A Constructive State-based Semantics and Interpreter for a Synchronous Data-flow Language with State machines

Jean-Louis Colaco[1] Michael Mendler[2]

Baptiste Pauget[1,3] Marc Pouzet[3]

1. ANSYS, Toulouse, France
2. University of Bamberg, Germany
3. ENS-PSL and INRIA, Paris, France

EMSOFT 2023, Hamburg, Sept. 20

Synchronous Prog. of Reactive Systems [Halbwachs, 93]

Synchronous Prog. of Reactive Systems [Halbwachs, 93]

Domain specific languages for reactive control software;

Synchronous Prog. of Reactive Systems [Halbwachs, 93]

Domain specific languages for reactive control software;

a program is an ideal deterministic and synchronous (zero-delay) model;

Synchronous Prog. of Reactive Systems [Halbwachs, 93]

Domain specific languages for reactive control software;

a program is an ideal deterministic and synchronous (zero-delay) model;

a reference specification for

Synchronous Prog. of Reactive Systems [Halbwachs, 93]

Domain specific languages for reactive control software;

a program is an ideal deterministic and synchronous (zero-delay) model;

a reference specification for

validation, e.g., simulation/testing/formal proofs

Synchronous Prog. of Reactive Systems [Halbwachs, 93]

Domain specific languages for reactive control software;

a program is an ideal deterministic and synchronous (zero-delay) model;

a reference specification for

validation, e.g., simulation/testing/formal proofs

and the generation of executable embedded code.

The compiler plays a central role

The compiler plays a central role

- It generates an implementation (e.g., C code).

The compiler plays a central role

- It generates an implementation (e.g., C code).
- It statically reject models for which it cannot ensure important safety properties, e.g.:

The compiler plays a central role

- It generates an implementation (e.g., C code).
- It statically reject models for which it cannot ensure important safety properties, e.g.:
- the program does react (no deadlock),

The compiler plays a central role

- It generates an implementation (e.g., C code).
- It statically reject models for which it cannot ensure important safety properties, e.g.:
- the program does react (no deadlock),
  the reaction is unique (determinacy),

The compiler plays a central role

- It generates an implementation (e.g., C code).
- It statically reject models for which it cannot ensure important safety properties, e.g.:
- the program does react (no deadlock),
  the reaction is unique (determinacy),
  the generated sequential code runs in bounded time and space.

The compiler plays a central role

- It generates an implementation (e.g., C code).
- It statically reject models for which it cannot ensure important safety properties, e.g.:
- the program does react (no deadlock),
  the reaction is unique (determinacy),
  the generated sequential code runs in bounded time and space.
- Checks are part of the language specification: if they fail, no code is generated.

The compiler plays a central role

- It generates an implementation (e.g., C code).
- It statically reject models for which it cannot ensure important safety properties, e.g.:
- the program does react (no deadlock),
  the reaction is unique (determinacy),
  the generated sequential code runs in bounded time and space.
- Checks are part of the language specification: if they fail, no code is generated.
- Compiler correctness is a difficult challenge.

The compiler plays a central role

- It generates an implementation (e.g., C code).
- It statically reject models for which it cannot ensure important safety properties, e.g.:
- the program does react (no deadlock),
  the reaction is unique (determinacy),
  the generated sequential code runs in bounded time and space.
- Checks are part of the language specification: if they fail, no code is generated.
- Compiler correctness is a difficult challenge.
- E.g., a compiler like Scade is used in certified applications.

The compiler plays a central role

- It generates an implementation (e.g., C code).
- It statically reject models for which it cannot ensure important safety properties, e.g.:
- the program does react (no deadlock),
  the reaction is unique (determinacy),
  the generated sequential code runs in bounded time and space.
- Checks are part of the language specification: if they fail, no code is generated.
- Compiler correctness is a difficult challenge.
- E.g., a compiler like Scade is used in certified applications.

How to ensure it for a language and compiler that needs to evolve?

A lot of work has been done on the formalization of semantics for synchronous languages!

A lot of work has been done on the formalization of semantics for synchronous languages!

on paper and/or with computer-checked proofs (cf talk by B. Pesin Monday).

This work

- We consider a first-order synchronous functional language
- a large subset of Scade.

