An Executable Constructive Semantics for Zélus

Marc Pouzet

Ecole normale supérieure
`Marc.Pouzet@ens.fr`

SYNCHRON
November 25, 2021

Motivation

Objective

- A reference semantics for Zélus,
- that is constructive/executable, i.e., the basis of an interpreter;
- that applies *directly* to the source before any compilation step;
- both discrete and continuous-time systems.

Used for compiler testing, debugging of partial models; to prove compiler steps.

Approach

- Zélus is a two level language: a kernel synchronous language on top of which continuous-time operations are added.
- Define the semantics as a functor parameterized by the ODE and zero-crossing solvers.

The language kernel

A first-order subset of Zélus.

$$
\begin{array}{rcl}
d & ::= & \texttt{let}\ f = e \mid \texttt{let}\ f\ \ p\ \texttt{=}\ e \\
 & & \qquad\qquad\quad \mid \texttt{let}\ \texttt{node}\ f\ \ p\ \texttt{=}\ e \\
 & & \qquad\qquad\quad \mid \texttt{let}\ \texttt{hybrid}\ f\ \ p\ \texttt{=}\ e \mid d\ d \\
p & ::= & () \mid x \mid x, \ldots, x \\
e & ::= & c \mid x \mid f\,(e, \ldots, e) \\
 & & \mid \texttt{pre}\ e \mid e\ \texttt{fby}\ e \mid (e, \ldots, e) \mid () \\
 & & \mid \texttt{let}\ E\ \texttt{in}\ e \mid \texttt{let}\ \texttt{rec}\ E\ \texttt{in}\ e \\
 & & \mid \texttt{up}\, e \mid \texttt{last}\, x \\
 & & \\
E & ::= & p\ \texttt{=}\ e \mid E\ \texttt{and}\ E \\
 & & \qquad\quad \mid \texttt{der}\ xe
\end{array}
$$

Hybrid extension

A first-order functional synchronous language.

Three new constructs:

- `der` x `=` e defines the time derivative of x to be the value of e;
- `up` e defines a zero-crossing event at time when e crosses 0;
- `let hybrid` f p `=` e defines a hybrid node, i.e., a system whose base time is $\mathbb{R}$ (instead of $\mathbb{N}$).

Constructive/executable Semantics

Define a semantics that is executable. For hybrid systems, make the semantics parameterized by the solver for ODEs and zero-crossing detection.

A Coiterative Semantics

- A reformulation of the old "coiterative semantics" [Caspi and Pouzet, 1998].
- An executable semantics and reference interpreter [1].

Language expressiveness

- first-order subset of Zélus;
- 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.

---

[1] `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

A value $f = CoP(f^t, s)$ defines a stream function thanks to the function $run(.)(.)$:

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

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 $CoP(f^t, s)$ 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:

$$fix\left(f^{t}\right) = \lambda s.let\ rec\ v, s' = f^{t}\ s\ v\ in\ v, s'$$

where v is defined recursively.

$CoF(fix\left(f^{t}\right), s)$ 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, $fix\left(.\right)$ is not a total function, e.g., it may diverge for some functions.

Idea: Complete a set T with $\perp$ to expliticly represent divergence and compute a bounded fix-point.

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 $f : T \rightarrow T'$ 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 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 ($fix\,(f) = \lim\limits_{n\to\infty}(f^n(\perp))$.

Yet, this does not lead to a computational definition because the height of D can be unbounded.

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

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

The idea of bounded iteration was exploited in [Edward and Lee, 2003].

Bounded Fixpoint

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

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

with:

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

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

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

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

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

with:

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

The semantics of an expression e is:

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

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;
- $[\![e]\!]_\rho^{Step}$ 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 $CoP(p, s)$.

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

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

Fby

Two cases: either the first argument is a constant or not.

$$\begin{array}{lcl}
[\![v \ \texttt{fby}\ e]\!]_\rho^{Init} & = & (v, [\![e]\!]_\rho^{Init}) \\
[\![v \ \texttt{fby}\ e]\!]_\rho^{Step}(m, s) & = & m, let\ v, s = [\![e]\!]_\rho^{Step}(s)\ in\ (v, s) \\
 & & \\
[\![e_1 \ \texttt{fby}\ e_2]\!]_\rho^{Init} & = & (None, [\![e_1]\!]_\rho^{Init}, [\![e_2]\!]_\rho^{Init}) \\
 & & \\
[\![e_1 \ \texttt{fby}\ e_2]\!]_\rho^{Step}(None, s_1, s_2) & = & let\ v_1, s_1 = [\![e_1]\!]_\rho^{Step}(s)\ in \\
 & & v_1, let\ v_2, s_2 = [\![e_2]\!]_\rho^{Step}(s_2)\ in \\
 & & \quad (Some(v_2), s_1, s_2) \\
[\![e_1 \ \texttt{fby}\ e_2]\!]_\rho^{Step}(Some(v), s_1, s_2) & = & v, let\ v_1, s_1 = [\![e_1]\!]_\rho^{Step}(s)\ in \\
 & & \quad let\ v_2, s_2 = [\![e_2]\!]_\rho^{Step}(s_2)\ in \\
 & & \quad (Some(v_2), s_1, s_2)
\end{array}$$

