Causality analysis

Marc Pouzet

ENS

Marc.Pouzet@ens.fr

Notes de cours
MPRI
October 2024

Three practical questions

- Determinacy: is the system deterministic, e.g., given the very same sequence of inputs, it produces the same sequence of outputs?
- Deadlock freedom: is the system reactive (or productive), that is, at every instant and for any valid input, it produces an output?
- Finally, is it possible to generate code that run in bounded time and space?

The problem is simpler than in an asynchronous model because all processes in parallel run in lock step.

Analyse instantaneous dependences only: does the current output of a signal depends instantaneously on itself?

Causality in synchronous programs

There have been a lot of work. Read [2]!

Several answers have been given.

The simplest is that of Lustre (and its successor): statically reject instantaneous (unconditional) cycle. It can be expressed as a dedicated type system; moreover, it combines quite well with higher-order.

Esterel and Signal have experimented a more general definition. Cycles are allowed under certain conditions. We illustrate it with Zrun [1] interpreter.

The static analysis is more complicated and involve boolean reasonning. Its expression as a type system applied modularily is an open question.

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

Two examples in Esterel

The program P13 from the Esterel primer V5.91.
`https://github.com/marcpouzet/zrun/blob/master/tests/esterel-primer-p13.zls`

```
Section 5.1.4
 module P13:
   input I;
   output O1, O2;
   present I then
      present O2 then emit O1 end
   else
      present O1 then emit O2 end
   end present
 end module
```

This program is constructively causal.

The program P14 from the Esterel primer V5.91.
https://github.com/marcpouzet/zrun/blob/master/tests/esterel-primer-p14.zls

```
module P14:
  output O1, O2;
  present O1 then emit O2 end;
  pause;
  present O2 then emit O1 end
 end module
```

The cyclic circuit of Malik

https://github.com/marcpouzet/zrun/blob/master/tests/fog_gof.zls

```
let node mux(c, x, y) returns (o)
  if c then o = x else o = y

let node f(x) returns (o) o = 2 * x
let node g(x) returns (o) o = x - 1

let node fog_gof(c, x) returns (y)
  local x1, x2, y1, y2
  do 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)
  done

(* Same output with no cycle for reference *)
let node fog_gof_ref(c, x) returns (y)
  y = mux(c, g(f(x)), f(g(x)))
```

The cyclic token ring arbiter

`https://github.com/marcpouzet/zrun/blob/master/tests/arbiter.zls`

Constructive causality

```
(* Constructiveness in the sense of Esterel *)
(* Verbatim from The Esterel Primer, V5.91, Berry, 2000
*-
*- 1. An unknown signal can be set present if it is emitted.
*- 2. An unknown signal can be set absent if no emitter
      can emit it.
*- 3. The then branch of a test can be executed if the
      test is executed and the signal is present.
*- 4. The else branch of a test can be executed if the
      test is executed and the signal is absent.
*- 5. The then branch of a test cannot be executed if the
      signal is absent.
*- 6. The else branch of a test cannot be executed if the
      signal is present.
*)

(* Moreover, Esterel makes a special treatment of the two
   boolean operators
*- (or and &) that are considered parallel and not sequential.
*)
```

The problem

Some equations deadlock like `x = x + 1` or `x = y and y = x`.

That is, the transition functiom produces bottom values.

Causality analysis has two objectives in a synchronous language compiler.

1. find sufficient conditions to ensure that the transition function produce non bottom values.
2. generate statically schedule code.

In Lustre and Lucid Synchrone, the causality analysis is performed after the clock-calculus and independently of it.

It considers that stream operations are *length preserving*: either the output instantaneously depend on the input or not.

Moreover, causality analysis does not depend on values of streams.

The simplified problem

It amount at considering a language with only two basic operations.

- Lifting: lift a scalar into a constant stream; lift a n-ary function to apply it pointwise.
- A unit delay, initialized or not. E.g., `(x : xs) fby ys = x : ys`.
- function definition, possibly higher-order, application and a fix-point operator for defining mutually recursive streams only.

Two questions

1. Detect and reject stream equations that are not productive, i.e., ensure that all streams are infinite; "reactivity and determinism"
2. Generate statically scheduled code which compute a stream step by step. Recursion and laziness forbidden. "compile the parallelism"

Several works address the question of productivity and proof techniques for languages manipulating infinite data structures.