Objective

- A formal semantics that is executable, i.e., an interpreter;
- that is constructive, i.e., defined as a total function in a typed functional language with strong normalization (all program terminate);
- that can apply directly to the source, before compilation starts;
- is independent of a compiler.

For what?

- An oracle for compiler testing (e.g., fuzzing);
- to explore new language constructs before implementing them;
- to execute partial models or that are rejected by the compiler;
- to prove compiler steps.

A prototype in OCaml

- A reference implementation in purely functional style.
- `https://zelus.di.ens.fr/zrun/emsoft2023`
- `https://zelus.di.ens.fr/zrun/emsoft2023/work`

A synchronous language kernel

A first-order Lustre-like synchronous language of streams.

Global declarations (d), patterns (p), expressions (e), equations (E), immediate constants (v):

$$
\begin{array}{llll}
(\text{declaration}) & d & ::= & \texttt{let}\ f = e \mid \texttt{let node}\ f\ \ p\ \texttt{=}\ e \mid d\ d \\
(\textit{pattern}) & p & ::= & () \mid x \mid x, ..., x \\
(\textit{expression}) & e & ::= & v \mid x \\
 & & & \mid f\,(e, ..., e) \mid (e, ..., e) \mid () \\
 & & & \mid e\ \texttt{fby}\ e \\
 & & & \mid \texttt{let rec}\ E\ \texttt{in}\ e \\
(\textit{equation}) & E & ::= & p\ \texttt{=}\ e \mid E\ \texttt{and}\ E
\end{array}
$$

Examples

```
let node forward_euler(h, x0, x') =
  (* [x(0) = x0(0)
     /\ forall n in Nat. x(n) = x(n-1) + h(n-1) * x'(n-1)] *)
  let rec x = x0 fby (x +. h *. x') in x

let node backward_euler(h, x0, x') =
  (* [x(0) = x0(0)
     /\ forall n in Nat. x(n) = x(n-1) + h(n) * x'(n)] *)
  let rec x = x0 -> pre(x) +. h *. x' in x

let node pi(p, i, u) = p *. u +. backward_euler(h, 0.0, i *. u)

let node sin_cos(h) =
  let rec sin = forward_euler(h, 0.0, cos)
  and cos = backward_euler(h, 1.0, -. sin) in
  (sin, cos)
```

Which semantics ?

A signal s is an infinite sequence:

$$stream(T) \stackrel{def}{=} \mathbb{N} \rightarrow T$$

or the solution of the fix-point equation:

$$stream(T) = T \times stream(T)$$

It is an infinite object.

A deterministic system is a stream function:

$$system(T, T') \stackrel{def}{=} stream(T) \rightarrow stream(T')$$

Parallel and feed-forward composition are easy = function composition.

Feedback: a function $fix\,(.) : (stream(T) \rightarrow stream(T)) \rightarrow stream(T)$

such that $fix\,(f)$ is a solution of the stream equation:

$$x = f(x)$$

Feedback

$fix\,(f)$ does not always exist, e.g., $f = \lambda x.\lambda n.x(n) + 1$.

Idea: complete a set T with $\perp$ to explicitly represent an undefined value (e.g., divergence, deadlock);

A 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 $(D, \leq, \perp)$ is a Complete Partial Order (CPO) and f, a continuous function $f : D \to D$.

It has a lfp $(fix\,(f) = \lim_{n\to\infty}(f^n(\perp))$.

This is not an effective computational definition because the height of D may be unbounded

E.g., the CPO of streams $(stream(T_\perp), \leq_s, \perp_s)$, with $\perp_s = \lambda n.\perp$ is the bottom stream and $\leq_s$ the prefix order:

$$x \leq y \text{ iff } \forall n \in \mathbb{N}.x(n) \neq y(n) \Rightarrow \forall m \geq n.x(n) = \perp$$

How to define $\mathit{fix}\,(.)$ constructively, as a total function?

How to define $\mathit{fix}\,(.)$ constructively, as a total function?

where the meta-language is a statically typed functional language

How to define *fix* (.) constructively, as a total function?

where the meta-language is a statically typed functional language

with strong normalization, i.e., all functions terminate?

How to define *fix* (.) constructively, as a total function?

where the meta-language is a statically typed functional language

with strong normalization, i.e., all functions terminate?

e.g., the programming language of Coq.

We used ideas introduced in several works

The PhD. thesis of Georges Gonthier [Gonthier, 1988] who introduced the idea of a computational semantics for Esterel.