$$
\begin{aligned}
\llbracket f\ (e_1, ..., e_n) \rrbracket^{Init}_{\rho} &= \llbracket e_1 \rrbracket^{Init}_{\rho}, ..., \llbracket e_n \rrbracket^{Init}_{\rho} \\
\llbracket f\ (e_1, ..., e_n) \rrbracket^{Step}_{\rho} &= \lambda s.\, let\, (v_i, s_i = \llbracket e_i \rrbracket^{Step}_{\rho}(s_i))_{i \in [1..n]}\ in \\
&\qquad fo(v_1, ..., v_n), s \\
&\text{if } \gamma(f) = Val(fo)
\end{aligned}
$$

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

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

$$
\text{where } s = \llbracket e \rrbracket^{Init}_{\rho + [\bot/x_1, ..., \bot/x_n]} \text{ and } p = \lambda s, (v_1, ..., v_n).\llbracket e \rrbracket^{Step}_{\rho + [v_1/x_1, ..., v_n/x_n]}(s)
$$

Equations

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

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

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

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

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

$\|E\|$ is the number of variables defined by E.

Let $Def(E) = \{x_1, ..., x_n\}$, the set of defined variables in E.

$$
\begin{aligned}
[\![\texttt{let}\ E\ \texttt{in}\ e']\!]_\rho^{Init} &= [\![E]\!]_\rho^{Init}, [\![e']\!]_{\rho+[\bot/x_1,...,\bot/x_n]}^{Init} \\
[\![\texttt{let}\ E\ \texttt{in}\ e']\!]_\rho^{Step} &= \lambda(s,s').let\ \rho', s = [\![E]\!]_\rho^{Step}(s)\ in \\
&\qquad\qquad let\ v', s' = [\![e']\!]_{\rho+\rho'}^{Step}(s')\ in \\
&\qquad\qquad v', (s, s')
\end{aligned}
$$

$$
\begin{aligned}
[\![\texttt{let}\ \texttt{rec}\ E\ \texttt{in}\ e']\!]_\rho^{Init} &= [\![e]\!]_\rho^{Init}, [\![e']\!]_{\rho+[\bot/x_1,...,\bot/x_n]}^{Init} \\
[\![\texttt{let}\ \texttt{rec}\ E\ \texttt{in}\ e']\!]_\rho^{Step} &= \lambda(s,s').let\ \rho', s = [\![\texttt{rec}\ E]\!]_\rho^{Step}(s)\ in \\
&\qquad\qquad let\ v', s' = [\![e']\!]_{\rho+\rho'}^{Step}(s')\ in \\
&\qquad\qquad v', (s, s')
\end{aligned}
$$

Control Structures

Equations are extended with local definitions:

$$\begin{array}{lcl} E & ::= & \ldots \mid \texttt{local}\; v \;\texttt{in}\; E \mid \texttt{reset}\; E \;\texttt{every}\; e \mid \texttt{if}\; e \;\texttt{then}\; E \;\texttt{else}\; E \\ v & ::= & x \mid x \;\texttt{init}\; e \mid x \;\texttt{default}\; e \end{array}$$

Expressions are extended with a construct to access the last value of a stream:

$$e \quad ::= \quad \ldots \mid \texttt{last}\; x$$

Environment

The construct `local` x `in` E declares x to be local in E.

The construct `local` x `init` e `in` E declares x to be local and the *last computed value of* x to be initialized with the value of e.

The construct `local` x `default` e `in` E declares x to be local and the *default value of* x to be the value of e, at instants where no definition of x is given.

