A Coiterative Synchronous Semantics (Work in progress)

Marc Pouzet

Modeliscale IPL
October 15, 2020

This is a joint work with Guillaume Baudart, Jean-Louis Colaco, Louis Mandel and Michael Mendler.

This work was initiated in June 2019 at Bamberg Univ. while preparing a class on synchronous programming and working in parallel with Guillaume Baudart and Louis Mandel on the semantics of ProbZelus [BMA+20].

A preliminary version was presented at SYNCHRON, Dec. 2019.

A first prototype is implemented in (purely functional) OCaml.

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

No publication yet. All comments/questions/criticism is welcome !

Our purpose

- Define a reference and executable semantics for a synchronous language which comprise modern programming constructs, e.g., the program :

  `https://github.com/marcpouzet/zrun/blob/master/tests/watch_in_scade.scade`.

- Before any compilation step or static verification (typing).
- That leads to a reference interpreter.
- Test a compiler ; execute unfinished programs or program that do not pass static checks, for debugging purposes.
- To prototype and experiment with new language features.
- If possible, useful to prove properties on programs or compilation steps, e.g., type correctness, program transformations.

- One motivation is not personal (asked by Scade users) : to be able to execute Scade models on the fly, in the graphical interface, for debug/design.
- One is an old quest : have an executable semantics for Lustre and its relative (Lucid Synchrone, Scade, etc.) to clarify the difference between Esterel causality and Lustre causality.
- Possibly treat all Zelus with a semantics that is parameterized by an ODE/zero-crossing solver and keep it functional.

The two works we used

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

The (wonderful) paper "Circuits as streams in Coq, verification of a sequential multiplier" by Christine Paulin [PM95].

The language kernel

A first-order, Lustre-like kernel.

$$
\begin{array}{rcl}
d & ::= & \texttt{let}\ f = e \mid \texttt{let node}\ f\ x = e \mid d\ d \\
e & ::= & c \mid x \mid (e, e) \mid f\ e \mid \texttt{run}\ f\ e \mid \texttt{pre}_c(e) \mid e\ \texttt{fby}\ e \\
 & & \mid \texttt{fst}(e) \mid \texttt{snd}(e) \\
 & & \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{let rec}\ x = e\ \texttt{in}\ e \\
 & & \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
 & & \mid \texttt{present}\ e\ \texttt{do}\ e\ \texttt{else}\ e \mid \texttt{reset}\ e\ \texttt{every}\ e
\end{array}
$$

- $f\ e$ is the application of a combinatorial function.
- $\texttt{run}\ f\ e$ is the application of a node.
- $\texttt{pre}_c(e)$ is the delay initialised with the constant c.
- e_1 -> e_2 is a shortcut for *if* $\texttt{true fby false}$ *then* e_1 *else* e_2

Semantics

Stream processes

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

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

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

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

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

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

Synchronous Stream Processes

A stream function should be a value from :

$$CoStream(T, S) \rightarrow CoStream(T', S')$$

Let us consider a simpler class of stream functions that are length functions or synchronous.

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

$$type\ SFun(T, T', S) = CoP(S \rightarrow T \rightarrow T' \times S, S)$$

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

Remark that $s : CoStream(T, S)$ can be represented by a value of type $SFun(Unit, T, S)$ with $Unit$ the type containing a single value `()`.

Feedback loop/Fixpoint

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

$$v, s' = f\ s\ v$$

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

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

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

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

$$fix\,(f) = \lambda s.let\ rec\ v, s' = f\ s\ v\ in\ v, s'$$

The value v is defined recursively. Yet, $fix\,(f)$ may not be defined for all f.

Justification of its existence

To make function total, complete the set of values with a special value $T_\perp = T + \perp$. We model this set by a data-type :

$$\mathit{Value}(T) = \mathtt{Bot} + \mathtt{V}(T)$$

$\perp$ is a short-cut for "Causality Error" or "Deadlock".

with associated lifting functions.

$$\begin{array}{lcl}
\mathit{lift}_0(v) & = & \mathtt{V}(v) \\
\mathit{lift}_1(f)(\mathtt{Bot}) & = & \mathtt{Bot} \\
\mathit{lift}_1(f)(\mathtt{V}(v)) & = & \mathtt{V}(f(v)) \\
\mathit{lift}_2(f)(\mathtt{Bot}, y) & = & \mathtt{Bot} \\
\mathit{lift}_2(f)(x, \mathtt{Bot}) & = & \mathtt{Bot} \\
\mathit{lift}_2(f)(\mathtt{V}(v_1), \mathtt{V}(v_2)) & = & \mathtt{V}(f(v_1)(v_2))
\end{array}$$

That is, `Bot` is absorbing and all functions applied point-wise are total.