The paper "Circuits as streams in Coq, verification of a sequential multiplier" by Christine Paulin [Paulin-Mohring, 1995].

The paper "a Coiterative Characterization of Synchronous Stream Functions" by Paul Caspi and Marc Pouzet [Caspi and Pouzet, 1998].

"The semantics and execution of a synchronous block-diagram language", by Stephen Edwards and Edward Lee [Edwards and Lee, 2003]

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:

$$\begin{array}{lcl} nth(CoF(f, s))(0) & = & let\, v, s = f\ s\, in\, v \\ nth(CoF(F, s))(n) & = & let\, v, s = f\ s\, in\, nth(CoF(f, s))(n-1) \\ concrete(v) & = & CoF(\lambda n.(v(n), (n+1)), 0) \end{array}$$

$$\begin{array}{lcl} nth(.)(.) & : & coStream(T, S) \to stream(T) \\ concrete(.) & : & stream(T) \to coStream(T, \mathbb{N}) \end{array}$$

One can go from a stream-based to a concrete-based interpretation, and back.

Definition (Equivalence)

Two concrete streams $CoF(f, s)$ and $CoF(f', s')$ are equivalent iff they produce the same stream:

$$nth(CoF(f, s)) = nth(CoF(f', s'))$$

We write $CoF(f, s) \cong CoF(f', s')$ for equivalence of concrete streams.

Taking $stream(x)(n) \stackrel{def}{=} nth(x)(n)$, $concrete(stream(x)) \cong x$ and $stream(concrete(x)) = x$.

Find an inductive relation R that is verified on initial states, that is, $R(s, s')$ and preserved by the application of the two transition functions.

$$\forall s, s'. R(s, s') \Rightarrow (fst\ o\ f)(s) = (fst\ o\ f')(s') \wedge R((snd\ o\ f)(s), (snd\ o\ f')(s'))$$

E.g., to prove that forall $x, y. 0 \texttt{ fby } (x + y) \cong (0 \texttt{ fby } x) + (0 \texttt{ fby } y)$, take $R(s1 + s2, (s1, s2))$.

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 class of synchronous or length preserving functions.

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

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

Synchronous Application

A value $f = CoP(ft, s)$ defines a stream function:

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

with

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

$run(.)(.)$ convert a synchronous function into a stream function.

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 want to define $fix\,(.)$ such that $fix\,(f)$ is a fixpoint of f, that is:

$$fix\,(f) = f(fix\,(f))$$

Suppose that f is the image of a synchronous function, that is, it exists $CoP(ft, s)$ such that $f\ y \cong run(CoP(ft, s_0))(y)$.

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

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

Consider $f = CoP(ft, s)$ with $ft : S \to T \to T \times S$.

We want $feedback\,(ft) : S \to T \times S$ such that

$$feedback\,(ft)\,(s) = v, s'$$

where

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

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

$$feedback\,(ft) = \lambda s.let\ rec\ v, s' = ft\ s\ v\ in\ v, s'$$

where v is defined recursively.

$CoF(feedback\,(ft), s)$ computes 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 at every instant.

It can be programmed directly in Haskell (or OCaml, using module `Lazy`);

it leads to an interpretor.

Where is the devil?

feedback (.) is not a total function, e.g., it diverges or deadlocks.

For example, *feedback* $(\lambda s, x.x + 1, s)$ which corresponds to:

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

Or *feedback* $(\lambda s, (x, y).(y, x), s)$:

```
let f () =
  let rec x = y and y = x in (x, y)
```

As a consequence, we cannot define *feedback* (.) in Coq.

Bounded Fixpoint

By restricting to length preserving functions, we have made the problem of computing the fix-point function simpler.

Indeed, if $(D, \leq, \perp)$ is a CPO of bounded height, the fixpoint can be reached in a finite number of steps.

We have replaced an unbounded iteration by an bounded one [Caspi and Pouzet, 1998].

The idea of computing a fix-point for every reaction was introduced in a "computable semantics" for Esterel [Gonthier, 1988], lately called "constructive semantics" [Berry, 1999]).

This idea of bounded iteration was exploited in [Edwards 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:

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

How many iterations?