Conditionals over Equations

If e is an expression whose type is a sum type $t = C_1 \mid \dots \mid C_n$,

- `match` e `with` $C_{i_1} \rightarrow E_1 \;\mid\; \dots \;\mid\; C_{i_n} \rightarrow E_n$ activates equation E_j such that i_j is the first index such that $e = C_{i_j}$, with $1 \leq i_1, \dots, i_n \leq n$.
- `if` e `then` E_1 `else` E_2 a short-cut for `match` e `with` `true` $\rightarrow E_1 \;\mid\;$ `false` $\rightarrow E_2$

$$E \quad ::= \quad \dots \mid \texttt{match}\ e\ \texttt{with}\ C \rightarrow E \;\mid \dots \;\mid C \rightarrow E$$

Reset

Two ways:

- Recompute the initial state of E when the reset condition e is true or;
- duplicate the initial state of E and use this state every time e is true. [2]

We adopt the later solution.

$$\begin{array}{lcl}
[\![\texttt{reset}\ E\ \texttt{every}\ e]\!]_\rho^{Init} & = & [\![E]\!]_\rho^{Init}, [\![E]\!]_\rho^{Init}, [\![e]\!]_\rho^{Init} \\
[\![\texttt{reset}\ E\ \texttt{every}\ e]\!]_\rho^{Step}(s_0, s_1, s_2) & = & \mathit{let}\ v, s_2 = [\![e]\!]_\rho^{Step} s_2\ \mathit{in} \\
 & & \mathit{let}\ s_1 = \mathit{if}\ s_2\ \mathit{then}\ s_0\ \mathit{else}\ s_1\ \mathit{in} \\
 & & \mathit{let}\ \rho, s_1 = [\![E]\!]_\rho^{Step} s_1\ \mathit{in} \\
 & & \rho, (s_0, s_1, s_2)
\end{array}$$

---

[2] This idea is due to Louis Mandel.

Hierarchical Automata

A automaton which describe a system with several modes and transitions between them.

Such an automaton is characterized by:

- A finite set of states.
- In every state, a set of equations with variables that are possibly local to the state.
- A set (possibly empty) of “weak transitions” (keyword `until`) which define the active state for the next reaction.
- A set (possibly empty) of “strong transitions” (keyword `unless`) which define the active set of equations for the current reaction.
- Transitions can be by “reset” (condition `then`) or by “history” (condition `continue`).
- By default, the initial state is the first in the list. If given, ... `init` *se* defines the initial state of the automaton to be the value of *se*.

Rmq: Contrary to Scade 6 and Lucid Synchrone that implement [?], in Zelus, weak and strong transitions cannot be mixed inside an automaton.

The syntax is extended in the following way.

$$\begin{array}{lcl}
E & ::= & \ldots \mid \texttt{automaton}\ (S(p) \rightarrow u\ wt)^{+}\ \texttt{init}\ se^{\epsilon} \\
 & & \quad \mid \texttt{automaton}\ (S(p) \rightarrow u\ st)^{+}\ \texttt{init}\ se^{\epsilon} \\
u & ::= & \texttt{local}\ v\ \texttt{in}\ u \mid \texttt{do}\ E \\
st & ::= & \texttt{unless}\ t^{*} \\
wt & ::= & \texttt{until}\ t^{*} \\
t & ::= & e\ \texttt{then}\ S(e, \ldots, e) \ \mid e\ \texttt{continue}\ S(e, \ldots, e) \\
se & ::= & S(e, \ldots, e) \mid \texttt{if}\ e\ \texttt{then}\ se\ \texttt{else}\ se
\end{array}$$

se stands for an expression which returns a state. It is used at the first instant of activation or reset of the automaton.

Examples in Zelus

```
type t = Incr | Decr | Idle

let f(c) =
   local o init 0
   do
    match c with
    | Idle -> (* o keeps its previous value, i.e., o = last o *)
              do done
    | Incr -> do o = last o + 1 done
    | Decr -> do o = last o - 1 done
   in o
```

Examples in Zelus

```
let node controller(auto, error, input) = output where rec
  automaton
  | Manual -> do output = input unless auto then Auto
  | Auto -> do output = run pid(p, i, d, error)
            unless (not auto) then Manual

let node await(a) = go where rec
  automaton
  | Await -> do go = false unless a then Run
  | Go -> do go = true done

let node abro(a, b, r) = go where rec
  reset automaton
        | Await -> do go = false
                   unless (run await(a) && run await(b))
                   then Go
        | Go -> do go = true done
  every r
```

Semantics

Environment

The environement is complemented to possibly associate a default or initial value to a variable.

$$\rho ::= \rho + [v/x] \mid \rho + [v/\mathit{default}\ x] \mid [v/\mathit{last}\ x] \mid []$$

If ρ and ρ' are two environments, we write $\rho \,\mathtt{by}\, \rho'$ the completion of ρ with default or initial values from ρ'.