Hamming's exercise in SASL. [Dijkstra, 1981]

On the productivity of recursive list definitions. [Sijtsma, 1989].

Proving the correctness of reactive systems using sized types. [Hugues & Pareto & Sabry, 1996];

Guarded recursion in proof assistants. E.g.,:

Infinite objects in type theory. [Coquand, 1993];

Structural recursive definitions in type theory. [Gimenez, 1994];

Termination checking in the presence of nested inductive and coinductive types. [Danielson, Altenkirch, 2010];
Beating the Productivity Checker Using Embedded Languages. [Danielson, 2010];
(Many others: Abel, Bertot, Buchholz, Di Gianantonio & Miculan, Hancock & Pattinson & Ghani, McBride, Morris & Altenkirch & Ghani, etc.)

Related works

Those works address question (1) for a more general language where stream functions can be length preserving or not and/or mixed with inductive structures.

E.g: is the following equation productive?

```
x = 0 : 1 : tl x
```

where `tl (x : xs) = xs`

We only have an operator "delay" that make streams longer but not shorter. [2]

We adress a simpler problem: we forbid to write `tl` which is not length preserving.

Those works do not address question (2) which is specific to infinite streams programs.

---

[2] `tl` can be defined by `tl x = x when (false fby true)`. `when` is not a length preserving function.

Operators

Hence, the language has essentially the following features:

1. Define mutually recursive equations;
2. point-wise application of an operations (e.g., $+$);
3. unit delay: `pre`, `fby`.
4. The non length preserving operators: `when` and `merge` are considered as if they were length preserving, from the causality analysis point-of-view.

A trivial solution

Build a dependence graph from the syntax such that:

- For every equation $x = e$, state that x depends on all variables appearing in e but those on the right of a unit delay (`pre` or `fby`).
- compute the transitive closure;
- reject recursive definitions if the corresponding graph is cyclic.

This solution is easy to implement. It accepts:

```
let node int(x') = x where
  rec x = 0 fby (x' + 1)

let node fix(g)(x0) = x where
  rec x = g(x0 fby x) in x
```

But rejects:

```
let node f(x) = (y, z) where
  rec (y, z) = let t = x + 1 in (z, t)

let node copy(x, y) = (x, y)

let node main(x) = (t, u) where
  rec = copy(x, t)
```

It is very sensitive to naming and the syntactic structure. It does not treat modularity — the ability to define a function, compute some information about it once and reuse it later.

We propose a *type based representation of input/output dependences*.

The idea of representing causality information as a type was firt introduced in the language Lucid Synchrone [4].

This is the way it is done in Scade [3] but consider a first-order language only.

Here, we go a bit further by considering higher order with a new formulation of dependences and algorithm for type simplification.

A few examples in Zélus [3]

[3] zelus.di.ens.fr

Summary

- Represent the instantaneous input/output depependence by a type
- A stream expression is associated to a tag.
- Tags must be partially ordered.

Examples

```
let node forward_euler(t)(k, x0, u) = output where
  rec output = x0 -> pre (output +. (k *. t) *. u)

let node backward_euler(t)(k, x0, u) = output where
  rec output = x0 -> (pre output) +. (k *. t) *. u

let node filter(n)(h)(k, u) = udot where
  rec udot = n *. (k *. u -. f)
  and f = forward_euler(h)(n, 0.0, udot)

val forward_euler : {}. 'a -> 'a  * 'b  * 'a -> 'b
val backward_euler : {}. 'a -> 'a  * 'a  * 'a -> 'a
val filter : {}. 'a -> 'b -> 'b  * 'b -> 'b
```

Summary

```
let node bad_filter(n)(h)(k, u) = udot where
  rec udot = n *. (k *. u -. f)
  and f = backward_euler(h)(n, 0.0, udot)
```

```
File "examples.zls", line 17, characters 10-41:
>  and f = backward_euler(h)(n, 0.0, udot)
>          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Causality error: This expression has causality type
'c, whereas it should be less than 'd
Here is an example of a cycle:
f at 'd < udot at 'c; udot at 'c < f at 'd
```