It depends on the type T. Define:

$$\begin{array}{lcl} \|int\| & = & 0 \\ \|T_\perp\| & = & 1 + \|T\| \\ \|T_1 \times T_2\| & = & \|T_1\| + \|T_2\| \\ \|T_1 \to T_2\| & = & \|T_2\|^{|T_1|} \end{array}$$

where $|T|$ is the cardinality of T.

It is enough to give only a credit of $\|T\|$ iterations for a fixpoint on a value of type T.

For n recursively defined stream variables, iterate n times at most.

Continuity is replaced by monotony and the function $fix\,(.)$ is total.

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 $CoP(p, s)$.

Semantics of Expressions

$$\llbracket v \ \texttt{fby} \ e \rrbracket^{Init}_{\rho} \stackrel{def}{=} (v, \llbracket e \rrbracket^{Init}_{\rho})$$

$$\llbracket v \ \texttt{fby} \ e \rrbracket^{State}_{\rho}(m, s) \stackrel{def}{=} (m, let\ v, s = \llbracket e \rrbracket^{State}_{\rho}(s)\ in\ (v, s))$$

$$\llbracket x \rrbracket^{State}_{\rho}(s) \stackrel{def}{=} (\rho(x), s)$$

$$\llbracket c \rrbracket^{Init}_{\rho} \stackrel{def}{=} ()$$

$$\llbracket c \rrbracket^{State}_{\rho}(s) \stackrel{def}{=} (c, s)$$

$$\llbracket (e_1, ..., e_2) \rrbracket^{Init}_{\rho} \stackrel{def}{=} (\llbracket e_1 \rrbracket^{Init}_{\rho}, ..., \llbracket e_2 \rrbracket^{Init}_{\rho})$$

$$\llbracket (e_1, ..., e_2) \rrbracket^{State}_{\rho}(s) \stackrel{def}{=} let\ (v_i, s_i = \llbracket e_i \rrbracket^{State}_{\rho}(s_i))_{i \in [1..n]}\ in\ (v_1, ..., v_n), (s_1, ..., s_n)$$

Node application

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

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

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

Equations

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

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

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

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

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

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

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

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

A Complete Language

This semantics extends to a rich language, e.g., that mix a data-flow and control-flow programming style.

by-case definition of streams with default value (see paper);

hierarchical automata (see paper);

arrays and iterators, static parameters [1].

[1] `https://zelus.di.ens.fr/zrun/emsoft2023`

A focus on Causality

It has been the subject of strong debates!

Dynamic causality (is a value $\perp$?) vs static causality (what the compiler can approximate safely).

There is no absolute notion of causality: there is not one that is better than the other.

Some are more powerful (they accept more program); but at the price of a greater complexity, lack of modularity of analysis and code generation.

The choice is determined by the code you target, e.g., circuit or software.

For circuits, if cyclic circuits are forbidden by the synthesis tool, why fighting for constructive causality?

For software, different compromises, e.g., code size of the target code.

Demo

Examples available at:
`https://zelus.di.ens.fr/zrun/emsoft2023/work/tests/good/`.

A simple counter, etc.
`https://zelus.di.ens.fr/zrun/emsoft2023/work/tests/good/ex0.zls`.

The cyclic circuit of Malik.
`https://zelus.di.ens.fr/zrun/emsoft2023/work/tests/good/malik.zls`.

The Bus arbiter by R. de Simone.
`https://zelus.di.ens.fr/zrun/emsoft2023/work/tests/good/arbiter.zls`.

Type `zrun.exe -s main -n 10 arbiter.zls`

The causality in Lustre vs Signal vs Esterel correspond to different interpretations of the conditional.

With `zrun`, you can try several.

Syntactic Causality (Lustre)

$$
\begin{array}{lcl}
{}^{\star}\mathit{if}\ \bot\ \mathit{then}\ \_\ \mathit{else}\ \_ & \stackrel{def}{=} & \bot \\
{}^{\star}\mathit{if}\ \_\ \mathit{then}\ \bot\ \mathit{else}\ \_ & \stackrel{def}{=} & \bot \\
{}^{\star}\mathit{if}\ \_\ \mathit{then}\ \_\ \mathit{else}\ \bot & \stackrel{def}{=} & \bot \\
{}^{\star}\mathit{if}\ \mathit{true}\ \mathit{then}\ x\ \mathit{else}\ \_ & \stackrel{def}{=} & x \\
{}^{\star}\mathit{if}\ \mathit{false}\ \mathit{then}\ \_\ \mathit{else}\ y & \stackrel{def}{=} & y
\end{array}
$$