This operation is used to define the value of a variable in

$$\begin{array}{lcl}
\rho \,\mathtt{by}\, [] & = & \rho \\
\rho \,\mathtt{by}\, (\rho' + [v/\mathit{default}\ x]) & = & (\rho + [v/x]) \,\mathtt{by}\, \rho' \\
\rho \,\mathtt{by}\, (\rho' + [v/\mathit{last}\ x]) & = & (\rho + [v/x]) \,\mathtt{by}\, \rho' \\
\rho \,\mathtt{by}\, (\rho' + [v/x]) & = & \rho \,\mathtt{by}\, \rho'
\end{array}$$

If p is a pattern and v is a value, `match` v `with` p builds the environment by matching v by p such that:

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

Notation: If $\rho = \rho' + [v/x]$, $\rho \backslash x = \rho'$.

Let $n = \|E\| + 1$.

$$\begin{array}{lcl} [\![\texttt{local}\, x\, \texttt{in}\, E]\!]^{Init}_{\rho} & = & [\![E]\!]^{Init}_{\rho} \\ [\![\texttt{local}\, x\, \texttt{in}\, E]\!]^{Step}_{\rho}(s) & = & let\, \rho', s = fix\,(n)\,(\lambda s, \rho'.[\![E]\!]^{Step}_{\rho+\rho'}(s))(s)\; in \\ & & \rho' \backslash x, s \end{array}$$

$$\begin{array}{lcl} [\![\texttt{local}\, x\, \texttt{default}\, v\, \texttt{in}\, E]\!]^{Init}_{\rho} & = & [\![E]\!]^{Init}_{\rho} \\ [\![\texttt{local}\, x\, \texttt{init}\, v\, \texttt{in}\, E]\!]^{Init}_{\rho} & = & (v, [\![E]\!]^{Init}_{\rho}) \end{array}$$

$$\begin{array}{l} [\![\texttt{local}\, x\, \texttt{default}\, v\, \texttt{in}\, E]\!]^{Step}_{\rho}(s) = \\ \qquad let\, \rho', s = fix\,(n)\,(\lambda \rho', s.[\![E]\!]^{Step}_{\rho+\rho'+[v/default\; x]}(s))(s)\; in \\ \qquad \rho' \backslash x, s \end{array}$$

$$\begin{array}{l} [\![\texttt{local}\, x\, \texttt{init}\, v\, \texttt{in}\, E]\!]^{Step}_{\rho}(w, s) = \\ \qquad let\, \rho', s = fix\,(n)\,(\lambda \rho', s.[\![E]\!]^{Step}_{\rho+\rho'+[w/last\; x]}(s))(s)\; in \\ \qquad \rho' \backslash x, (\rho'(x), s) \end{array}$$

Semantics for conditionals

The semantics for a conditional must consider the case where a branch defines a value for a variable `x` in one branch but not the other branch. We take the following convention:

- If a variable x is declared with a default value v, then a missing equation for x in a branch means that $x = v$ in that branch.
- Otherwise, $x = \texttt{last}\, x$, that is, x keeps its previous value.
- If x is declared with an initial value for $\texttt{last}\, x$, this means that x has a definition in every branch. Otherwise, there is a potential initialisation issue which has to be checked by other means.

Semantics for Conditionals

The Initial State

$$[\![\texttt{match}\ e\ \texttt{with}\ (C_i \rightarrow E_i)_{i\in[1..n]}]\!]_\rho^{Init} = ([\![e]\!]_\rho^{Init}, [\![E_1]\!]_\rho^{Init}, ..., [\![E_n]\!]_\rho^{Init})$$

The Transition Function:

$$\begin{array}{l} [\![\texttt{match}\ e\ \texttt{with}\ (C_i \rightarrow E_i)_{i\in[1..n]}]\!]_\rho^{Step}(s, s_1, ..., s_n) = \\ \qquad \mathit{let}\ v, s = [\![e]\!]_\rho^{Step}(s)\ \mathit{in} \\ \qquad \mathit{match}\ v\ \mathit{with} \\ \qquad \left(\begin{array}{l} C_i \rightarrow \mathit{let}\ \rho_i, s_i = [\![E_i]\!]_\rho^{Step}(s_i)\ \mathit{in} \\ \qquad\qquad \rho_i\ \text{by}\ \rho[N \backslash N_i], (s, s_1, ..., s_n) \end{array} \right)_{i\in[1..n]} \\ \text{where}\ N = \cup_{i\in[1..n]}(N_i)\text{and}\ N_i = \mathit{Def}(E_i) \end{array}$$

The Last Computed Value:

$$\begin{array}{lcl} [\![\texttt{last}\ x]\!]_\rho^{Init} = () & & \\ [\![\texttt{last}\ x]\!]_\rho^{Step} & = & \lambda s.\rho(\mathit{last}\ x), s \end{array}$$

Hierarchical Automata

We consider here only the case where no initialisation state *se* is given.

