A Coiterative Synchronous Semantics for Scade (work in progress)

Marc Pouzet

Ecole normale supérieure
Paris

Marc.Pouzet@ens.fr

Cafein 2020
Les Angles

Objective

Give a direct executable (functional) semantics to a synchronous program.

Without having to compile: before scheduling, normalisation, inlining, etc.

Make proofs based on simple unfolding/computations.

Treat both data-flow and control structures (reset, hierarchical automata).

An old idea of Florence Maraninchi: execute unfinished programs.

E.g., programs that do have a semantics but are rejected by the compiler because its checks are overly constraining.

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.

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

- $f\ e$ 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* $\texttt{pre}_{\texttt{true}}(\texttt{false})$ *then* e_1 *else* e_2

Static Typing

Typing rules

We consider only first order functions.

$$
\begin{array}{lcl}
\sigma & ::= & \forall \alpha_1, ..., \alpha_n . gt \mid gt \\
gt & ::= & t \xrightarrow{k} t \mid t \\
t & ::= & t \times t \mid bt \mid \alpha \\
k & ::= & 0 \mid 1
\end{array}
$$

- $t_1 \xrightarrow{k} t_2$ with $k \in \{0, 1\}$ its sort is the type of a function.
- 0 means that the function is combinatorial;
- 1 means that the function is stateful;
- $(t_1 \times t_2)$ is the product type;
- bt is a base type (e.g., `bool`, `int`, `float`).

Historial note: Kinds were introduced in Lucid Synchrone [Pou06] in version 2 (2000); they are used in the type system of Scade 6 [CPP17].

Examples (in Zelus)

E.g., the following functions (written in Zelus) are well typed. [1]

```
let node from(x) =
  let rec f = x fby (f + 1) in f

let incr x = x + 1
```

On the contrary, the following is rejected.

```
let from(x) =
  let rec f = x fby (f + 1) in f
```

```
>  let rec f = x fby (f + 1) in f
>              ^^^^^^^^^^^^^^
Type error: this is a stateful discrete expression and
is expected to be combinatorial.
```

[1] The second form ask `incr` to be a combinatorial function, i.e., to have a type of the form $. \xrightarrow{0} .$

Semantics

We give a semantics to well-typed expressions and definitions only.

To simplify the presentation, we consider the same language but where every expression/sub-expression is annotated with its kind and type.

Streams processes

A *stream process* producing values in the set 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 $CoF(f, s)$, $Nth(v)(n)$ 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:

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

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.

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

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 the set $SFun(Unit, T, S)$ with $Unit$ the set with a single element `()`.

Fixpoint

Consider a synchronous stream function $f : S \to T \to T \times S$. Write $\mathit{fix}\,(f) : S \to T \times S$ for the smallest fix-point of f.

$\mathit{fix}\,(f)(s) = v, s'$ such that:

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

That is, given an initial state $s : S$, we want $\mathit{fix}\,(f)$ to be a solution of the following equation:

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

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

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

The value v is defined recursively.

Justification of its existence

In order to apply the Kleene theorem that state the existence of a smallest fix-point, all functions must be total.

$$Value(T) = \bot + \texttt{Some}(T)$$

$\bot$ is a short-cut for "Causality Error".

Define lifting functions.

$$\begin{array}{lcl}
lift_0(v) & = & \texttt{Some}(v) \\
lift_1(f)(\bot) & = & \bot \\
lift_1(f)(\texttt{Some}(v)) & = & \texttt{Some}(f(v)) \\
lift_2(f)(\bot, y) & = & \bot \\
lift_2(f)(x, \bot) & = & \bot \\
lift_2(f)(\texttt{Some}(v_1), \texttt{Some}(v_2)) & = & \texttt{Some}(f(v_1)(v_2))
\end{array}$$

That is, $\bot$ is absorbing and all functions applied point-wise are total.

Flat Order

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

$$\begin{array}{lll} \bot & \leq_T & x \\ \texttt{Some}(v) & \leq_T & \texttt{Some}(v) \end{array}$$

Shortcut: we write simply $\leq$.

Pairs:

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

Functions:

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

The bottom stream

The bottom element is:

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

Call $\bot_{CoStream(T,S)}$ 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 $\leq_{CoStream(T,S)}$ such that (noted $\leq$):

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

Define $\leq_{SFun(T,T,S)}$ such that (noted $\leq$):

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

If $f : SFun(Value(T), Value(T), Value(S))$ is continuous, $fix\,(f)$ exists.

Bounded Fixpoint

Yet, we cannot define the fix-point operator in Coq, at least as a function.

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:

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

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. 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.

ρ map values to identifiers.

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

$$
\begin{array}{lcl}
[\![\texttt{run}\ f\ e]\!]_\rho^{Init} & = & \rho(f)_I, [\![e]\!]_\rho^{Init} \\
[\![\texttt{run}\ f\ e]\!]_\rho^{State} & = & \lambda(m,s).let\ v,s = [\![e]\!]_\rho^{State}(s)\ in \\
 & & \qquad let\ r,m' = \rho(f)_S\ m\ v\ in \\
 & & \qquad r,(m',s)
\end{array}
$$

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

such that $s = [\![e]\!]_\rho^{Init}$

and $p = \lambda s, v.[\![e]\!]_{\rho+[v/x]}^{State}(s)$

Fixpoint

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

Using a recursion on value, it corresponds to:

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

Note that v is recursively defined

Control structure

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

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

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*:

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

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

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

with:

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

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

Where are the bottom values?

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", that is $fst(f\ s\ \bot) = \bot$.

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, $\bot < fst(f\ s\ \bot)$

We say that 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?

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

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

and is obtained after 4 iterations.

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

The inpact of static code generation

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

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

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:

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

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 $f : CoStream(T, S) \rightarrow CoStream(T, S')$ whose semantics is $CoP(f_t, s_t)$.

The semantics of an equation $y = f(y)$ is: [2]

$$\begin{array}{lcl} [\![\texttt{let rec } y \texttt{ = } f(y) \texttt{ in } y]\!]_\rho^{Init} & = & s_t \\ [\![\texttt{let rec } y \texttt{ = } f(y) \texttt{ in } y]\!]_\rho^{State} & = & \lambda s. let\ rec\ v, s' = f_t\ s\ v\ in\ v, s' \end{array}$$

[2] 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:

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

The recursion is no more necessary, that is:

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

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:

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

with initial state:

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

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

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?

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]

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

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)

val pid :
  {'a < 'b , 'c}. ('b -> 'c) -> ('a -> 'c) ->
                'c  * 'b  * 'd  * 'a -> 'c
```

To be continued

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.

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`.

Jean-Louis Colaco, Bruno Pagano, and Marc Pouzet.
Scade 6: A Formal Language for Embedded Critical Software Development.
In *Eleventh International Symposium on Theoretical Aspect of Software Engineering (TASE)*, Sophia Antipolis, France, September 13-15 2017.

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/`.

Marc Pouzet.
Lucid Synchrone, version 3. Tutorial and reference manual.
Université Paris-Sud, LRI, April 2006.
Distribution available at: `https://www.di.ens.fr/~pouzet/lucid-synchrone/`.