Causality

Lazy Causality

$$
\begin{aligned}
{}^{\star}\mathit{if}\ \perp\ \mathit{then}\ \_\ \mathit{else}\ \_ &\stackrel{\mathit{def}}{=} \perp \\[1em]
{}^{\star}\mathit{if}\ \mathit{true}\ \mathit{then}\ x\ \mathit{else}\ \_ &\stackrel{\mathit{def}}{=} x \\
{}^{\star}\mathit{if}\ \mathit{false}\ \mathit{then}\ \_\ \mathit{else}\ y &\stackrel{\mathit{def}}{=} y
\end{aligned}
$$

Constructive Causality

$$
\begin{aligned}
{}^{\star}\mathit{if}\ \perp\ \mathit{then}\ v_1\ \mathit{else}\ v_2 &\stackrel{\mathit{def}}{=} \mathit{if}\ v_1 = v_2\ \mathit{then}\ v_1\ \mathit{else}\ \perp \\[1em]
{}^{\star}\mathit{if}\ \mathit{true}\ \mathit{then}\ x\ \mathit{else}\ \_ &\stackrel{\mathit{def}}{=} x \\
{}^{\star}\mathit{if}\ \mathit{false}\ \mathit{then}\ \_\ \mathit{else}\ y &\stackrel{\mathit{def}}{=} y
\end{aligned}
$$

where $v_1 = v_2$ must be decidable.

Causality

With the following definition for the or/and operations:

$$
\begin{array}{rcl}
{}^{\star}or(x, y) & \stackrel{def}{=} & \textit{if } x \textit{ then true else } y \\
{}^{\star}and(x, y) & \stackrel{def}{=} & \textit{if } x \textit{ then } y \textit{ else false}
\end{array}
$$

With the first interpretation, the two operators are strict. With the second one, they are sequential, left-to-right; with the third one, it corresponds to the 3-valued logic for boolean operators.

$$
\begin{array}{rcl}
{}^{\star}or(true, \_) & = & true \\
{}^{\star}or(\_, true) & = & true \\
{}^{\star}or(false, x) & = & x \\
{}^{\star}or(x, false) & = & x \\
{}^{\star}and(false, \_) & = & false \\
{}^{\star}and(\_, false) & = & false \\
{}^{\star}and(true, x) & = & x \\
{}^{\star}and(x, true) & = & x
\end{array}
$$

Constructive Causality (Esterel)

The following program:

```
tobe = tobe or not tobe
```

is not causally correct in Esterel. Neither it is with the third encoding.

But the following one, that is not causally correct in Esterel:

```
x = if x then true else true
```

is with the third encoding.

Which one is better? no one.

It depends on the primitive operations of the target platform.

There are many and comparable ways of lifting a primitive
$f : T^1 \times ... \times T^n \to T$ into
${}^{\star}f : T^1_\perp \times ... \times T^n_\perp \to T_\perp$ [Schneider et al., 2005]

each with its own impact in term of static analyses and code generation.

Conclusion

- An executable semantics for a data-flow synchronous language.
- The input language has the main programming constructs of Scade.
- Constructiveness is a consequence that the semantics is expressible in a statically typed functional language with strong normalization, e.g., Coq.
- The semantics is rather abstract. By changing what is a value and how functions are lifted, we can experiment different run-time causalities.
- The state-based and co-iterative approach works surprisingly well.
- It can deal with error management with little change of the code.
- We prototyped several new language constructs not in Scade, in particular array operations.

References I

Berry, G. (1999).
The constructive semantics of pure esterel.
Draft book. Available at:
http://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.pdf.

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.

Edwards, S. A. and Lee, E. A. (2003).
The semantics and execution of a synchronous block-diagram language.
Science of Computer Programming, 48:21–42.

Gonthier, G. (1988).
Sémantiques et modèles d'exécution des langages réactifs synchrones.
PhD thesis, Université d'Orsay.

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/.

Schneider, K., Brandt, J., Schüle, T., and Türk, T. (2005).
Maximal causality analysis.
In *Conference on Application of Concurrency to System Design (ACSD'05)*, pages 106–115, St. Malo, France.