Initial state of the transition function

$$
\begin{array}{l}
[\![\texttt{automaton}\ (S_i(p_i) \rightarrow u_i\ wt_i)_{i\in[1..n]}]\!]_\rho^{Init} \quad = \\
\quad let\ (s_i = [\![u_i]\!]_\rho^{Init})_{i\in[1..n]}\ in \\
\quad let\ (s'_i = [\![wt_i]\!]_\rho^{Init})_{i\in[1..n]}\ in \\
\quad (S_1(), false, (s_1, \ldots, s_n), (s'_1, \ldots, s'_n)) \\
[\![\texttt{automaton}\ (S_i(p_i) \rightarrow u_i\ st_i)_{i\in[1..n]}]\!]_\rho^{Init} \quad = \\
\quad let\ (s_i = [\![u_i]\!]_\rho^{Init})_{i\in[1..n]}\ in \\
\quad let\ (s'_i = [\![st_i]\!]_\rho^{Init})_{i\in[1..n]}\ in \\
\quad (S_1(), false, (s_1, \ldots, s_n), (s'_1, \ldots, s'_n))
\end{array}
$$

$$
\begin{array}{l}
[\![\texttt{automaton}\ (S_i(p_i) \rightarrow u_i\ wt_i)_{i\in[1..n]}]\!]_\rho^{Step}(v, r, s, s') = \\
\quad let\ (\rho, v, r), (s, s') = [\![(S_i(p_i) \rightarrow u_i\ wt_i)_{i\in[1..n]}]\!]_\rho^{v,r}(s, s')\ in \\
\quad \rho, (v, r, s, s') \\
[\![\texttt{automaton}\ (S_i(p_i) \rightarrow u_i\ st_i)_{i\in[1..n]}]\!]_\rho^{Step}(v, r, s, s') = \\
\quad let\ (\rho, v, r), (s, s') = [\![(S_i(p_i) \rightarrow u_i\ st_i)_{i\in[1..n]}]\!]_\rho^{v,r}(s, s')\ in \\
\quad \rho, (v, r, s, s')
\end{array}
$$

$$
\begin{array}{l}
\llbracket (S_i(p_i) \to u_i\ wt_i)_{i\in[1..n]} \rrbracket_\rho^{v,r}((s_1,...,s_n),(s'_1,...,s'_n)) = \\
\quad \mathit{match}\ v\ \mathit{with} \\
\qquad \left(\begin{array}{ll} S_i(p_i) \to & \mathit{let}\ \rho, s_i = \llbracket u_i \rrbracket_\rho^r(s_i)\ \mathit{in} \\ & \mathit{let}\ (v,r), s'_i = \llbracket wt_i \rrbracket_\rho^{v,r}(s'_i)\ \mathit{in} \\ & \rho, (v,r,(s_1,...,s_n),(s'_1,...,s'_n)) \end{array} \right)_{i\in[1..n]}
\end{array}
$$

$$
\begin{array}{l}
\llbracket (S_i(p_i) \to u_i\ st_i)_{i\in[1..n]} \rrbracket_\rho^{v,r}((s_1,...,s_n),(s'_1,...,s'_n)) = \\
\qquad \mathit{let}\ (v,r,(s'_1,...,s'_n) = \\
\qquad\quad \mathit{match}\ v\ \mathit{with} \\
\qquad\qquad \left(\begin{array}{ll} S_i(p_i) \to & \mathit{let}\ (v,r), s'_i = \llbracket st_i \rrbracket_\rho^{v,r}(s'_i)\ \mathit{in} \\ & (v,r,(s'_1,...,s'_n)) \end{array} \right)_{i\in[1..n]} \\
\qquad \mathit{in} \mathit{match}\ v\ \mathit{with} \\
\qquad\qquad \left(\begin{array}{ll} S_i(p_i) \to & \mathit{let}\ \rho, s_i = \llbracket u_i \rrbracket_\rho^r(s_i)\ \mathit{in} \\ & \rho, (v,r,(s_1,...,s_n),(s'_1,...,s'_n)) \end{array} \right)_{i\in[1..n]}
\end{array}
$$