Summary

2/ A function can have an argument which is a function.

```
let node gfilter(int)(h)(n)(k, u) = udot where
  rec udot = n *. (k *. u -. f)
  and f = run (int(h)) (n, 0.0, udot)

let node gpid(int)(filter)(h)(p, i, d, u) = c where
  rec c_p = p *. u
  and i_p = run (int h)(i, 0.0, u)
  and c_d = run (filter h)(d, u)
  and c = c_p +. i_p +. c_d

val gfilter :
{'a < 'b}. ('c -> 'a  * 'd  * 'b -> 'b) -> 'c ->
           'a -> 'b  * 'b -> 'b
val gpid :
  {'a < 'b}.
    ('c -> 'd  * 'e  * 'a -> 'b) ->
    ('c -> 'f  * 'a -> 'b) -> 'c ->
    'b  * 'd  * 'f  * 'a -> 'b
```

Summary

```
let node filter_forward(h)(n)(k, u) =
  generic_filter(forward_euler)(h)(n)(k, u)

val filter_forward : {'a < 'b}. 'b -> 'a -> 'a * 'a -> 'a

(* This program is not causal *)
(* let node filter_backward(h)(n)(k, u) =
  generic_filter(backward_euler)(h)(n)(k, u) *)
```

```
>  gfilter(backward_euler)(h)(n)(k, u)
>          ^^^^^^^^^^^^^^^^
Causality error: This expression has causality type
'c -> 'd * 'e * 'f -> 'g, whereas it should be less than
'h -> 'i * 'j * 'k -> 'l
Here is an example of a cycle:
k < f; f < g; g < l; l < k
```

Remove administrative relations: `f < g` and `l < k` are contradictory

Examples in continuous time

The analysis is the same for functions on discrete-time and continuous-time signals. E.g.,:

```
let hybrid gfilter_c(int)(n)(k, u) = udot where
   rec udot = n *. (u -. f)
   and f = run int (k, 0.0, udot)

let hybrid gpid_c(int)(filter)(n)(p, i, d, u) = c where
  rec c_p = p *. u
  and i_p = run int(i, 0.0, u)
  and c_d = run (filter(n))(d, u)
  and c = c_p +. i_p +. c_d

val gfilter_c : {}.
  ('a  * 'b  * 'c -> 'c) -> 'c -> 'a  * 'c -> 'c
val gpid_c :
  {'a < 'b}.
    ('c  * 'd  * 'a -> 'b)
    -> ('e -> 'f  * 'a -> 'b) -> 'e
    -> 'b  * 'c  * 'f  * 'a -> 'b
```

Examples in continuous time

```
let hybrid int(k, x0, xprime) = x where
  rec der x = k *. xprime init x0

let hybrid pid_c(n)(p, i, d, u) =
  gpid_c(int)(gfilter_c(int))(n)(p, i, d, u)

val int : {}. 'a  * 'b  * 'a -> 'b
val pid_c : {}. 'a -> 'a  * 'b  * 'b  * 'a -> 'a
```

A language kernel

Definition of functions; variables, constant, application, fix-point, tuples and access functions.

$$
\begin{array}{lcl}
d & ::= & \texttt{let}\ f\ x = e \mid d; d \\
\\
e & ::= & x \mid v \mid \texttt{let rec}\ x = e\ \text{in}\ e \\
& & \mid (e, e) \mid \texttt{fst}(e) \mid \texttt{snd}(e) \\
& & \mid e(e) \mid e\ \texttt{fby}\ e
\end{array}
$$

v stand for values.

Typing constraints so that $\texttt{let rec}\ x = e\ \text{in}\ e'$ is limited such that the type of e is bounded: it has no function type in it.

Expressing Dependences/Causality with a type

Since all stream operations are length preserving, express instantaneous dependences only.

The dependence relation is a partial order.

Represent the instantaneous dependences of an expression by a type.

$$
\begin{array}{lcl}
bt & ::= & \alpha \\
t & ::= & bt \mid t \times t \mid t \to t \\
\sigma & ::= & \forall \alpha_1, ..., \alpha_n : C.t \mid t \\
\\
C & ::= & \{\alpha_i < \alpha_j\}_{i,j \in I}
\end{array}
$$

$\alpha_1, ..., \alpha_n, ...$ are tags (a tag is a "time stamp").

C must define a partial order (acyclic graph) between those tags.

