A Coiterative Semantics for Data-flow Hybrid Systems (work in progress)

Marc Pouzet

Ecole normale supérieure
Marc.Pouzet@ens.fr

Realize
February 5, 2021

Motivation

Objective

- To give a reference semantics for Zelus,
- that include both discrete and continuous-time signals/systems;
- executable (operational), the basis of an interpreter;
- for compiler testing and the proof of compiler steps.

Point of view

- a two level approach: a kernel synchronous language on top of which other constructs are added.
- functiorial, taking the ODE/zero-crossing solvers as a parameter.
- Focus on data-flow (functional) hybrid models, not relational (implicit) hybrid models, that is, Simulink not Modelica.

The language kernel

A first-order subset of Zelus.

$$
\begin{array}{lcl}
d & ::= & \texttt{let}\ f = e \mid \texttt{let}\ f\ p = e \\
 & & \mid \texttt{let node}\ f\ p = e \\
 & & \mid \texttt{let hybrid}\ f\ p = e \mid d\ d \\
p & ::= & () \mid x \mid x, \ldots, x \\
e & ::= & c \mid x \mid \texttt{last}\ x \mid f\,(e, \ldots, e) \\
 & & \mid \texttt{pre}\ e \mid e\ \texttt{fby}\ e \mid (e, \ldots, e) \mid () \\
 & & \mid \texttt{let}\ E\ \texttt{in}\ e \mid \texttt{let rec}\ E\ \texttt{in}\ e \\
 & & \mid \texttt{reset}\ e\ \texttt{every}\ e \\
 & & \mid \texttt{present}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
 & & \mid \texttt{up}(e) \\
 & & \\
E & ::= & p = e \mid \texttt{der}\ x = e\ \texttt{init}\ e \mid E\ \texttt{and}\ E
\end{array}
$$

A Hybrid extension

A small language of functions with data-flow equations. Blue operations are those of a synchronous language.

Three new constructs:

- `der` x = e_1 `init` e_2 defines the time derivative of x to be the value of e_1 and initial value that of e_2;
- `up`(e) defines a zero-crossing event at time when e crosses 0;
- `present` e `then` e_1 `else` e_2 tests the presence of event e.

Previous work

A stream semantics with infinitesimal steps [Benveniste et al., 2014]

- $[\![e]\!]^{\partial}_{T}(t) = v$ defines the value of e at instant $t \in T$ (T is the clock).
- Discrete-time signals (streams): $T \subseteq \mathbb{N}$.
- Continuous-time signals: $T \subseteq \{n\,\partial \mid n \in {}^{\star}\mathbb{N}\}$ and $\partial \approx 0$

It was very useful as a first step. Yet:

- It is not operational;
- it is difficult to define precisely what is a zero-crossing;
- a subtle interaction between $\perp$ and non standard reals;
- determinacy is hidden under the carpet. [1]

Can we do without in a way that is close to the way the compiler works?

---

[1] E.g., the IVP $y' = f(y)$ `init` 0 with $f(y) = y^{1/3}$ has two solutions: $y(t) = 0$ and $y(t) = c * t^{3/2}$ with $c = sqrt(2/3)$.

An more operational view

Define a semantics that is operational with the solver for ODEs and zero-crossing parameters of the semantics.

A Coiterative Semantics

- A reformulation of the old "coiterative semantics" [Caspi and Pouzet, 1998] [2].
- An executable semantics and reference interpreter [3].

Language expressiveness

- first-order subset of Zelus;
- mix of streams and hierarchical automata a la Lucid Synchrone;
- no continuous-time; neither ODEs nor zero-crossing.

Objective: extend the semantics to treat continuous-time operations.

---

[2] Talks at SYNCHRON'19 and course notes at MPRI, nov. 2019 and 2020.

[3] `https://github.com/marcpouzet/zrun`

A coiterative interpretation of streams [Jacobs and Rutten, 1997]

Streams as sequential processes [Paulin-Mohring, 1995]

A *concrete stream* producing values in the set T is 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 streams $CoF(f, s)$ and $CoF(f', s')$ are equivalent iff:

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

Synchronous Stream Processes [Caspi and Pouzet, 1998]

A stream function should be a value from:

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

that is:

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

Consider the particular class of length preserving functions.