$$
\begin{array}{lcl}
[\![\texttt{until}\ t^*]\!]_\rho^{Init} & = & [\![t^*]\!]_\rho^{Init} \\
[\![\texttt{unless}\ t^*]\!]_\rho^{Init} & = & [\![t^*]\!]_\rho^{Init} \\
[\![\texttt{until}\ t^*]\!]_\rho^{v,r}(s) & = & [\![t^*]\!]_\rho^{v,r}(s) \\
[\![\texttt{unless}\ t^*]\!]_\rho^{v,r}(s) & = & [\![t^*]\!]_\rho^{v,r}(s) \\
[\![\epsilon]\!]_\rho^{Init} & = & \texttt{()} \\
[\![e\ \texttt{then}\ se\ t^*]\!]_\rho^{Init} & = & ([\![e]\!]_\rho^{Init}, [\![se]\!]_\rho^{Init}) \\
[\![e\ \texttt{continue}\ se\ t^*]\!]_\rho^{Init} & = & ([\![e]\!]_\rho^{Init}, [\![se]\!]_\rho^{Init}) \\
[\![\epsilon]\!]_\rho^{v,r}(s) & = & (v, r), s
\end{array}
$$

$$
\begin{array}{l}
[\![e\ \texttt{then}\ se\ t^*]\!]_\rho^{v,r}((s_1,s_2),s_3) = \\
\quad \mathit{let}\ s_1 = \mathit{if}\ r\ \mathit{then}\ [\![e]\!]_\rho^{Init}\ \mathit{else}\ s_1\ \mathit{in} \\
\quad \mathit{let}\ s_2 = \mathit{if}\ r\ \mathit{then}\ [\![se]\!]_\rho^{Init}\ \mathit{else}\ s_2\ \mathit{in} \\
\quad \mathit{let}\ s_3 = \mathit{if}\ r\ \mathit{then}\ [\![t^*]\!]_\rho^{Init}\ \mathit{else}\ s_3\ \mathit{in} \\
\quad \mathit{let}\ c, s_1 = [\![e]\!]_\rho^{Step}(s_1)\ \mathit{in} \\
\quad \mathit{if}\ c\ \mathit{then}\ \mathit{let}\ v, s_2 = [\![se]\!]_\rho^{Step}(s_2)\ \mathit{in}\ (v, \mathit{true}), ((s_1,s_2),s_3) \\
\quad \mathit{else}\ \mathit{let}\ (v,r), s_2 = [\![t^*]\!]_\rho^{v,r}(s)\ \mathit{in}\ (v,r),(s_1,s_2) \\
[\![e\ \texttt{continue}\ se\ t^*]\!]_\rho^{v,r}((s_1,s_2),s_3) = \\
\quad \mathit{let}\ s_1 = \mathit{if}\ r\ \mathit{then}\ [\![e]\!]_\rho^{Init}\ \mathit{else}\ s_1\ \mathit{in} \\
\quad \mathit{let}\ s_2 = \mathit{if}\ r\ \mathit{then}\ [\![se]\!]_\rho^{Init}\ \mathit{else}\ s_2\ \mathit{in} \\
\quad \mathit{let}\ s_3 = \mathit{if}\ r\ \mathit{then}\ [\![t^*]\!]_\rho^{Init}\ \mathit{else}\ s_3\ \mathit{in} \\
\quad \mathit{let}\ c, s_1 = [\![e]\!]_\rho^{Step}(s_1)\ \mathit{in} \\
\quad \mathit{if}\ c\ \mathit{then}\ \mathit{let}\ v, s_2 = [\![se]\!]_\rho^{Step}(s_2)\ \mathit{in}\ (v, \mathit{false}), ((s_1,s_2),s_3) \\
\quad \mathit{else}\ \mathit{let}\ (v,r), s_2 = [\![t^*]\!]_\rho^{v,r}(s)\ \mathit{in}\ (v,r),(s_1,s_2)
\end{array}
$$

$$
\begin{array}{lcl}
[\![S(e_1,\ldots,e_n)]\!]_\rho^{Init} & = & [\![e_1]\!]_\rho^{Init},\ldots,[\![e_n]\!]_\rho^{Init} \\
[\![S(e_1,\ldots,e_n)]\!]_\rho^{Step} & = & \mathit{let}\ (v_i, s_i = [\![e_i]\!]_\rho^{Step}(s_i))_{i\in[1..n]}\ \mathit{in} \\
 & & S(v_1,\ldots,v_n),(s_1,\ldots,s_n)
\end{array}
$$

Interpretation

- The transition function associated with the automaton construct is executed in an initial state.
- This state if of the form (ps, pr, s, s'). ps is the current state of the automaton. It is initialised with the initial state of the automaton. pr is the reset status. It is initialized with the value false. s is the state to execute the code of the strong transitions; s' is the state to execute the body of the automaton; s' is the state to execute the transitions.
- For an automaton with weak transition, the body is executed, then the transitions.
- For an automaton with strong transitions, the code of transitions of the current state are executed. This determines the active state. Then, the corresponding body is executed.