Flat Order

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

$$\begin{array}{lll} \texttt{Bot} & \leq_T & x \\ \texttt{V}(v) & \leq_T & \texttt{V}(v) \end{array}$$

Shortcut : we write simply $\leq$.

Pairs :

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

The bottom stream

The bottom stream is :

$$CoF((\lambda s.(\bot, s)), \bot) : CoStream(Value(T), Value(S))$$

Call $\bot_{CoStream}$ or simply $\bot$, this *bottom stream* element.

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

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

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

Bounded Fixpoint

How can we define/program the fix-point ? It cannot be defined as a total function without hypothesis on its argument.

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

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

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

$$\begin{array}{lcl} \|\mathtt{int}\| & = & 1 \\ \|t_1 \times t_2\| & = & \|t_1\| + \|t_2\| \end{array}$$

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 :

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

We use two auxiliary functions.

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

ρ map values to identifiers.

$$
\begin{array}{lcl}
[\![\mathtt{pre}_c(e)]\!]_\rho^{Init} & = & (c, [\![e]\!]_\rho^{Init}) \\
[\![\mathtt{pre}_c(e)]\!]_\rho^{State} & = & \lambda(m,s).m, [\![e]\!]_\rho^{State}(s) \\
[\![f\ e]\!]_\rho^{Init} & = & [\![e]\!]_\rho^{Init} \\
[\![f\ e]\!]_\rho^{State} & = & \lambda s.\mathit{let}\ v, s = [\![e]\!]_\rho^{State}(s)\ \mathit{in}\ f(v), s \\
[\![x]\!]_\rho^{Init} & = & () \\
[\![x]\!]_\rho^{State} & = & \lambda s.(\rho(x), s) \\
[\![c]\!]_\rho^{Init} & = & () \\
[\![c]\!]_\rho^{State} & = & \lambda s.(c, s) \\
[\![(e_1, e_2)]\!]_\rho^{Init} & = & ([\![e_1]\!]_\rho^{Init}, [\![e_2]\!]_\rho^{Init}) \\
[\![(e_1, e_2)]\!]_\rho^{State} & = & \lambda(s_1, s_2).\mathit{let}\ v_1, s_1 = [\![e_1]\!]_\rho^{State}(s_1)\ \mathit{in} \\
 & & \quad \mathit{let}\ v_2, s_2 = [\![e_2]\!]_\rho^{State}(s_2)\ \mathit{in} \\
 & & \quad (v_1, v_2), (s_1, s_2)
\end{array}
$$

$$
\begin{array}{lll}
[\![\texttt{run}\ f\ e]\!]^{Init}_{\rho} & = & f_s, [\![e]\!]^{Init}_{\rho} \\
[\![\texttt{run}\ f\ e]\!]^{State}_{\rho} & = & \lambda(m,s).\mathit{let}\ v, s = [\![e]\!]^{State}_{\rho}(s)\ \mathit{in} \\
 & & \quad \mathit{let}\ r, m' = f_t\ m\ v\ \mathit{in} \\
 & & \quad r, (m', s) \\
 & \mathit{where} & \rho(f) = \mathit{CoP}(f_t, f_s)
\end{array}
$$

$$
\begin{array}{lll}
[\![\texttt{let node}\ f\ x = e]\!]^{Init}_{\rho} & = & \rho + [\mathit{CoP}(p,s)/f] \\
 & & \text{such that } s = [\![e]\!]^{Init}_{\rho} \\
 & & \text{and } p = \lambda s, v.[\![e]\!]^{State}_{\rho+[v/x]}(s)
\end{array}
$$

Fixpoint

$$
\begin{array}{lcl}
[\![\texttt{let rec } x = e \text{ in } e']\!]^{Init}_{\rho} & = & [\![e]\!]^{Init}_{\rho}, [\![e']\!]^{Init}_{\rho} \\
[\![\texttt{let rec } x = e \text{ in } e']\!]^{State}_{\rho} & = & \lambda(s, s').\mathit{let}\ v, s = \mathit{fix}\ (\lambda s, v.[\![e]\!]^{State}_{\rho+[v/x]}(s))\ \mathit{in} \\
 & & \quad \mathit{let}\ v', s' = [\![e']\!]^{State}_{\rho+[v/x]}(s')\ \mathit{in} \\
 & & \quad v', (s, s')
\end{array}
$$