Initial conditions

$$
\begin{array}{lcl}
(+) & : & \forall\alpha.\alpha \times \alpha \to \alpha \\
\texttt{if . then . else .} & : & \forall\alpha.\alpha \times \alpha \times \alpha \to \alpha \\
\texttt{pre}\ \cdot & : & \forall\alpha_1, \alpha_2 : \{\alpha_2 < \alpha_1\}.\alpha_1 \to \alpha_2 \\
\cdot\ \texttt{fby}\ \cdot & : & \forall\alpha_1, \alpha_2 : \{\alpha_1 < \alpha_2\}.\alpha_1 \times \alpha_2 \to \alpha_1
\end{array}
$$

The typing predicate: $C, H \vdash e : t$, where:

$H = [x_1 : \sigma_1, ..., x_n : \sigma_n]$ and $Acyclic(C)$ as an implicit side condition.

(fundef)

$$\frac{C, H[x : t_1] \vdash e : t_2}{H \vdash \texttt{let}\ f\ x = e : H[f : Gen(C)(t_1 \to t_2)]}$$

(app)

$$\frac{C, H \vdash f : t_1 \to t_2 \qquad C, H \vdash e : t_1}{C, H \vdash f\ e : t_2}$$

(const)

$$C, H \vdash v : bt$$

(tuple)

$$\frac{C, H \vdash e_1 : t_1 \quad C, H \vdash e_2 : t_2}{C, H \vdash (e_1, e_2) : t_1 \times t_2}$$

(var)

$$\frac{C_x, t \in \mathit{Inst}(\sigma)}{C + C_x, H[x : \sigma] \vdash x : t}$$

(sub)

$$\frac{C, H \vdash e : t_1 \quad C \models t_1 < t_2}{C, H \vdash e : t_2}$$

(rec)

$$\frac{C, H[x : t] \vdash e : t_e \quad C \models t_e < t \qquad C, H[x : t_e] \vdash e' : t'}{C, H \vdash \texttt{let rec}\ x = e\ \text{in}\ e' : t'}$$

Generalisation

$$\mathit{Gen}(C)(t) \quad = \quad \forall \alpha_1, ..., \alpha_n : C.t \ \text{where} \ \mathit{Vars}(t) = \{\alpha_1, ..., \alpha_n\}$$
$$\text{provided } \mathit{Acyclic}(C)$$

Instanciation

$$C[\vec{\alpha'}/\vec{\alpha}], t[\vec{\alpha'}/\vec{\alpha}] \in \mathit{Inst}(\forall \vec{\alpha} : C.t) \text{ provided } \mathit{Acyclic}(C[\vec{\alpha'}/\vec{\alpha}])$$

The dependence order

The relation is strict.

(tuple)
$$\frac{C \models t_1 < t_1' \qquad C \models t_2 < t_2'}{C \models t_1 \times t_2 < t_1' \times t'2}$$

(fun)
$$\frac{C \models t_2 < t_2' \qquad C \models t_1' < t_1}{C \models t_1 \to t_2 < t_1' \to t_2'}$$

(trivial)
$$C[\alpha_1 < \alpha_2] \models \alpha_1 < \alpha_2$$

(trans)
$$\frac{C \models t_1 < t_2 \qquad C \models t_2 < t_3}{C \models t_1 < t_3}$$

Strict order vs non strict order

We only consider a strict order because it is enough to answer the question "is there an instantaneous feedback?".

A more conventional system would use both $<$ and $\leq$, replacing rules (*sub*) and (*rec*) by:

$$\text{(sub)}\quad \frac{C, H \vdash e : t_1 \quad C \models t_1 \leq t_2}{C, H \vdash e : t_2}$$

$$\text{(rec)}\quad \frac{C, H[x : t] \vdash e : t_e \quad C \models t_e < t \qquad C, H[x : t_e] \vdash e' : t'}{C, H \vdash \texttt{let rec}\ x = e\ \text{in}\ e' : t'}$$

Problems

Sub-typing constraints have to be simplified.

The type system for causality is similar to a type system with intersection and union types. The relation:

- $t_1 < t \wedge t_2 < t$ corresponds to $t_1 \cup t_2 < t$;
- $t < t_1 \wedge t < t_2$ corresponds to $t < t_1 \cap t_2$.

The current system do not have relations of the form $\alpha < t$ or $t < \alpha$, where t is not a variable.

The reason is that causality typing is done after typing: we use the type structure to construct causality skeleton types.