Adding ODEs, zero-crossing and hybrid nodes.

A hybrid node

The language is extended with continuous-time operators.
`der` .. and `up` . that must only appear in the body of a `hybrid` node.

Idea: Interpret a hybrid node as a regular node

`der` *xe*
defines a state variable $\{cin; cout; dout\}$ with three fields:

- the current value of x (input from the solver);
- the current derivative of x (output to the solver).
- the current value of x (output to the solver).

`up` *e*
defines a state variable $\{zin; zout\}$ with two fields:

- a boolean value, true when e crosses zero (input from the solver).
- the current value of e (output to the solver).

Those states are used to communicate with the solver.

$$[\![\texttt{der}\ x\ \texttt{=}\ e]\!]_{\rho}^{Init} = [\![e]\!]_{\rho}^{Init}$$

$$\begin{array}{l}[\![\texttt{der}\ x\ \texttt{=}\ e]\!]_{\rho}^{Step}(s) = let\ v, s = [\![e]\!]_{\rho}^{Step}(s)\ in \\ \qquad\qquad\qquad\qquad\qquad [v/der\ x], s\end{array}$$

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

$$\begin{array}{l}[\![\texttt{up}\ e]\!]_{\rho}^{Step}(\{zin, zout\}, s) = let\ v, s = [\![e]\!]_{\rho}^{Step}(s)\ in \\ \qquad\qquad\qquad\qquad\qquad\qquad zin, \{zin; zout = v\}, s)\end{array}$$

Let $n = \|E\| + 1$.

$$\begin{array}{l}[\![\texttt{local}\ \texttt{der}\ x\ \texttt{init}\ v\ \texttt{in}\ E]\!]_{\rho}^{Init} = \\ \qquad (\{cin = 0.0; cout = v; dout = 0.0\}, [\![e]\!]_{\rho}^{Init}, [\![E]\!]_{\rho}^{Init})\end{array}$$

$$\begin{array}{l}[\![\texttt{local}\ \texttt{der}\ x\ \texttt{init}\ v\ \texttt{in}\ E]\!]_{\rho}^{Step}(\{cin; cout; dout\}, s) = \\ \quad let\ \rho', s = fix\,(n)\,(\lambda\rho', s.[\![E]\!]_{\rho+\rho'+[cin/default\ x][cout/last\ x]}^{Step}(s))(s)\ in \\ \quad \rho'\backslash x, (\{cin; cout = \rho'(x); dout = \rho'(der\ x)\}, s)\end{array}$$

Provide access 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

Hybrid Node

Let f be a hybrid node defined by `let hybrid` $f\ p$ `=` e. Defines its semantics $CoP(f^t, s)$ as if it were defined as a node (see slide 19). Defines the following three functions:

- Derivative:

$$f^d : S \rightarrow I \rightarrow (Y \rightarrow Y') = \lambda s, x, y. let\, v, s = f^t\ (cset(s, y))\ x\ in\, dget(s)$$

- Zero-crossing function:

$$f^z : S \rightarrow I \rightarrow (Y \rightarrow Zo) = \lambda s, x, y. let\, v, s = f^t\ (cset(s, y))\ x\ in\, zget(s)$$

- Output function:

$$f^{out} : S \rightarrow I \rightarrow (Y \rightarrow O) = \lambda s, x, y. let\, v, s = f^t\ (cset(s, y))\ x\ in\, v$$

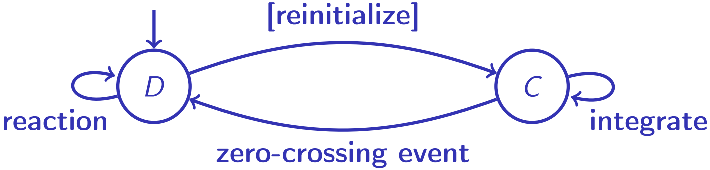
- Step function:

$$\begin{aligned} f^{step} : S \rightarrow Y \rightarrow Zi \rightarrow I \rightarrow O \times S \times Y = \\ \lambda s, y, zi, x. let\, v, s = f^t\ (zset(cset(s, y), zi))\ x\ in\, v, (s, cget(s)) \end{aligned}$$

The input x is considered to be piece-wise constant during integration, i.e., the solver calls $(f^d\, s\, x) : Y \rightarrow Y'$.