Using a recursion on value, it corresponds to :

$$
\begin{array}{lcl}
[\![\texttt{let rec } x = e \text{ in } e']\!]^{State}_{\rho} & = & \lambda(s, s').\mathit{let\ rec}\ v, ns = [\![e]\!]^{State}_{\rho+[v/x]}(s)\ \mathit{in} \\
 & & \quad \mathit{let}\ v', s' = [\![e']\!]^{State}_{\rho+[v/x]}(s')\ \mathit{in} \\
 & & \quad v', (ns, s')
\end{array}
$$

Note that v is recursively defined

Control structure

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

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

The "if/then/else" always executes its arguments but not the "present" :

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 :

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

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* :

$$e \quad ::= \quad \texttt{let rec} \quad \& \quad \& \quad x = e ... x = e \text{ in } e$$

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

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

with :

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

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 :

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

Or :

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

Indeed :

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

Def-use chains

The two previous examples have an instantaneous feedback.

Some functions are "strict", i.e., a function g such that $fst(g\, s\, \bot) = \bot$.

Some are not, e.g. :

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

Its semantics is $CoP(f, 0)$ with :

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

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

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 ?

```
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.

```
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 y = mux(c, g(f(x)), f(g(x)))
```

Question : is the semantics enough to prove they are equivalent ? 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.

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

The following example 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, y are not bottom values, taking true for c1 and c2, for example.

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 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 :

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

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

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

First : let us unfold the semantics

Consider the recursive equation :

```
rec x = (0 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 $f : S \to X \times S$ and $s : S$ the initial state.

Write $x.step$ for f and $x.init$ for $x : init$ for s.

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 :

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

The bottom stream, to start with, is :

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

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

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

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

The semantics of an equation $y = f(y)$ is :[1]

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

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

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

Two cases can happen :

- Either $f_t\, s$ is strictly increasing and the evaluation succeeds.
- or there is an instantaneous loop.

When $f_t\, s\, v$ does not need v to return the value part, the recursive evaluation of the pair v, 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 `pre`.

A synchronous compiler takes advantage of this in order to produce non recursive code like the co-iterative *nat* expression given above.

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

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

The recursion is no more necessary, that is :

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

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 :

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

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 : Is the semantics adequate to prove correctness of this variant semantics for fix-points ?

Next

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) Ouputs 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]

$$\begin{aligned} \mathit{CLValue}(T) &= \mathtt{E} + \mathtt{V}(T) \\ \mathit{CLStream}(T, S) &= \mathit{CoStream}(\mathit{CLValue}(T), S) \end{aligned}$$

Add $\perp$ as "Clocking error". When a program is well clocked, it does not generate a value $\perp$.

Higher-order stream functions

Deal with Zelus functions like the following one.

```
let 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 interpretor in OCaml has been written (this spring).

The semantics is also defined for the automata of Zelus. They are a bit more expressive than that of Scade. In particular, states can be parameterized.

Give the semantics for ODEs and zero-crossing by making the semantics parameterized by an ODE solver and zero-crossing solver.

Is this work useful for proving compiler steps and be integrated to Velus[2] ?

2. `https://velus.inria.fr`

References I

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.

Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet.

A Hybrid Synchronous Language with Hierarchical Automata : Static Typing and Translation to Synchronous Code.
In *ACM SIGPLAN/SIGBED Conference on Embedded Software (EMSOFT'11)*, Taipei, Taiwan, October 2011.

Timothy Bourke, Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet.

A Synchronous-based Code Generator For Explicit Hybrid Systems Languages.
In *International Conference on Compiler Construction (CC)*, LNCS, London, UK, April 11-18 2015.

Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin.

Reactive Probabilistic Programming.
In *International Conference on Programming Language Design and Implementation (PLDI)*, London, United Kingdom, June 15-20 2020. ACM.

Paul Caspi and Marc Pouzet.

A Co-iterative Characterization of Synchronous Stream Functions.
In *Coalgebraic Methods in Computer Science (CMCS'98)*, Electronic Notes in Theoretical Computer Science, March 1998.
Extended version available as a VERIMAG tech. report no. 97–07 at `www.di.ens.fr/~pouzet/bib/bib.html`.

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/`.