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

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

It is the classical definition of a Mealy machine.

Synchronous Application [Caspi and Pouzet, 1998]

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

$$\begin{aligned} run(CoP(f_t, f_s))(CoF(x, x_s)) = CoF\ \lambda(m, s).\ & let\ v, x_s = x\ x_s\ in \\ & let\ v, m = f_t\ m\ v\ in \\ & v, (m, x_s) \\ & (f_s, x_s) \end{aligned}$$

with

$$\begin{aligned} run(.)(.) : sNode(T, T', S') \ & \rightarrow coStream(T, S) \\ & \rightarrow coStream(T', S' \times S) \end{aligned}$$

Feedback (fixpoint)

Consider:

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

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 $CoP(f_t, s_0)$ such that $f\ y = run(CoP(f_t, s_0))(y)$.

If $y_n = nth(y)(n)$, we should have:

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

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

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

where v is defined recursively.

$CoF(fix\,(f_t), s)$ is a stream that is a solution of the 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.

Yet, $fix\,(.)$ is not a total function; it may diverge for some functions f_t.

To study its existence, complete the set of values with $\perp$ that denotes divergence [Reynolds, 1998].

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.

Fixpoint and 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 can be unbounded: it may contains a chain (comparable elements) of unbounded length.

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

We exploit this intuition for the computation of the fix-point

Bounded Fixpoint

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

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

with:

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

or the equivalent form $fix\,(f)\,(s)(n)(\bot)$ with:

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

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

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

with:

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

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 give only a credit of $\|T\| + 1$ iterations for a fixpoint on a value of type T. This simple combinatorial argument was used in [Edward and Lee, 2003] to give a denotational semantics for a synchronous block diagram language.

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.

We suppose the existence of a environment γ for global definitions. It is kept implicit in the following definitions.

$\gamma(x)$ returns either a value $Val(v)$ or a node $CoP(p, s)$.

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

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

$$
\begin{array}{lcl}
[\![f\ (e_1, ..., e_n)]\!]^{Init}_{\rho} & = & [\![e_1]\!]^{Init}_{\rho}, ..., [\![e_n]\!]^{Init}_{\rho} \\
[\![f\ (e_1, ..., e_n)]\!]^{State}_{\rho} & = & \lambda s.\, let\ (v_i, s_i = [\![e_i]\!]^{State}_{\rho}(s_i))_{i \in [1..n]}\ in \\
 & & \quad fo(v_1, ..., v_n), s \\
 & & \text{if } \gamma(f) = Val(fo)
\end{array}
$$

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

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

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 structures (conditional/automata/reset)

This semantics extends to control structures. [4]

[4] Talk at SYNCHRON'19; MPRI course notes, nov. 19 and 20.

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:

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

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

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

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

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

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

Removing Recursion

Yet, the semantics we have given computes a step function which must be evaluated lazilly. Is this better than 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 $f : \mathit{coStream}(T, S) \to \mathit{coStream}(T, S')$ whose semantics is (f_t, s_t) with $f_t : S' \to T \to T' \times S'$ and $s_t : S'$.

The semantics of an equation $y = f(y)$ is: [5]

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

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

[5] 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:

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

Compilation

Define an intermediate language, possibly purely function, to represent the state and transition function.

$[\![.]\!]_{\cdot}^{\cdot}$ becomes a function which, given an expression e returns an expression c whose semantics is that of e.

The initial version of Lucid Synchrone was done this way.

Adding ODEs, zero-crossing and hybrid nodes.

A hybrid node

The language is a small synchronous language with two novel constructs `der . = .` and `up(.)` that must only appear in the body of a hybrid node.

Interpret a hybrid node as if it were a regular node:

`der` x = e `init` e_2

defines a state variable with two fields:

- One that contains the current value of x;
- One that contains the current derivative of x

`up`(e)

defines a state variable with two fields:

- A boolean value, true when e crosses zero, false otherwise;
- The current value of e.