The Simulation Loop [Bourke et al., 2015]

Alternate discrete steps and integration steps



$$o, s', y' = f^{step}(s)(y)(zi)(x) \quad zo = f^{z}(s)(x)(y)$$
$$\dot{y} = f^{d}(s)(x)(y)$$

The purpose of the compiler is to generate:

- f^{step} gathers all discrete state changes.
- f^{z} define the zero-crossing signals.
- f^{d} define the time derivative of continuous-state variables.

The Simulation Loop [Bourke et al., 2015]

The execution can be defined as a function which is parameterised by two functions *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 $f : Y \to Y'$ and $y : Y$, $csolve(f)(y) = h, dky$. *dky* is a *dense* solution, that is:

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

Given $g : Y \to Zo$, $zsolve(g)(dky)(h') = h, zi$ locates the zero-crossing of g between time 0 and h'.

It either returns $h = h'$ and $zi = \texttt{false}^l$ if no zero-crossing occurs;

or the earliest instant $h \in [0, h']$ and the vector zi with for all $k \in [1..l]$, $zi[k] = \texttt{true}$ if $g(y)[k]$ crosses zero.

Given a hybrid node f and semantics $CoP(f^t, s)$. Defines s_0, f^d, f^z, f^{step} and access functions.

Let $p : \mathbb{N} \rightarrow I$ an input signal. The simulation computes o such that:

A cyclic execution of:

1. The initial state is the discrete mode with ly_0 is a vector of zeros and zi_0 is a vector of false, i.e., $ly_0[i] = 0$ for all $i \in cget(s_0)$ and $zi_0[i] = \texttt{false}$ for all $i \in zget(s_0)$.
2. In the discrete mode, compute:

$$\begin{array}{lcl} o_n, s_{n+1}, y_{n+1} & = & f^{step}\, s_n\, ly_n\, zi_n\, p_n \\ lp_{n+1} & = & p_n \end{array}$$

3. In the integration mode, compute:

$$\begin{array}{lcl} h'_n, dky_n & = & csolve(f^d\, s_n\, lp_n)(y_n)) \\ h_n, zi_{n+1} & = & zsolve(f^z\, s_n\, lp_n)(dky_n)(h'_n) \\ ly_{n+1} & = & dky_n(h_n) \end{array}$$

When no equation is given, streams keep their previous values.

This simulation interprets a hybrid node with an input of type I and an output of type O as a stream function. It is also possible to return the stream h as an extra output of this function.

Instead of taking a stream of values of type I, one can take a stream of values of type $(h : \mathit{Time}) \times ([0, h] \to I)$, that is, a duration $h : \mathit{Time} \subseteq \mathbb{R}^+$ and a function $f : [0, h] \to I$.

Instead of returning a stream of values of type O, one can return of stream of values of type $(h : \mathit{Time}) \times ([0, h] \to O)$.

This time, the f^d, f^z, f^{step}, f^{out} functions must be modified to take insto account that the input is continuously changing.

The f^{out} function is used in the integration mode to produce the output.

Alternatively

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

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

where the semantics value of an expression becomes of the form: $CoH(f^{d}, f^{z}, f^{out}, f^{step}, s, y)$

- f^d defines the derivative;
- f^z defines the zero-crossings;
- f^{out} defines the output from the current discrete state and continuous state;
- f^{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.

This is ongoing work

A preliminary prototype (June 2000); no hybrid constructs:
`https://github.com/marcpouzet/zrun`

A new one based on Zelus (Spring 2021):
`https://github.com/INRIA/zelus`, branch work. Hybrid constructs.

Purely functional OCaml code (except for code for debugging).

Use a generic library for the computation of fix-points [3]. Some preliminary work done by Antonin Reitz (Spring 2021).

Make the semantics more abstract, e.g.,:

- replace concrete values by a set (e.g., integers by intervals) in order to perform set-based simulation;
- replace concrete values by a symbolic expressions.

---

[3] The library Fix `https://gitlab.inria.fr/fpottier/fix` by Francois Pottier.

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.

Benveniste, A., Bourke, T., Caillaud, B., and Pouzet, M. (2011).
Divide and recycle: types and compilation for a hybrid synchronous language.
In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES'11)*, Chicago, USA.

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/`.