Type simplification for systems with intersection/union types has been studied a lot, in particular by Aiken & Wimmers, Pottier, Smith & Trifonov, Castagna et al.

Input/Output relation

We apply the simplification algorithm that uses the Input/output relation of Pouzet & Raymond [5].

$InOut(p)(t)$ computes the set of inputs and outputs. $p \in \{-, +\}$ is a polarity. $neg(-) = +$ and $neg(+) = -$.

$$
\begin{array}{lcl}
InOut(+)(\alpha) & = & \emptyset, \{\alpha\} \\
InOut(-)(\alpha) & = & \{\alpha\}, \emptyset \\
InOut(p)(t_1 \to t_2) & = & \mathit{let}\ i_1, o_1 = InOut(neg(p))(t_1)\ \mathit{in} \\
 & & \mathit{let}\ i_2, o_2 = InOut(p)(t_2)\ \mathit{in} \\
 & & i_1 \cup i_2, o_1 \cup o_2 \\
InOut(p)(t_1 \times ... \times t_n) & = & \mathit{let}\ (i_k, o_k = InOut(p)(t_k))_{k \in [1..n]}\ \mathit{in} \\
 & & \cup_{k \in [1..n]} i_k, \cup_{k \in [1..n]} o_k
\end{array}
$$

Given a set of variables V and a set of constraints C between them.
$I \subseteq V$ the set of inputs; $O \subseteq V$ the set of outputs. I and O not necessarily disjoint.

- $Out(a) = \{b \in O \mid C \vdash a \leq b\}$
- $In(a) = \{b \in I \mid C \vdash b \leq a\}$
- $IO(a) = \{b \in I \mid Out(a) \subseteq Out(b)\}$

For every input and output variable, computes its IO set.

Associate a unique key (a fresh variable) to every IO set.

Replace the relation $<$ by the relation between IO sets, that is:

if $IO(a) \subseteq IO(b)$, *with* a' *the key of* $IO(a)$ *and* b' *the key for* $IO(b)$, *then* $a' < b'$.

There is a canonical form (i.e., unicity) that minimises the number of variables.

Extra simplification

Some dependences can be removed.

Only keep dependences of the form $\alpha^p < \beta^q$ where the polarities p is $-$ or $+-$ and q is $+$ or $+-$.

It gives extremely short type signature in practice.

Open question: does it simplify more than existing simplification methods for type systems with sub-typing constraints?

The second question

Given a function, generate a statically scheduled, non recursive, non lazy implementation.

For a first-order language, it is a particular form of the Optimal Static Scheduling problem [5].

But this algorithm did not consider higher-order functions. E.g.:

```
let node f(g)(h)(x) =
  let rec y = g(x, y, z) and z = h(x, y, z) in (y, z)
```

In how many pieces `g` and `h` must be decomposed to generate a transition function for `f`?

One solution is to inline all higher order functions (i.e., they become macros). After that, the main function only call first order functions. Then, apply a algorithm to solve the OSS problem.

An intermediate approach

If we target a sequential langage like OCaml, full inlining is not always necessary.

Try to inline "on demand": only inline if it is not possible to generate a statically schedule function.

Do not necessarily inline higher order function. E.g., the function

```
let node easy_fixpoint(f)(i, x) =
  let rec o = run f (i fby o, x) in o
```

can be compiled into a transition function without knowing the implementation of `f`

Inlining "on demand" using causality types

Inline or not a function call according to the causality type.

Consider the function call $(f^{t_1}\ e^{t_2})^{t_3}$.

Let $o_{in} = Out(t_1) \cup Out(t_2)$ and $o_{out} = Out(t_3)$.

Do not inline if for all $i \in o_{in}, o \in o_{out}$:

$$(o \not< i) \wedge (IO(i) \subseteq IO(o))$$

If the function is not inlined, add the dependence $i < o$, that is, consider the function call to be strict.

This strategy is correct for a first order language (that is, Lustre).

Conjecture: it is correct and complete for a higher-order function too.