Its semantics is $CoP(f_t, s_0)$ for some function f_t and initial state s_0.

$$[\![\texttt{der}\ x\ \texttt{=}\ e_1\ \texttt{init}\ e_2]\!]_\rho^{Init} = (0, 0, \texttt{true}, [\![e_1]\!]_\rho^{Init}, [\![e_2]\!]_\rho^{Init})$$

$$\begin{array}{l}
[\![\texttt{der}\ x\ \texttt{=}\ e_1\ \texttt{init}\ e_2]\!]_\rho^{State} = \\
\quad \lambda(cx, dx, i_0, s_1, s_2). \\
\quad\quad \mathit{let}\ v, s_1 = [\![e_1]\!]_\rho^{State}(s_1)\ \mathit{in} \\
\quad\quad \mathit{let}\ cx, i_0, s_2 = \mathit{if}\ i_0\ \mathit{then}\ \mathit{let}\ v_2, s_2 = [\![e_2]\!]_\rho^{State}(s_2)\ \mathit{in} \quad \mathit{in} \\
\quad\quad\quad\quad\quad\quad\quad\quad (v_2, \texttt{false}, s_2) \\
\quad\quad\quad\quad\quad\quad \mathit{else}\ (cx, i_0, s_2) \\
\quad\quad ([cx/x], (cx, v, i_0, s_1, s_2))
\end{array}$$

$$[\![\texttt{up}(e)]\!]_\rho^{Init} = (\texttt{false}, \mathit{nil}, [\![e]\!]_\rho^{Init})$$

$$\begin{array}{l}
[\![\texttt{up}(e)]\!]_\rho^{State}(zi, zo, s) = \mathit{let}\ v, s = [\![e]\!]_\rho^{State}(s)\ \mathit{in} \\
\quad\quad\quad\quad\quad\quad\quad\quad zi, (zi, v, s)
\end{array}$$

Provide accessing functions:

- $cset(s, y)$ stores the position of the continuous state y into s;
- $cget(s)$ output the position of the continuous state y from s;
- $dget(s)$ outputs the derivative of the continuous state y from s;
- $zset(s, z)$ sets the zero-crossing values;
- $zget(s)$ outputs the zero-crossing values to be observed

Build the following three functions:

- Derivative:

$$f : S \to Y \to Y' = \lambda s, y.let\ v, s = f_t\ (cset(s, y))\ nil\ in\ dget(s)$$

- Zero-crossing function:

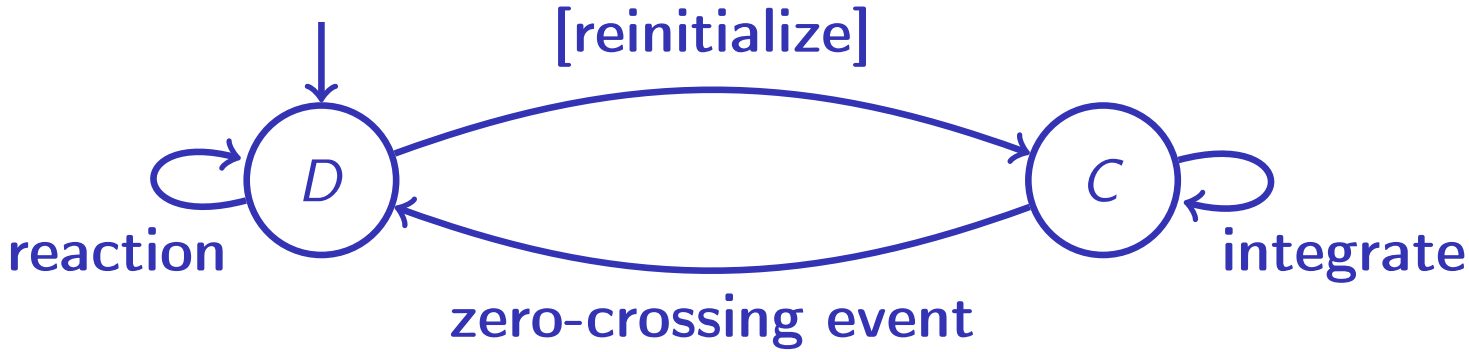
$$g : S \to Y \to Zo = \lambda s, y.let\ v, s = f_t\ (cset(s, y))\ nil\ in\ zget(s)$$

