste, Timothy Bourke, Benoit Caillaud, Bruno Pagano, and Marc Pouzet.
A Type-based Analysis of Causality Loops in Hybrid Systems Modelers.
In *International Conference on Hybrid Systems: Computation and Control (HSCC)*, Berlin, Germany, April 15–17 2014. ACM.

Paul Caspi and Marc Pouzet.
A Co-iterative Characterization of Synchronous Stream Functions.

Jean-Louis Colaco, Bruno Pagano, and Marc Pouzet.

Christine Paulin-Mohring.
Circuits as streams in Coq, verification of a sequential multiplier.

Marc Pouzet.
Université Paris-Sud, LRI, April 2006.
Distribution available at: https://www.di.ens.fr/~pouzet/lucid-synchrone/.