Examples

Right after the causality analysis, a pass annotates some of the function call to be “inlined”.

```
let node filter(n)(h)(k, u) = udot where
  rec udot = n *. (k *. u -. f)
  and f = inline forward_euler(h)(n, 0.0, udot)

  ...
let node gpid(int)(filter)(h)(p, i, d, u) = c where
  rec c_p = p *. u
  and i_p = inline run (inline (int h))(i, 0.0, u)
  and c_d = inline run (inline (filter h))(d, u)
  and c = c_p +. i_p +. c_d

let node pid_forward_no_filter =
  (inline gpid forward_euler derivative h (p, i, d, u))
```

Example: two mutually recursive streams

```
let node int(h, x0, xprime) = x where
  rec x = x0 -> pre(x +. h *. xprime)

let node main () = sin, cos where
  rec sin = int(0.1, 0.0, cos)
  and cos = int(0.1, 1.0, -. sin)
```

One of the two function calls has to be inlined. The one to be inlined is arbitrary; here the order in which equations are traversed.

```
let main () = sin, cos where
  rec sin = int(0.1, 0.0, cos)
  and cos = inline int (0.1, 1.0, -. sin)
```

Extra notes (to be continued).

The curse of non linear functions

```
let node fix(f)(x) = o where rec o = run f (x, o)
let node twice(f)(x) = o where rec o = run f (run f (x))
let node f(x) = x + 1
val f : {}. 'a -> 'a

val twice : {'a < 'b}. ('b -> 'a) -> 'b -> 'a
```

The type of `twice` says that the output of `f` must not depend on its input whereas it does not appear in any recursive stream equation!

This is a consequence of the contravariance rule and the fact that the sub-typing rule uses a strict order. We get the same type by writting:

```
let node twice(f)(x) = o2 where
  rec o1 = run f (x)
  and o2 = run f (o1)
val twice : {'a < 'b}. ('b -> 'a) -> 'b -> 'a
```

Union/intersection types? Use non strict order for sub-typing?

Atomic functions

```
let node twice_f(x) = twice(f)(x)
>let node twice_f(x) = twice(f)(x)
>                              ^^^
Causality error: This expression has causality type
'b -> 'c, whereas it should be less than 'd -> 'e
Here is an example of a cycle:
d < b; b < c; c < e; e < d
```

One way to impose the strongest constraints on an input function is to consider it to be atomic, that is, as if all of its outputs would depend on all of its inputs.

```
let node twice_atomic(f)(x) = o where
  rec o = run (atomic f) (run (atomic f) (x))

let node twice_atomic_f(x) = twice_atomic(f)(x)

val twice_atomic : {'a < 'b}. ('a -> 'b) -> 'b -> 'b
val twice_atomic_f : {}. 'a -> 'a
```

The typing rule for atomic values

$$
\begin{aligned}
\texttt{skeleton}(C)(\alpha)(\alpha') &= \alpha, C + [\alpha < \alpha'] \\
\texttt{skeleton}(C)(\alpha)(t_1 \to t_2) &= \mathit{let}\ t_1', C_1 = \texttt{skeleton}(C)(\alpha')(t_1)\ \mathit{in} \\
&\quad \mathit{let}\ t_2', C_2 = \texttt{skeleton}(C_1)(\alpha)(t_2)\ \mathit{in} \\
&\quad t_1' \to t_2', C_2 + [\alpha' < \alpha] \\
\texttt{skeleton}(C)(\alpha)(t_1 \times t_2) &= \mathit{let}\ t_1, C_1 = \texttt{skeleton}(C)(\alpha)(t_1)\ \mathit{in} \\
&\quad \mathit{let}\ t_2, C_2 = \texttt{skeleton}(C_2)(\alpha)(t_2)\ \mathit{in} \\
&\quad t_1' \times t_2', C_2
\end{aligned}
$$

$$
\frac{t', C' = \texttt{skeleton}(\emptyset)(\alpha)(t)}{C + C', H[f : t] \vdash \texttt{atomic}\ f : t'}\ (\text{atomic})
$$

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.

Gérard Berry.

The constructive semantics of pure esterel, draft, version 3.
Draft book. Available at:
`http://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.pdf`, 2002.

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.

Pascal Cuoq and Marc Pouzet.

Modular Causality in a Synchronous Stream Language.
In *European Symposium on Programming (ESOP'01)*, Genova, Italy, April 2001.

Marc Pouzet and Pascal Raymond.

Modular Static Scheduling of Synchronous Data-flow Networks: An efficient symbolic representation.
In *ACM International Conference on Embedded Software (EMSOFT'09)*, Grenoble, France, October 2009.