- Step function:

$$step : S \to Y \to Zi \to T \to T' \times S \times Y =$$
$$\lambda s, y, zi, x.let\ v, s = f_t\ (zset(cset(s, y), zi))\ x\ in\ v, (s, cget(s))$$

The Simulation Loop [Bourke et al., 2015]

Alternate discrete steps and integration steps



$$s', y' = step(s)(y) \qquad upz = g(s)(y) \qquad \dot{y} = f(s)(y)$$

The purpose of the compiler is to generate:

- *step* gathers all discrete state changes.
- g define the zero-crossing signals.
- f define the time derivative of continuous-state variables.

Ensure that f and g are pure (side-effect free)? Differentiable?

The Simulation Loop [Bourke et al., 2015]

The execution can be defined as a function which is parameterised by a function *csolve* and *zsolve*.

$$csolve : (Y \to Y') \to Y \to (Time \times (Time \to Y))$$

$$zsolve : (Y \to Zo) \to (Time \to Y) \to Time \times Time \to (Time \times Zi)$$

Given $f\ s$ and y, $csolve(f\ s)(y) = h, dky$. dky is a *dense* solution, that is:

$$y(t) \approx dky(t) \text{ for } t \in [0, h]$$

$zsolve(g\ s)dky(h_0, h_1) = h, zi$ locates the zero-crossing of $g\ s$.

It either returns $h = h_1$ and $zi = \texttt{false}^l$ if not zero-crossing occurs;

or the earliest instant $h \in [h_0, h_1]$ and the vector zi with for all $k \in [1..l]$, $zi[k] = \texttt{true}$ if $(g\ s)(y)[k]$ crosses zero.

A cyclic execution of:

1. In the integration mode, the fix-point for all variables defined by an ODE is computed globally by the ODE solver, returning the actual horizon and a dense solution;
2. Then zero-crossing are detected, returning the actual value for $y : Y$ and $zi : Zi$;
3. Then a discrete step is performed.

Alternatively

Instead of generating a single step function with a state that contains positions, derivatives and zero-crossing information, define directly the elements of a hybrid expression:

$$
\begin{aligned}
&hNode(T, T', S, Y, Zi, Zo) = \\
&\quad CoH\ (S \to Y \to Y', \\
&\qquad S \to Y \to Zo, \\
&\qquad S \to Y \to T', \\
&\qquad S \to Y \to Zi \to T \to T' \times S \times Y, \\
&\qquad S, \\
&\qquad Y)
\end{aligned}
$$

where the semantics value of an expression becomes of the form: $CoH(f, g, out, step, s, y)$

- f defines the derivative;
- g defines the zero-crossings;
- *out* defines the output from the current discrete state and continuous state;
- *step* defines the step function to be evaluated at a zero-crossing instant;
- s is the initial discrete state;
- y is the initial continuous state.

Probabilistic extension

Can we follow a similar approach?

This is ongoing work

Coiteration interprets the generation of a stream as the infinite iteration of a transition function from an initial state.

It replaces a recursion by an iteration and corresponds to what a synchronous compiler output from a synchronous program.

Extend it to give an ideal semantics for the kernel language extended with ODEs.

Revisit the dependence/causality analysis and state the theorem of [Benveniste et al., 2014] without the use of non standard analysis.

Make it even more abstract, replacing the concrete interpretation of arithmetic by set operations to perform set-based simulation.

Is there a way to treat the probabilistic extension the same way, as a two level language?

References I

Benveniste, A., Bourke, T., Caillaud, B., Pagano, B., and Pouzet, M. (2014).

A Type-based Analysis of Causality Loops in Hybrid Systems Modelers.
In *International Conference on Hybrid Systems: Computation and Control (HSCC)*, Berlin, Germany. ACM.

Bourke, T., Colaço, J.-L., Pagano, B., Pasteur, C., and Pouzet, M. (2015).

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

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`.

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.

Jacobs, B. and Rutten, J. (1997).

A tutorial on (co)algebras and (co)induction.
EATCS Bulletin, 62:222–259.

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/`.

Reynolds, J. C. (1998).

Theories of Programming Languages.
Cambridge University Press.